GY20-0591-1

**Type III Class A Program**

# Control Program-67/Cambridge Monitor System (CP-67/CMS) Version 3.1
# CMS Program Logic Manual
# Program No. 360D-05.2.005

The Cambridge Monitor System (CMS) is a conversational monitor system that provides a comprehensive, easy-to-use set of programs (commands) giving the CMS user a wide variety of functions, including the ability to create additional commands or subsystems to satisfy his special requirements.

This manual provides a detailed description of the internals of CMS.

# REFERENCES

The following documents are referenced in this manual:

*CP-67/CMS User's Guide*, Form GH20-0859
*CP-67/CMS Installation Guide*, Form GH20-0857
*SRL/I: A String Processing Language*, Form 320-2005,
IBM Cambridge Scientific Center
*SCRIPT: An Online Manuscript Processing System*,
Form 320-2023, IBM Cambridge Scientific Center
*OS/360 – Supervisor & Data Management*
Macro-Instructions, Form GC28-6647

Second Edition (October 1971)

This Type III Program performs functions that may be fundamental to the operation and maintenance of a system. It has not been subjected to formal test by IBM.

Until this program is reclassified, IBM will provide for it: (1) Central Programming Service, including design error correction and automatic distribution of corrections; and (2) FE Programming Service, including design error verification, APAR documentation and submission, and application of Program Temporary Fixes or development of an emergency bypass when required. IBM does not guarantee service results or represent or warrant that all errors will be corrected.

The user is expected to make the final evaluation as to the usefulness of this program in his own environment.

THE FOREGOING IS IN LIEU OF ALL WARRANTIES EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

This edition applies to Version 3, Modification Level 1, of Control Program-67/Cambridge Monitor System (360D-05.2.005) and to all subsequent versions and modifications until otherwise indicated in new editions or Technical Newsletters.

Changes are continually made to the information herein; therefore, before using this publication, consult the latest System/360 SRL Newsletter (GN20-0360) for the editions that are applicable and current.

Copies of this and other IBM publications can be obtained through IBM branch offices.

A form has been provided at the back of this publication for readers' comments. If this form has been removed, address comments to: IBM Corporation, Technical Publications Department, 1133 Westchester Avenue, White Plains, New York 10604.

# CONTENTS

# ILLUSTRATIONS

Figure

## SECTION 1: INTRODUCTION TO CMS

The Cambridge Monitor System (CMS) is a single-user, conversational operating system. It is designed to allow full use of an IBM System/360 through a simple terminal-orientated command language. CMS gives the user a full range of capabilities - creating and managing files, compiling and executing problem programs, and debugging - requiring only the use of his remote terminal.

CMS also provides a batch version that will process non-conversational user jobs. Job control cards that are imbedded within the input stream dictate which batch function will be executed.

### COMPONENTS & FACILITIES PROVIDED BY CMS

### CMS Commands

A CMS command is (1) the name of a program resident in the nucleus or on any CMS disk, or (2) the name of a file containing other CMS commands. CMS commands fall into seven categories: file creation, maintenance, and manipulation; language processors; execution control; debugging facilities; utilities; control commands; and library facilities.

### File Creation, Maintenance and Manipulation

The file handling commands of CMS allow the user to create and modify disk files via a context editor as well as to rename, copy, combine, split, update, erase, and print disk files. The user can also print or punch files on unit record equipment. He can create disk files from cards read via the card reader. He also can format text information for letters, documents, or reports by utilizing the SCRIPT facility of CMS.

### Language Processors

Some Operating System/360 Language Processors are used under CMS. These include Assembler (F), FORTRAN IV (G), and PLI (F). The Assembler produces object programs that may be executed under either CMS or OS, depending on the macros used in the source program. Special file handling routines for macro libraries are included. The FORTRAN and PL/I compilers also produce OS-compatible object programs. Diagnostics from the OS compilers are printed at the terminal unless suppressed by the user or directed to disk. Because the CMS file system does not provide as many access methods as OS/360, some features of PL/I are not supported at program execution time.

SCRIPT, a text processor, is also provided. There are two additional processors available as Type III programs from the IBM Program Information Department; they are SNOBOL, a string processing language, and BRUIN, an interpretive language. BRUIN, BRown University INterpreter, was adapted from the OS version of BRUIN developed at Brown University, Providence, Rhode Island. BRUIN provides two modes of operation: a desk calculator mode and a stored program mode.

## Execution Control

The execution control commands allow the user to load his programs from single object decks called TEXT files (the filetype TEXT is reserved for relocatable object programs) or from a program library. The user can pass a list of parameters to his program from the terminal and specify the point at which execution is to begin. To bypass the relocating loader for each execution of the program, he can create a file consisting of an image of the portion of core storage containing his program and load that non-relocatable copy back at any time. Since the loading commands can be accessed by executing programs, overlay structures may be set up and dynamic loading can occur.

During program execution under CMS the user can fully interact with his programs. For example, in FORTRAN G under OS/360 a READ to logical unit 5 reads the SYSIN device and a WRITE to logical unit 6 writes to the SYSOUT device; under CMS, a READ to 5 reads the operator's console (remote terminal under CP) and a WRITE to 6 writes the console.

The user also has the facility to create a procedure that is a series of commands and then to execute these commands by typing a single instruction; logic statements can be placed in the file with the commands so that the order of command execution may be dynamically set or altered. This capability is called EXEC and allows a user to develop his own command language or sets of procedures. EXEC also allows for the passing of variable arguments from the terminal as well as between EXEC files, since EXEC files can be nested and/or recursive.

## Debugging Facilities

A permanent, nucleus-resident debugging facility is available to the user. It allows stoppage of programs at predetermined points and examination of registers, PSW, and storage, and permits modification of these if it is desired. This information may be typed out at a user's terminal or printed offline. A program interrupt gives control to DEBUG, as does the external interrupt caused by the EXTERNAL console function. The user may also employ the program tracing routines, which record all SVC transfers, or just those SVC's in which an error return is made.

## Utilities

The utility functions in CMS provide tape copying facilities, disk file comparison, a disk file sort, and the dumping of files either by name onto the console or by cylinder locations onto the offline printer, as well as the facility to dump files to tape and reload them onto disk. There are commands also for converting files of fixed-length records to variable-length records, for converting BCDIC files to EBCDIC, and for obtaining statistics on file space.

## Control Commands

There are other commands that give the user the facility to suppress the typeout at his terminal, to restore typing at his terminal once the typeout is suppressed, and to kill program execution. The user can redefine the logical line-end character, the character delete and line delete characters, as well as the blip character that notifies him of CPU utilization. He can also rename any command and define his own abbreviations to be used.

## Library Facilities

CMS provides library facilities for program libraries. The user can generate his own libraries or add, delete, or list entries in existing libraries. He can also specify which libraries to use for program assemblies as well as program execution.

## Page Release Facility

Certain CMS routines include a page release facility. This means that following a successful completion and before returning to the user or caller, the routine references NUCON and turns a page release flag on. When the routine then returns to INIT, INIT checks this flag. If it is on, INIT issues a diagnose X'10' to CP to release user pages from X'12000' up to the value in LOWEXT.

For the user to prevent this release of pages, the CMS VSET RELPAG OFF command should be issued. The commands that have the page release facility are: ASSEMBLE, CEDIT, COMBINE, COMPARE, EDIT, FORTRAN, MACLIB, MAPPRT, PLI, SORT, SPLIT, TAPE, TXTLIB, and UPDATE.

## CMS Core Requirements

CMS has a prerequisite for 80K bytes of virtual memory for the nucleus, transient area, and loader tables. At login time, core space for user file directories is allocated dynamically as required. The rest of core storage is available to user programs.

## CMS Batch Monitor

As well as being a conversational monitor, CMS provides a batch facility for running CMS jobs. The CMS batch monitor accepts a job stream from a tape unit or from the card-reader and writes the output either on tapes, the printer, or the card-punch. The job stream can consist of a System/360 Operating System SYSIN job stream with FORTRAN (G), and Assembler (F) compile, load, and go jobs or it can consist of CMS commands along with control cards and card decks for compile, load, and go jobs for all the CMS supported compilers.

Just as the conversational CMS does, the batch monitor can run from either a virtual machine or a real machine. Under CP, it can be used as a background monitor along with other conversational CMS users.

To eliminate the possibility of one job modifying the CMS batch monitor's nucleus in such a way as to affect the next job, the batch monitor is re-IPLed before each job begins. Files can also be written onto the batch monitor's primary disk and then punched or printed, such as files written by FORTRAN programs; these files should be of limited size and considered as temporary, as they are erased at the completion of each job.

MACHINE CONFIGURATION

Whether running on a real (see Note below) or a virtual machine, CMS expects the following machine configuration:

| Device | Virtual Address | Symbolic Name | |
|---|---|---|---|
| 1052 | 009 | CON1 | console |
| 2311, 2314 | 190 | DSK1 | system disk (read-only) |
| 2311, 2314 | 191** | DSK2 | primary disk (user files) |
| *2311, 2314 | 192** | DSK3 | temporary disk (work space) |
| *2311, 2314 | 000** | DSK4 | A disk (user files) |
| *2311, 2314 | 000** | DSK5 | B disk (user files) |
| *2311, 2314 | 19C** | DSK6 | C disk (user files) |
| 1403 | 00E | PRN1 | line printer |
| 2540 | 00C | RDR1 | card reader |
| 2540 | 00D | PCH1 | card punch |
| *2400 | 180 | TAP1 | tape drive |
| *2400 | 181 | TAP2 | tape drive |

at least 256K bytes of core storage, 360/40 and up

*The 2311 or 2314 for the temporary disk, the A, B and C disks, and the two 2400 tape drives are optional devices; they are not included in the minimum configuration.

**The specified virtual addresses may be changed at any time by the CMS LOGIN command.

Note: For use on a real machine not having this I/O configuration, the device addresses can be redefined at 'load' time.

Under CP, of course, these devices are simulated and mapped to different addresses and/or different devices. For instance, CMS expects a 1052 printer-keyboard operator's console, but most remote terminals are 2741's; CP handles all channel program modifications necessary for this simulation.

CMS allows the user to add his own programs for I/O devices not supported by the standard system. CMS also provides for dynamic specification of SVC routines.

The system disk, located at address 190, is read-only and contains the CMS system commands. These system programs are physically divided into two groups: nucleus functions and disk-resident command modules. The nucleus programs are loaded into main storage during initial program load (IPL) and remain resident throughout system operation. The disk resident modules are loaded into main storage only when their services are needed. Certain disk resident programs are loaded into the transient area. The primary disk, 191, is a read-write disk and normally is the first user disk. Files that the user wishes to retain for use across terminal sessions are stored on one of the user's disks. Information stored on the primary disk remains there until it is deliberately erased or destroyed by the user. Commands and input files are entered into the system from the console (that is, the terminal located at address 009). Output files, program results, and error and prompting messages are directed from within CMS to the console. The card reader, located at address 00C, may be used as the input medium for files, source decks, and data to be processed by user programs. The card punch, address 00D, may receive user output files, processor object decks, and various other types of data. The printer, address 00E, may receive user program results, and Assembler, FORTRAN and PLI listings. A tape, located at address 181, may be used in Dump/Restore operation and as an input/output medium for files.

## SECTION 2: INTERNAL CHARACTERISTICS

This section describes the internal characteristics of CMS, including the way in which control is passed among the programs that make it up, the manner in which it manages main storage, disk space, and files, and the nature of its I/O operations.

### INTERNAL LINKAGE SCHEME

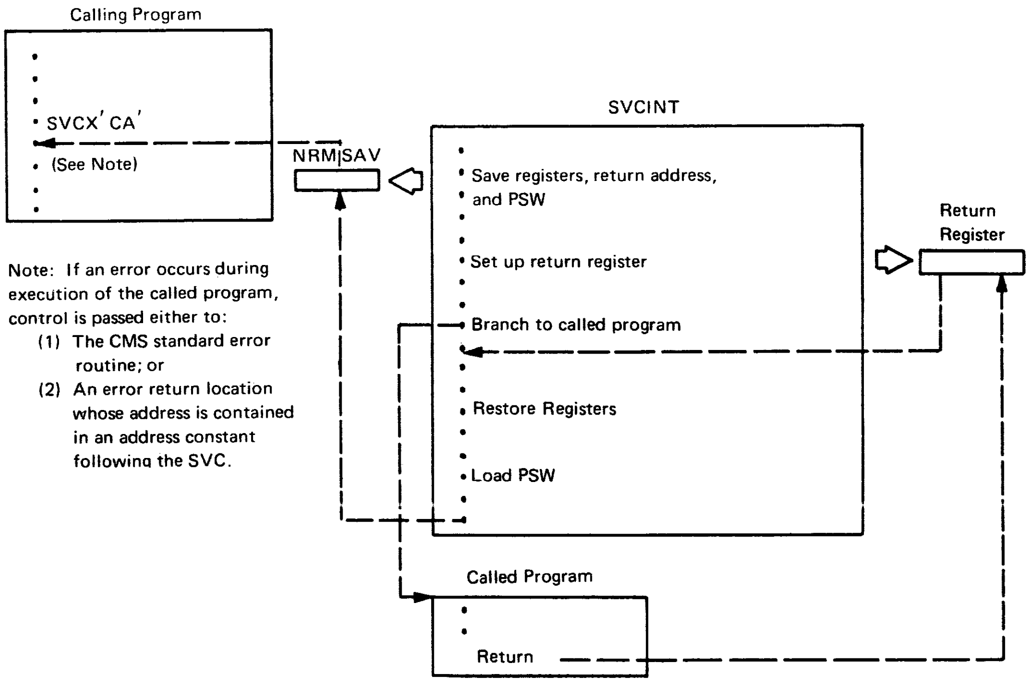
In CMS, control is generally passed from one program to another (for example, from a command program to a function program) by means of a supervisor call (SVC) instruction. When one CMS program requires the services of another, it issues a special SVC of the form SVC X'CA'. (This SVC may be followed by a 4-byte address constant containing an error return address. ) Associated with the SVC is a parameter list that identifies, by name, the program whose services are desired (that is, the called program). The execution of the SVC instruction causes an interruption, and control passes to a SVC interruption handler (SVCINT). When SVCINT receives control, it saves (1) contents of the calling program's registers and (2) the SVC old PSW. This PSW contains the address within the calling program to which control is to be returned when execution of the called program is complete. Having saved these items, SVCINT sets up a return register with an address pointing to a location within itself. This location is where the called program is to return control. It then branches to the called program. When execution of the called program is complete, it returns to SVCINT through the return register.

In an error occurred during execution of the called program, SVCINT returns control:

- To the error return address, if one followed the SVC X'CA' instruction.
- To the CMS standard error routine (STDERR), if no error return address was given after the SVC in the calling program.

If no errors occurred during execution of the called program, SVCINT restores the calling program's registers, and loads the saved PSW. Control returns to the calling program at the executable instruction following the SVC X'CA' instruction. Figure 1 shows how this scheme works.

Control within CMS may be passed via SVC's to a level of twenty calls. For example, one CMS program may call another (first-level call), which may call another (second-level call), which may call another (third-level call), etc. SVCINT intercepts each of these nested calls and takes steps to ensure proper return to the calling program. It does this for each call by: (1) storing the SVC old PSW that results from the interruption and the contents of the registers as they exist at the time for the interruption in a last-in, first-out list, and (2) passing control to the called program. Upon return from the last called program (the program at the lowest level will return to SVCINT first), SVCINT restores the registers with the saved register data stored in the last entry in the list, deletes that entry from the list, and loads the saved PSW. Control is thereby returned to the program at the next higher level. When execution of this program is complete, SVCINT follows a similar procedure to return control to the program that called it. Figure 2 shows how this scheme works.



Note: If an error occurs during execution of the called program, control is passed either to:
(1) The CMS standard error routine; or
(2) An error return location whose address is contained in an address constant followina the SVC.

Figure 1. Internal linkage scheme

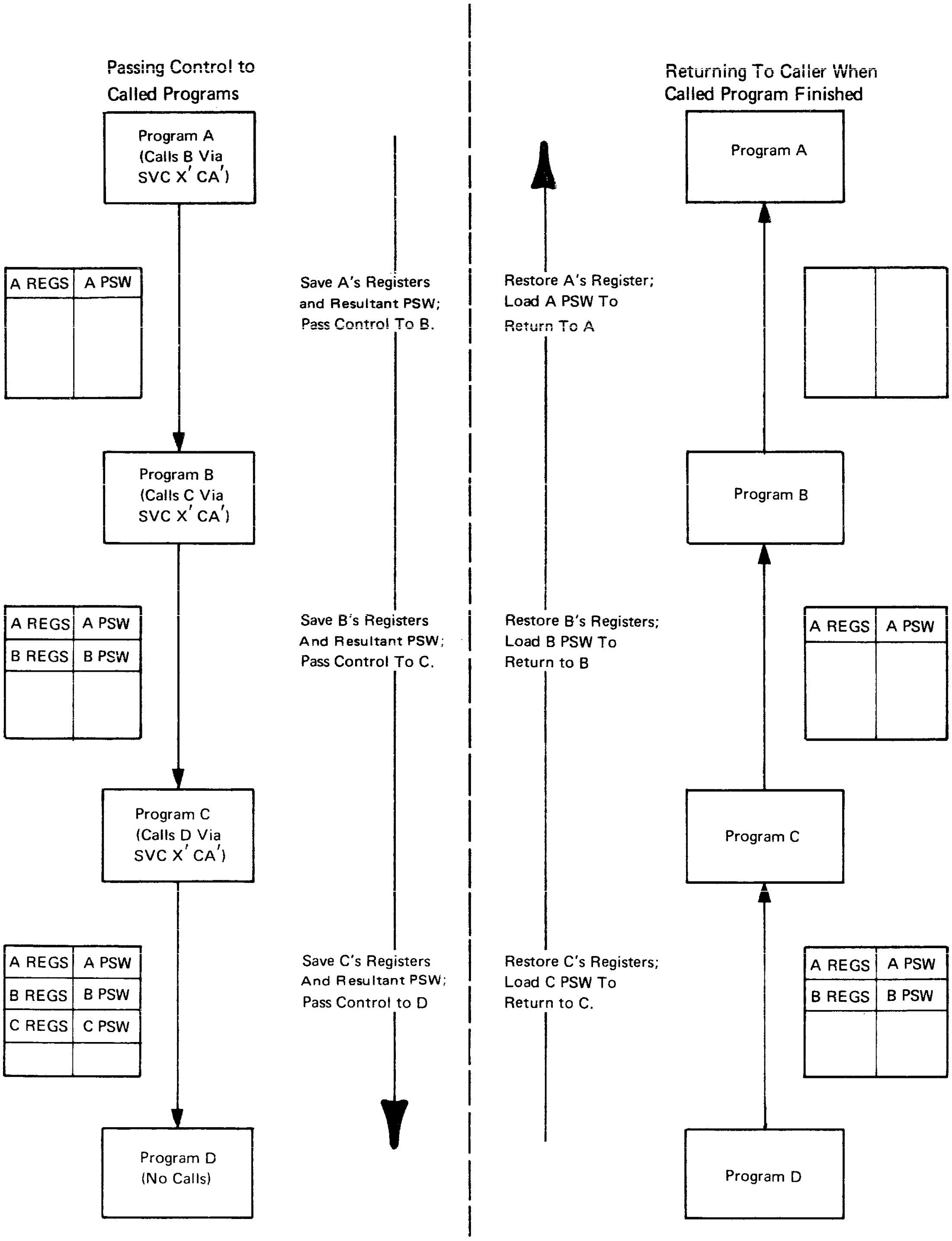
## RETURN CODE CONVENTIONS

When a program, called via an SVC X'CA' instruction, returns to the calling program, register 15 contains a positive, negative, or zero code. A positive code indicates that an error occurred during the execution of the called program. A negative code indicates that control was never passed to the called program. (This might occur if, for example, the user incorrectly types a command from the terminal.) A zero code indicates that the called program was executed successfully.

SVCINT is the only CMS program that returns negative error codes.

## MAIN STORAGE MANAGEMENT

Main storage is composed of five main areas: the nucleus, free storage area, a transient area, user program area, and a loader tables area. The nucleus contains the core resident portion of CMS; it begins at Page 0 of main storage and extends upward into Page 13. Free storage is that portion of main storage between the end of the nucleus and the start of Page 17 of main storage. Page 17 is a transient area into which certain CMS commands are loaded. The user program area, which starts at Page 18 (or above), is the area into which the user programs are loaded. The last two pages prior to the end of core are reserved for the loader tables used by the CMS commands LOAD, USE, and REUSE for loading in programs. (See Figure 3.)



Passing Control to Called Programs
Program A (Calls B Via SVC X' CA')
A REGS
A PSW
Save A's Registers and Resultant PSW; Pass Control To B.
Program B (Calls C Via SVC X' CA')
A REGS
A PSW
B REGS
B PSW
Save B's Registers And Resultant PSW; Pass Control To C.
Program C (Calls D Via SVC X' CA')
A REGS
A PSW
B REGS
B PSW
C REGS
C PSW
Save C's Registers And Resultant PSW; Pass Control to D
Program D (No Calls)
Returning To Caller When Called Program Finished
Program A
Restore A's Register; Load A PSW To Return To A
Program B
Restore B's Registers; Load B PSW To Return to B
A REGS
A PSW
Program C
Restore C's Registers; Load C PSW To Return to C.
A REGS
A PSW
B REGS
B PSW
Program D

Figure 2. Internal linkage scheme - multiple levels

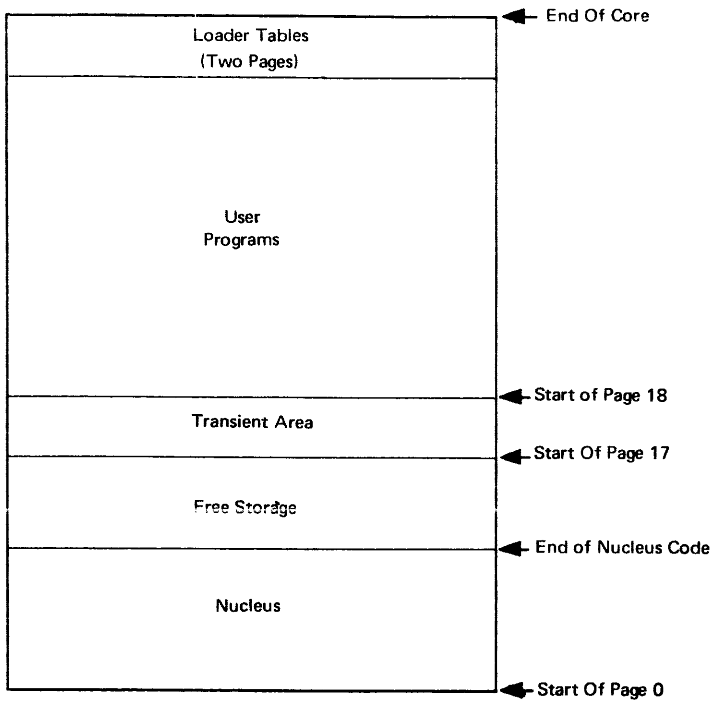


Figure 3. CMS main storage layout

Main storage management allocates and keeps track of the free storage that is available. Free storage is used by the various CMS programs requiring blocks of main storage for temporary use (for example, as buffer space into which data blocks are read for processing). When a program requires main storage, it calls the FREE storage management program indicating the number of double words required. The storage management program allocates the required storage and returns a pointer to its starting location to the calling program.

When a program has finished with such temporary storage, it returns it to a free storage via the FRET storage management program, indicating the size and starting location of the block being returned.

The storage management programs keep track of free storage through a series of pointers, the first of which originates in the storage cell FREELIST. FREELIST always contains a pointer to the free storage block at the lowest address. The free storage block pointed to from FREELIST starts with two fields; the first contains a pointer to the second free storage block (that is, the free storage block that starts at the next higher address); the second field contains the size (in bytes) of the block itself. Similar fields exist for the second, third, and subsequent free storage blocks. Thus, free storage blocks are always chained from low to high addresses. Figure 4 illustrates this concept.

Main storage management works in the following way. The first request for main storage causes Page 16 to be made available as free storage. A block of storage of the requested size is allocated to the caller from the high-numbered end of Page 16, FREELIST is initialized to point to the beginning (low-numbered end) of Page 16; the pointer and size fields

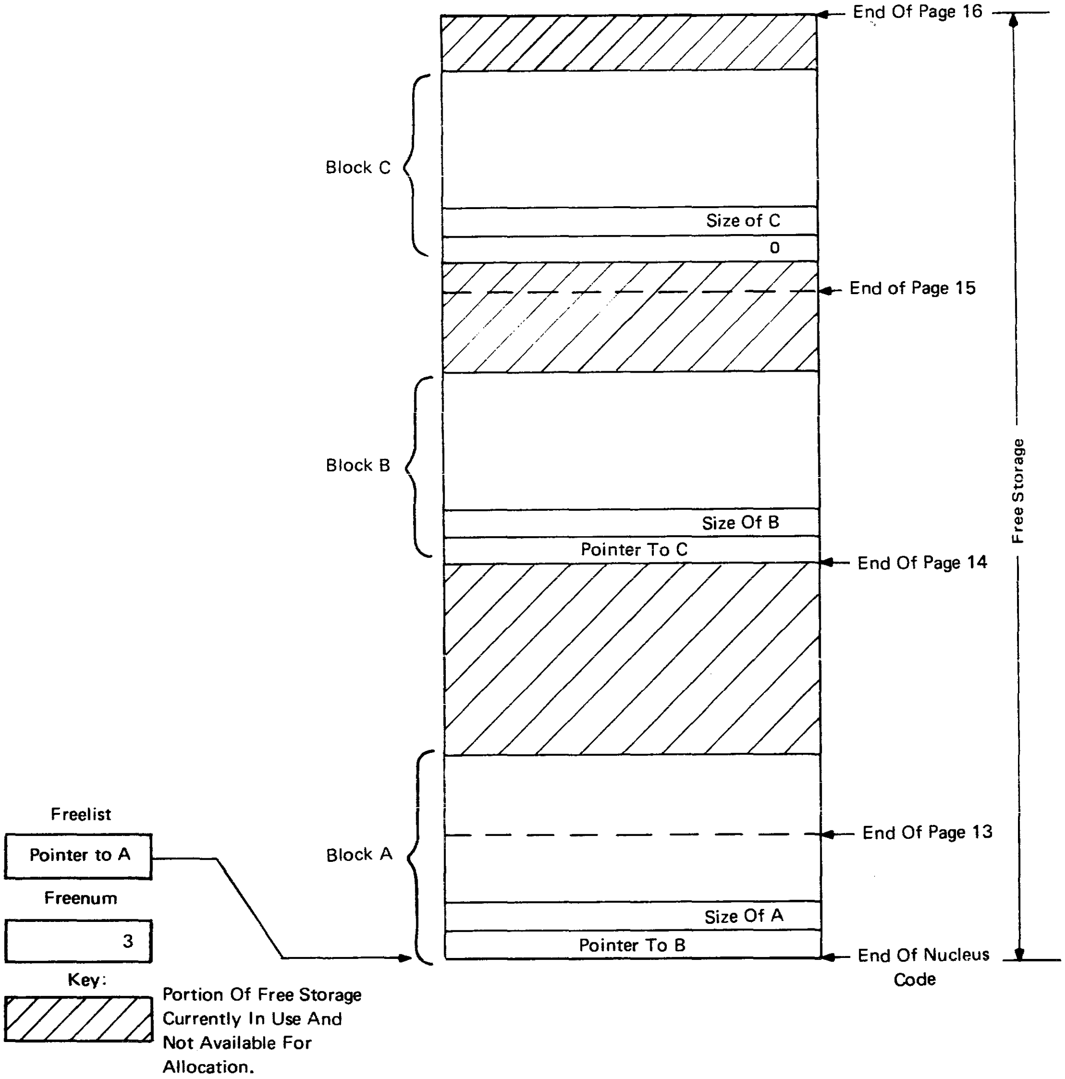


Figure 4. Example of chaining of free storage blocks

at the beginning of Page 16 are initialized appropriately, and FREENUM is set to one. (See Figure 5.) A second request for free storage causes the next available lower block near the end of Page 16 to be allocated to the caller, and the size field at the beginning of the one and only free storage block at the beginning of Page 16 is decremented by the number of bytes just allocated. (See Figure 6.)

Assume now that three additional requests for free storage blocks are satisfied (see Figure 7), but that after the last of these, the program that received the second block no longer needs it and returns it for use by other programs. The released block is chained into the free storage list as the second entry, and FREENUM is appropriately updated. (See Figure 8.)

For each subsequent request for free storage, the storage management programs scan the chain of free storage blocks for one that is equal in size to that requested. If one is found, it is immediately allocated to the caller, the chain of free storage blocks is broken

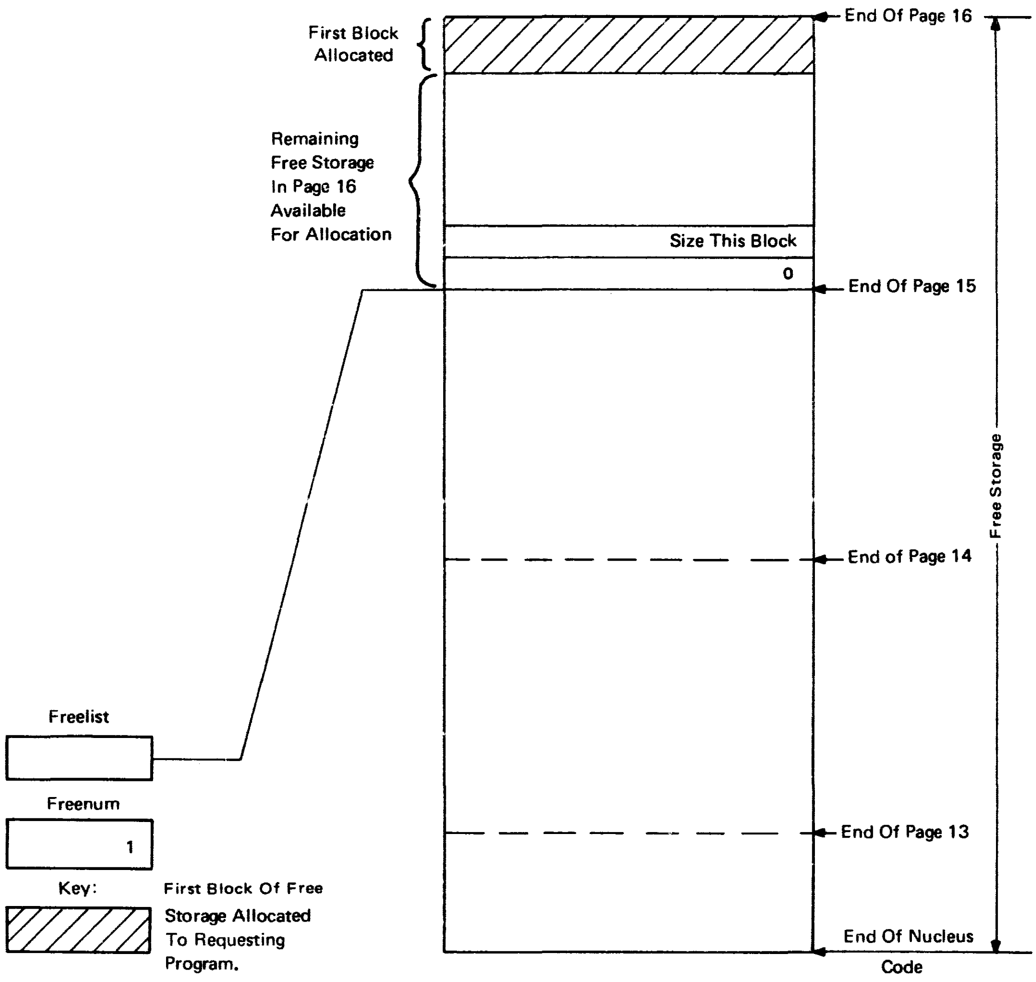


Figure 5. Example of free storage after first request

and relinked to delete the allocated block from the free storage list, and FREENUM is decremented by one. If a free storage block equal in size to that requested cannot be found, but one or more free storage blocks larger in size than that requested are found, storage is allocated to the caller from the high-numbered end of the last larger block that was found, and the size of that block is decremented appropriately.

If neither a matching nor a larger block of free storage is available, the EXTEND storage management program allocates another large block of free storage as available between the end of the CMS nucleus and the start of Page 17. This large block (for example, Page 15, later Page 14, then whatever free storage may be available at the end of block 13) is merged properly into the free storage chain (that is, FREELIST points to its beginning), and the caller's previously unsatisfied request is then handled in accordance with the revised chain of available storage. (Note in Figure 7 that Page 15 had to be obtained to allocate the third block, and later Page 14 was needed for allocation of the fifth block. )

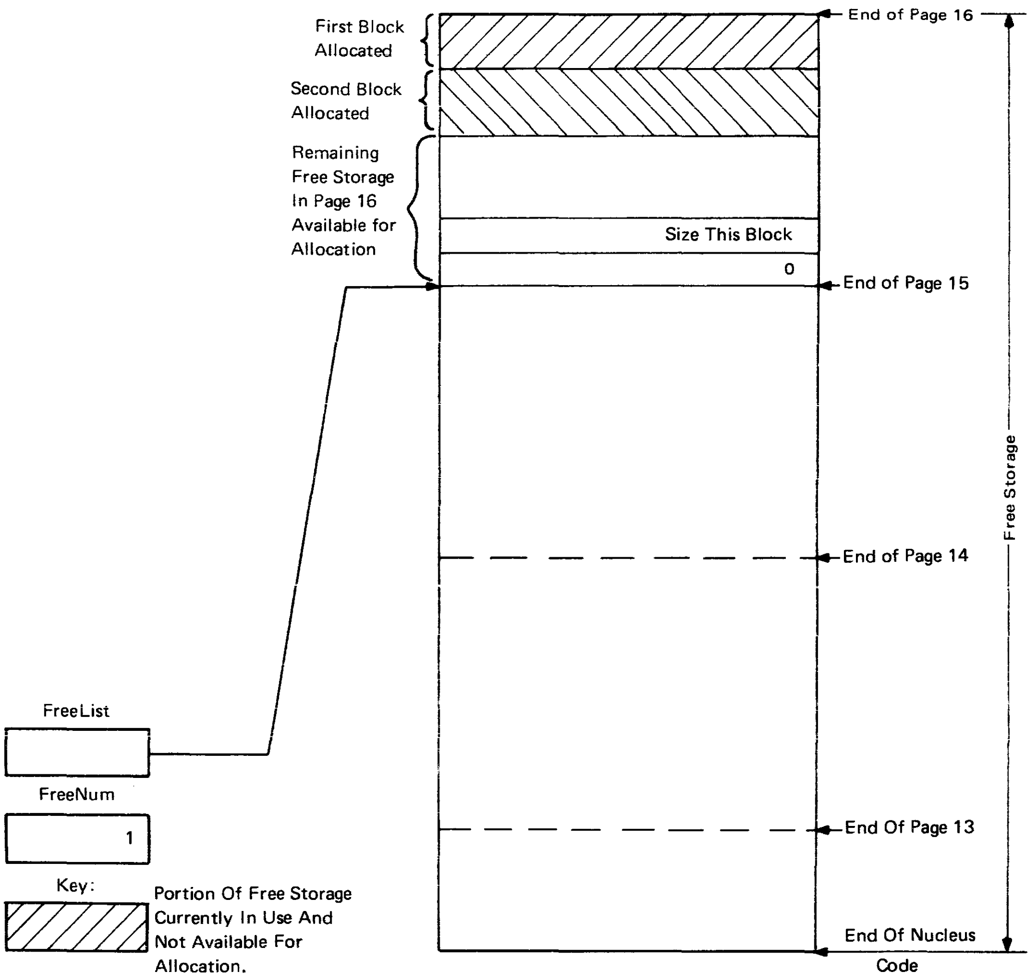


Figure 6. Example of free storage after second request

When a program returns a block of storage via FRET, it is linked into the chain in its proper place, and FREELIST, FREENUM, and the necessary pointer and size fields are appropriately updated as necessary. If the block returned abuts a free storage block (either above, below, or both), they are combined into a single block with pointers, sizes, FREELIST, and FREENUM being appropriatelv adjusted as needed to maintain proper chain sequence and block sizes.

If there is not sufficient free storage available in the free storage area following the nucleus, pages are allocated by EXTEND and merged into the chain as needed, one at a time, from the end of the user program area, just below the loader tables. If a page (or more) of such storage is returned, this storage is also "given back". Care is taken by the storage management programs to ensure that any such storage allocated from the end of the user program area does not run into conflict with loaded programs, storage allocated by the GETMAIN procedure, COMMON area, and the like.

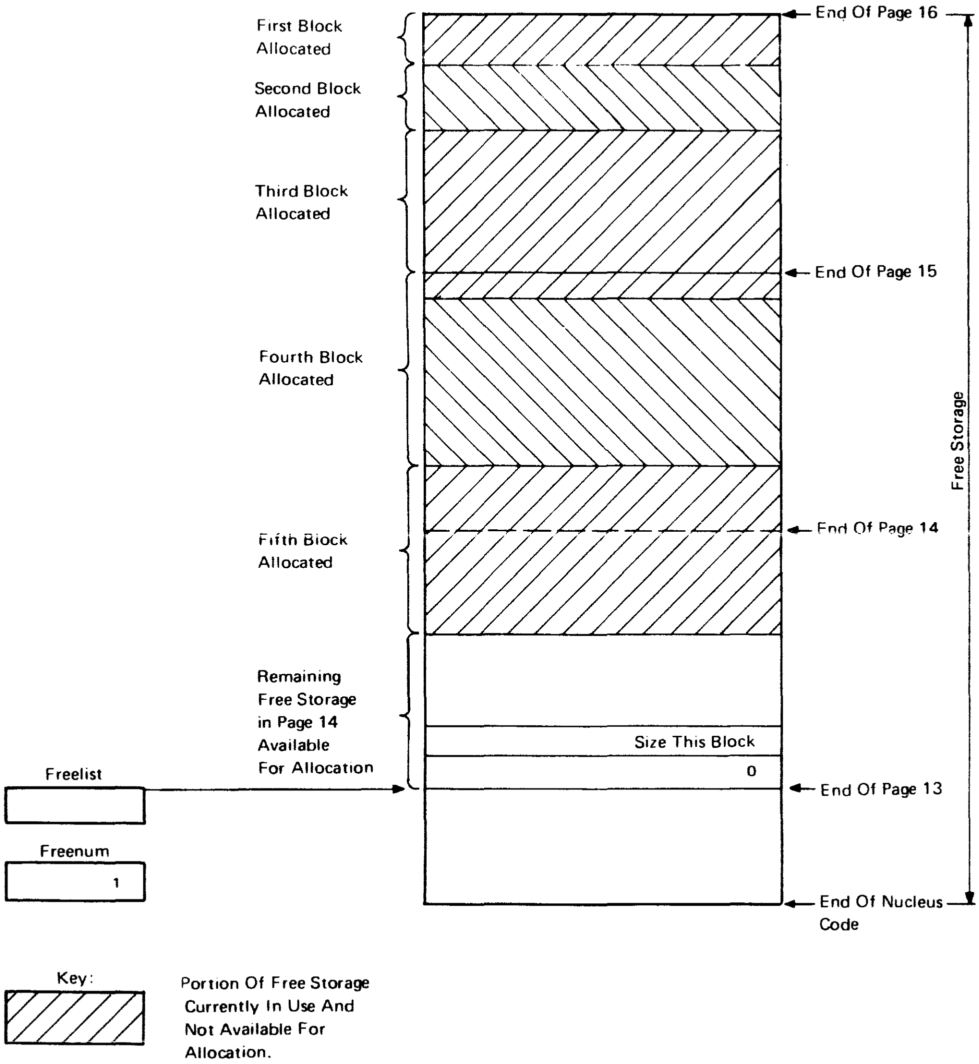


Figure 7. Example of free storage after five requests

## CMS NUCLEUS

A key portion of the CMS nucleus, located near the beginning of Page 0, is the Nucleus Constant Area Table (NUCON). NUCON contains (1) several parameters used by the loading routines and storage management programs, (2) a table of device addresses, and (3) a table of address constants giving the location of certain nucleus-resident CMS tables (for example, FVS, IADT), and routines called via BALR instructions (for example, FREE & FRET). Some of the constants and parameters in NUCON, with their meaning and their usual values on a 256K System/360, are as follows:

CORESIZ The size of core (computed at the beginning of a terminal session — for example, 40000 hex = 262144).

USFL User first location (12000 hex)

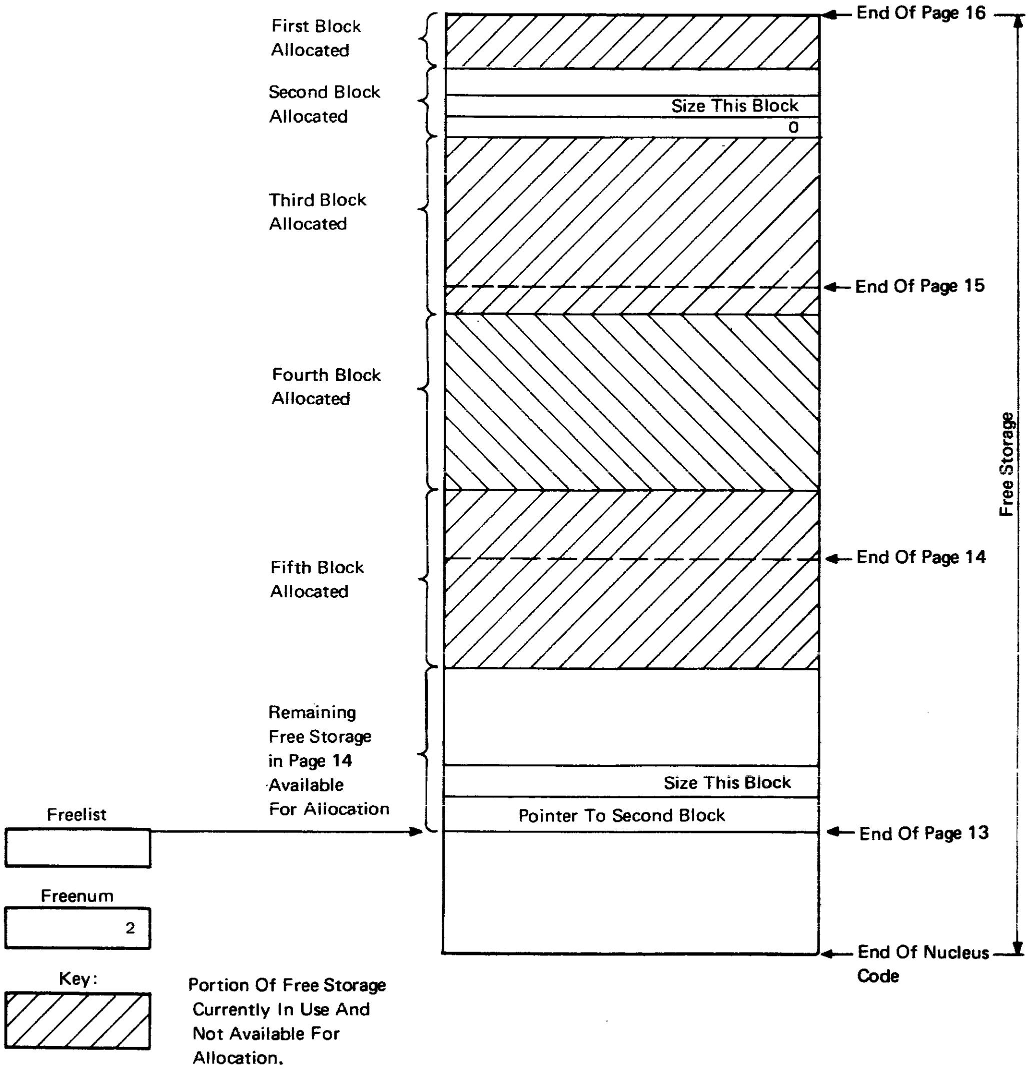


Figure 8. Free storage after second block has been returned

STADDR Address to start user execution (frequently 12000 hex)

LDRTBL End of loader tables (40000 hex = 262144)

LOCCNT Next core location to start (or resume) loading

LOWEXT Lowest core address given out by EXTEND routine of the storage management programs (starts at 3E000 hex)

HIMAIN Highest address given out by GETMAIN program

CONGEN An area used by the typewriter routines (WAITRD, TYPLIN, CONSI, etc.) for input and output areas, etc.

SYSREF Start of tables of address constants of certain CMS tables and routines.

To refer to NUCON and its contents, the CMSYSREF macro can be used. The CMSYSREF macro contains three main parts: Absolute, NUCON, and SYSREF. The Absolute section gives the displacements of absolute locations in page 0. The NUCON part gives relative displacements for the miscellaneous parameters and device addresses. The SYSREF part of the CMSYSREF macro gives relative displacements of the address constants starting at SYSREF. In both the NUCON and SYSREF parts, the label of each parameter constant begins with D. For example, the label for the displacement of LOWEXT in NUCON is DLOWEXT.

Note that NUCON and SYSREF are always known locations to the CMS loading programs. Thus, any disk resident program, by procedures outlined above using the CMSYSREF macro and the NUCON and SYSREF V-constants, can indirectly reference any quantities which are specified in the NUCON and SYSREF portions of the CMS Nucleus Constant Area.

For a complete list of the NUCON and SYSREF values, see the CMSYSREF macro as shown in Figures 9 and 10.

## CMS DISK FILE ORGANIZATION AND MANAGEMENT

CMS files are stored on read-write and read-only disks. The CMS file management scheme uses various commands, functions, routines, and tables to structure and keep track of these files.

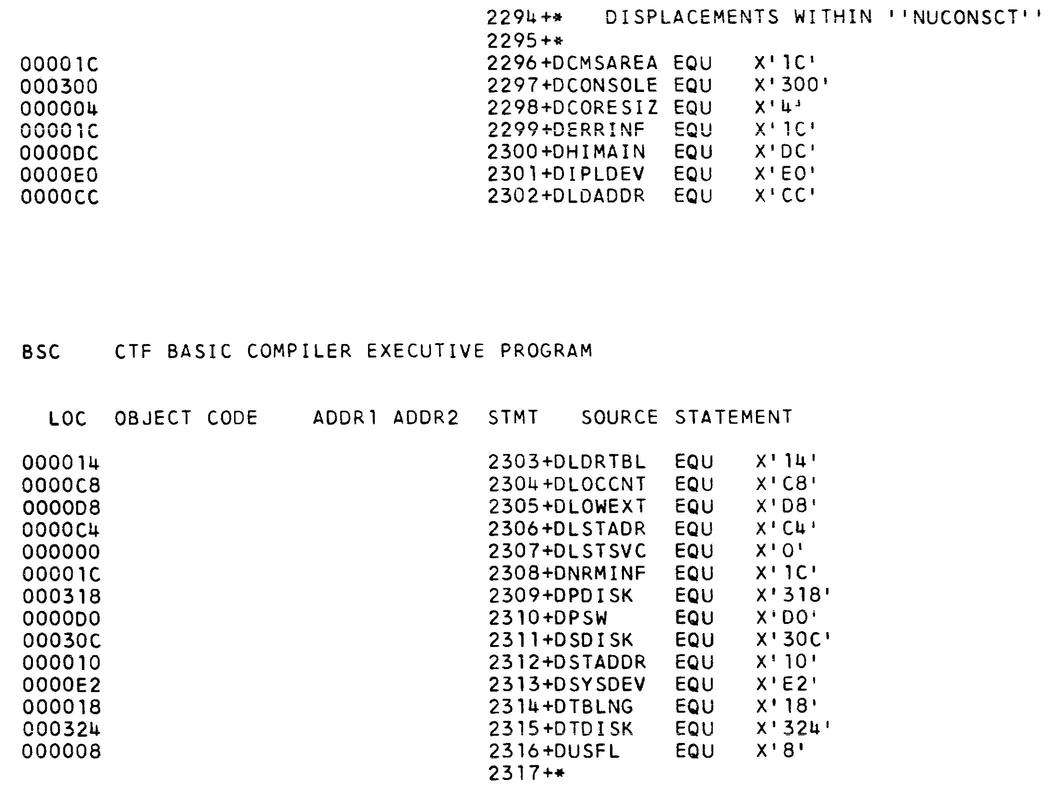
CMS handles up to six disks, one of which is the read-only System Disk (S-Disk), normally having a device-address of 190. Up to five other disks can be active at any given time, each of them either read-write or read-only. Usually one of these is a read-write disk called the P-Disk or Primary disk (normally with a device-address of 191), on which a user keeps his files, which are preserved by the CMS file system until they are purposely updated, erased, or replaced by the user or various command programs.

Each disk can be either a 2311 or 2314, from one to 203 cylinders. Other pertinent parameters for each disk are as follows:

- A 2311 holds a maximum of 8120 800-byte physical records
- A 2314 holds a maximum of 30,448 800-byte physical records
- Up to 3500 files may reside on a given disk
- Any given file can be as large as either
  1. 16060 800-byte physical records

     <u>or</u>
  2. 65533 logical records (items)

The following pages describe the file management scheme used by CMS for files on read-write and read-only disks. For further information, see the sections "Disk Space Management", "Active File Table (AFT) Management", and "Active Disk Table (ADT) Management".



```
                                          2294+*    DISPLACEMENTS WITHIN ''NUCONSCT''
                                          2295+*
00001C                                    2296+DCMSAREA EQU   X'1C'
000300                                    2297+DCONSOLE EQU   X'300'
000004                                    2298+DCORESIZ EQU   X'4'
00001C                                    2299+DERRINF  EQU   X'1C'
0000DC                                    2300+DHIMAIN  EQU   X'DC'
0000E0                                    2301+DIPLDEV  EQU   X'E0'
0000CC                                    2302+DLDADDR  EQU   X'CC'




BSC    CTF BASIC COMPILER EXECUTIVE PROGRAM


  LOC  OBJECT CODE     ADDR1 ADDR2  STMT    SOURCE STATEMENT

000014                                    2303+DLDRTBL  EQU   X'14'
0000C8                                    2304+DLOCCNT  EQU   X'C8'
0000D8                                    2305+DLOWEXT  EQU   X'D8'
0000C4                                    2306+DLSTADR  EQU   X'C4'
000000                                    2307+DLSTSVC  EQU   X'0'
00001C                                    2308+DNRMINF  EQU   X'1C'
000318                                    2309+DPDISK   EQU   X'318'
0000D0                                    2310+DPSW     EQU   X'D0'
00030C                                    2311+DSDISK   EQU   X'30C'
000010                                    2312+DSTADDR  EQU   X'10'
0000E2                                    2313+DSYSDEV  EQU   X'E2'
000018                                    2314+DTBLNG   EQU   X'18'
000324                                    2315+DTDISK   EQU   X'324'
000008                                    2316+DUSFL    EQU   X'8'
                                          2317+*
```

Figure 9. Displacements from NUCON

## Disk Management

Disk files are managed by a series of control blocks and tables. These blocks and tables are disk resident when the system is not in operation and, for the most part, main storage resident during a session. Because the blocks and tables are dynamically updated during the course of a session, an up-to-date copy of them is stored on the disk whenever necessary or requested. (The previous copy is deleted.) This is done to minimize the effect of a system malfunction.

### File Management Tables on Disk

A description of each existing file is maintained in a file status table (FST) for that file. The format of an FST entry is shown in Figure 11.

The file status tables for all files on the disk are grouped into a series of 800-byte disk records referred to as file status table blocks (FSTB). The file status table blocks are stored on the disk in available 800-byte records. Each file status table block can accommodate up to 20 file status tables. Each of the file status table blocks is pointed to by an entry in the master file directory (MFD). The master file directory is the major file management table for disk. It is an 800-byte disk record located at a fixed point on the disk (cylinder 00, track 0, record 4). Figure 12 shows the format of the master file directory. Figure 13 shows the relationship of the master file directory, file status table blocks and file status tables.

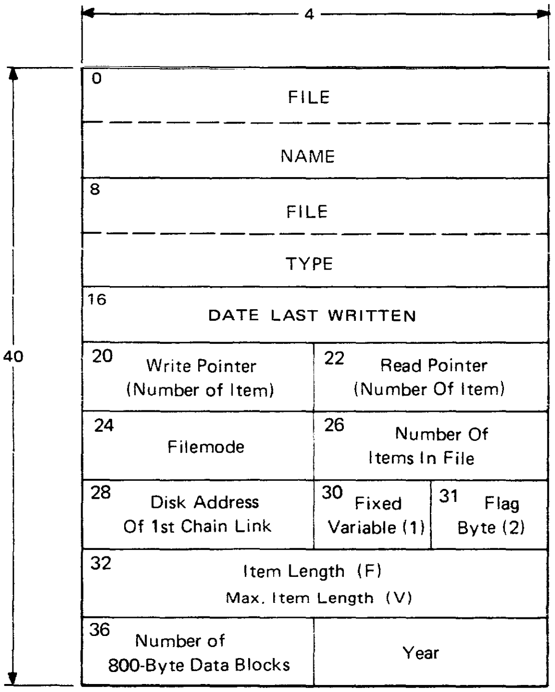
```
                                    2318+*   DISPLACEMENT WITHIN ''SYSREF'' --- COMMUNICATION VECTOR REGION
                                    2319+*
00009C                              2320+DADTLKP  EQU   156
000000                              2321+DADTLKW  EQU   192
000028                              2322+DADTP    EQU   40
00004C                              2323+DADTS    EQU   76
000040                              2324+DADTT    EQU   64
000050                              2325+DBTYPLIN EQU   80
000004                              2326+DBUFFER  EQU   4
000008                              2327+DCMSOP   EQU   8
0000AC                              2328+DCMSRET  EQU   172
0000BC                              2329+DCOMBUF  EQU   188
00000C                              2330+DDEVTAB  EQU   12
000084                              2331+DDIOSECT EQU   132
00007C                              2332+DDMPEXEC EQU   124
000064                              2333+DDMPLIST EQU   100
0000B4                              2334+DEXEC    EQU   180
0000C8                              2335+DEXISECT EQU   200
0000F0                              2336+DFCBTAB  EQU   240
000054                              2337+DFREDBUF EQU   84
000068                              2338+DFREE    EQU   104
00006C                              2339+DFRET    EQU   108
000010                              2340+DFSTLKP  EQU   16
00001C                              2341+DFSTLKW  EQU   28
000000                              2342+DFVS     EQU   0
000014                              2343+DGETCLK  EQU   20
000018                              2344+DGFLST   EQU   24
000024                              2345+DIADT    EQU   36
000030                              2346+DIOERRSP EQU   48
0000E8                              2347+DIONTABL EQU   232
00005C                              2348+DLNKLST  EQU   92
000090                              2349+DMACLIBL EQU   144
000094                              2350+DMACSECT EQU   148
0000B0                              2351+DNOTRKST EQU   176
000078                              2352+DNUMTRKS EQU   120
000008                              2353+DOPSECT  EQU   8
0000A8                              2354+DOSRET   EQU   168
000064                              2355+DOSVECT  EQU   100
000088                              2356+DOSTABLE EQU   136
000020                              2357+DPIE     EQU   32


BSC     CTF BASIC COMPILER EXECUTIVE PROGRAM


  LOC  OBJECT CODE    ADDR1 ADDR2 STMT   SOURCE STATEMENT

00002C                              2358+DPRTCLK  EQU   44
000034                              2359+DRDTK    EQU   52
000038                              2360+DSCAN    EQU   56
0000D4                              2361+DSCBPTR  EQU   212
000070                              2362+DSETCLK  EQU   112
00003C                              2363+DSSTAT   EQU   60
0000B8                              2364+DSTART   EQU   184
0000A4                              2365+DSTATEXT EQU   164
000060                              2366+DSTRINIT EQU   96
000098                              2367+DSVCSECT EQU   152
000044                              2368+DSWITCH  EQU   68
0000EC                              2369+DSYSCTL  EQU   236
000048                              2370+DTABEND  EQU   72
0000CC                              2371+DTBL2311 EQU   204
0000D0                              2372+DTBL2314 EQU   208
000074                              2373+DTXTLIBS EQU   116
0000A0                              2374+DUPUFD   EQU   160
0000C4                              2375+DUSABRV  EQU   196
0000D8                              2376+DUSER1   EQU   216
0000DC                              2377+DUSER2   EQU   220
0000E0                              2378+DUSER3   EQU   224
0000E4                              2379+DUSER4   EQU   228
00008C                              2380+DUSVCTBL EQU   140
000058                              2381+DWRTK    EQU   88
```

Figure 10. Displacements of V-cons from SYSREF



| | | | | | |
|---|---|---|---|---|---|
| 0 | FILE<br>NAME | | | | |
| 8 | FILE<br>TYPE | | | | |
| 16 | DATE LAST WRITTEN | | | | |
| 20 | Write Pointer (Number of Item) | 22 | Read Pointer (Number Of Item) | | |
| 24 | Filemode | 26 | Number Of Items In File | | |
| 28 | Disk Address Of 1st Chain Link | 30 | Fixed Variable (1) | 31 | Flag Byte (2) |
| 32 | Item Length (F)<br>Max. Item Length (V) | | | | |
| 36 | Number of 800-Byte Data Blocks | | Year | | |

Notes on Date and Year:

(1) Date Last written is in packed decimal format MM DD HH MM e.g. 02 20 14 07 represents February 20, 2:07 p.m.

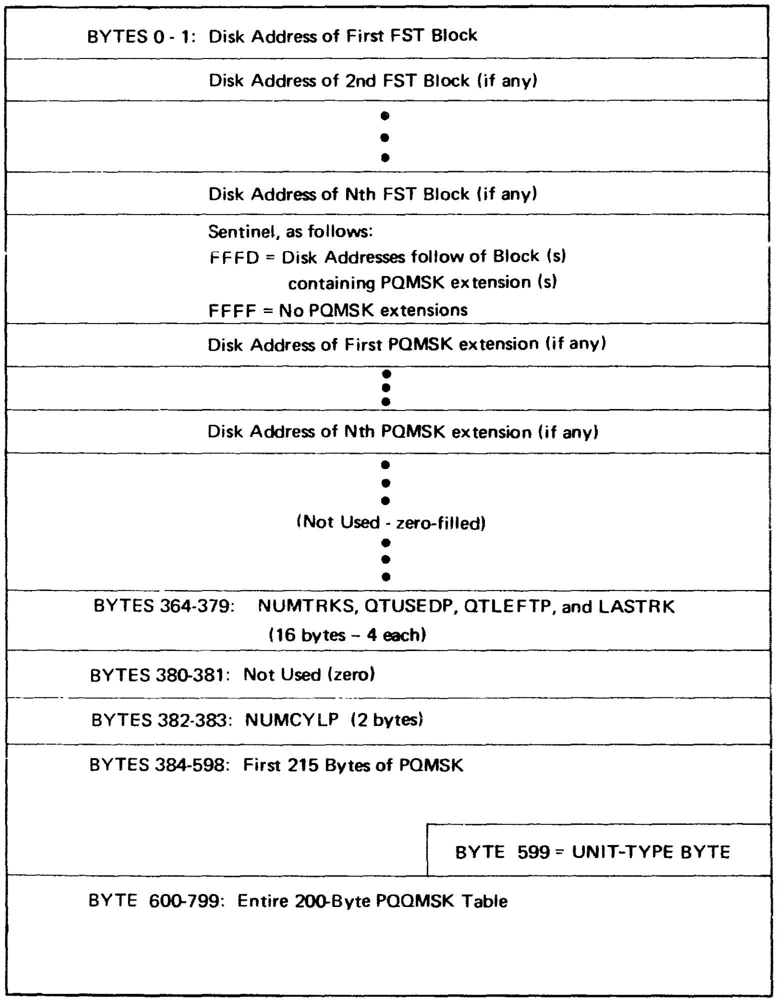
(2) Year is in character form e.g. C'69' for 1969

NOTE:

(1) F = Fixed Length Items
V = Variable Length Items

(2) Flag BYTE: = 0

Figure 11. File status table entry



BYTES 0 - 1: Disk Address of First FST Block

Disk Address of 2nd FST Block (if any)

•
•
•

Disk Address of Nth FST Block (if any)

Sentinel, as follows:
FFFD = Disk Addresses follow of Block (s)
containing PQMSK extension (s)
FFFF = No PQMSK extensions

Disk Address of First PQMSK extension (if any)

•
•
•

Disk Address of Nth PQMSK extension (if any)

•
•
•
(Not Used - zero-filled)
•
•
•

BYTES 364-379: NUMTRKS, QTUSEDP, QTLEFTP, and LASTRK
(16 bytes - 4 each)

BYTES 380-381: Not Used (zero)

BYTES 382-383: NUMCYLP (2 bytes)

BYTES 384-598: First 215 Bytes of PQMSK

BYTE 599 = UNIT-TYPE BYTE

BYTE 600-799: Entire 200-Byte PQQMSK Table

NOTES: NUMTRKS = Total number of 800-byte records on User's P-Disk
QTUSEDP = Number of records currently in use on User's P-Disk
QTLEFTP = Number of records left (NUMTRKS less QTUSEDP)
LASTRK = Relative byte-address of last record in use on P-Disk

NUMCYLP = Number of cylinders in User's P-Disk
UNIT-TYPE = 01 for 2311 Disk, 08 for 2314 Disk

Figure 12. Master file directory (MFD)

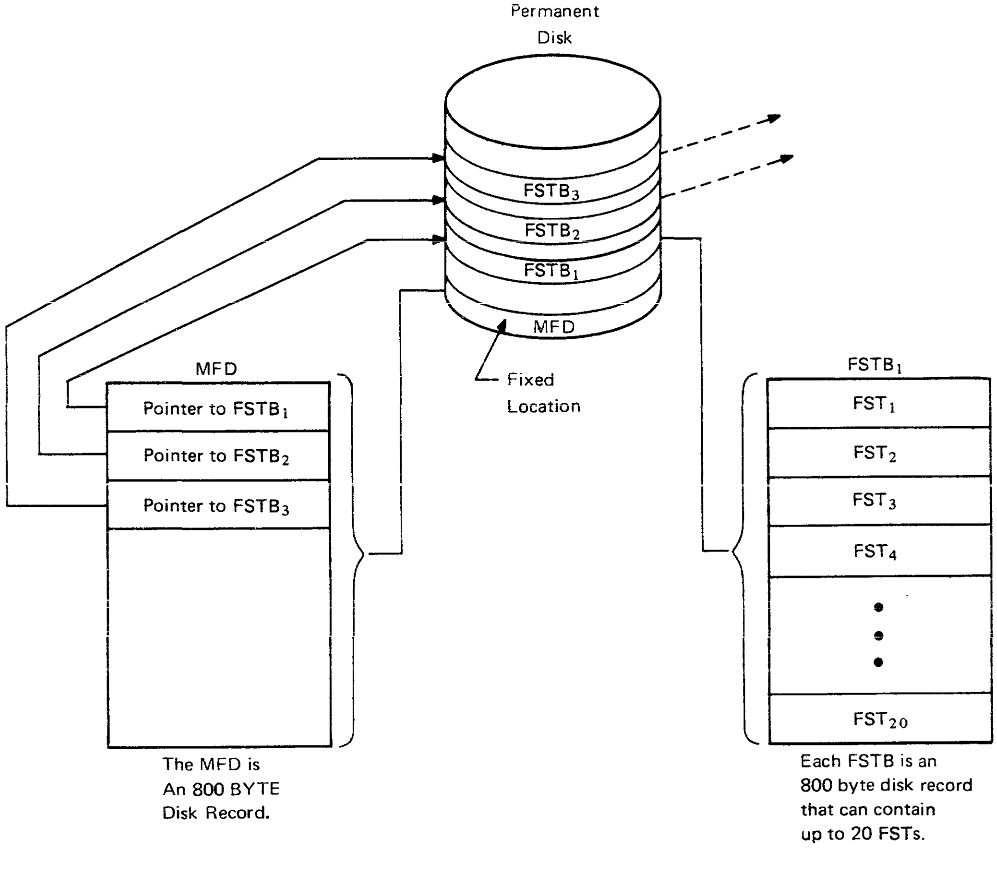


Figure 13. MFD, FSTB, and FST relationship

The individual items in a file (for example, card images, live print images) are stored in 800-byte disk records referred to as data blocks. A series of pointers that originate in the associated file status table keep track of the data blocks. This series of pointers is called a chain link. Within the file status table entry is a pointer to the first chain link, a 200-byte disk record that contains (1) the disk addresses of the first 60 data blocks for the file and (2) the disk addresses of up to 40 other chain links. Figure 14 shows the format of the first chain link. Each of the other chain links (the second through the forty-first) is an 800-byte disk record containing the disk addresses of up to 400 additional data blocks. The second chain link (if required) contains the disk addresses of data blocks 61 through 460; the third chain link (if required) contains the disk addresses of data blocks 461 through 860, etc. Figure 15 shows the format of the Nth chain link. Figure 16 shows the relationship of the file status table, chain links, data blocks, and items for a file.

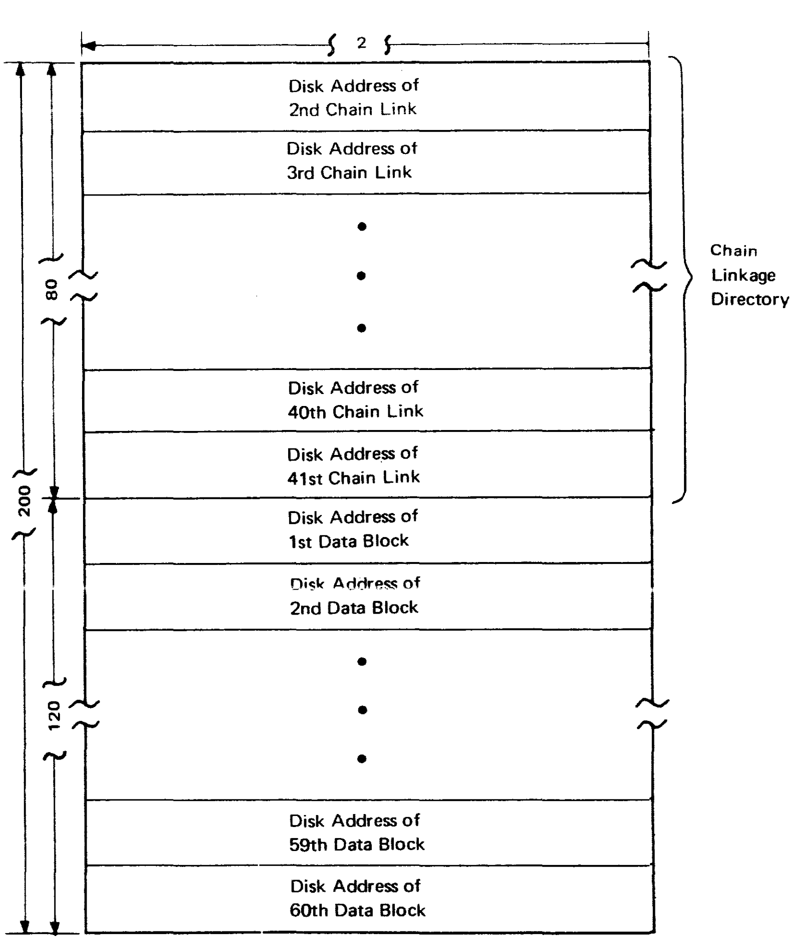

2
200
80
120
Disk Address of 2nd Chain Link
Disk Address of 3rd Chain Link
Disk Address of 40th Chain Link
Disk Address of 41st Chain Link
Disk Address of 1st Data Block
Disk Address of 2nd Data Block
Disk Address of 59th Data Block
Disk Address of 60th Data Block
Chain Linkage Directory

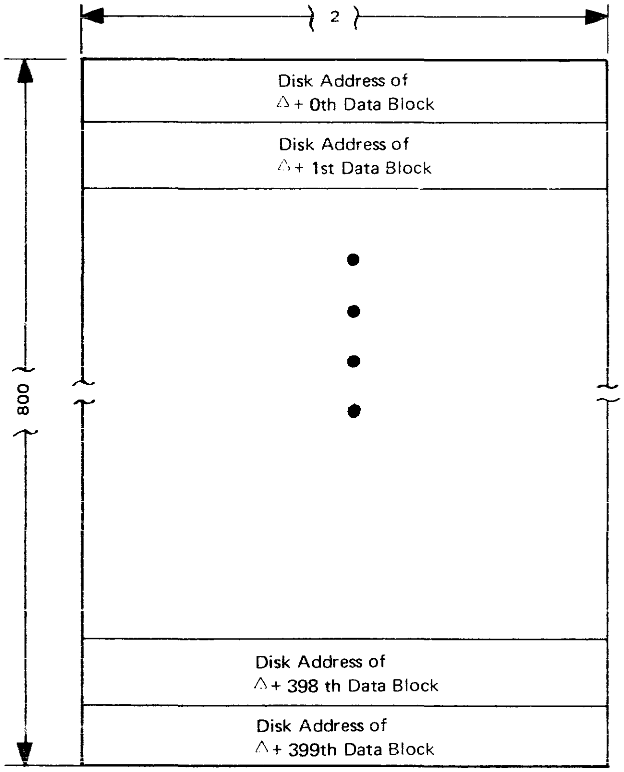
Figure 14. First chain link (FCL)

## File Management Tables in Main Storage

During a session, CMS maintains a user file directory for a user's active disk; it includes pertinent information on a user's files, his QMSK and QQMSK (see the next section), and the number of cylinders and other statistical data on his permanent disk.

The user file directory (UFD) is kept on the user's disk in the form of (1) a master file directory (MFD), which is kept in a fixed place on disk (cylinder 0, track 0, record 4), (2) an 800-byte record for the first file status table block, (3) additional records (as needed) for any additional file status table blocks and (4) any QMSK extensions as needed, if there is significant data beyond the first 215 bytes of QMSK.

At CMS initialization time, the user file directory is brought into core by the LOGIN command program (unless the user's first command is LOGIN NO-UFD, FORMAT P, or FORMAT P ALL). The file status table blocks that contain the individual file status tables required for the session are linked together into a chain that originates in the data area PSTAT. See Figure 17. The user file directory is then maintained in core in the active disk table by the various file maintenance and disk programs, for the duration of this session. At the completion of each typed-in command, between commands executed under the EXEC command, upon CMS LOGOUT, and at other key points as needed, the user file directory is updated on the user's disk by the UPDISK routine.



Δ = (n-2) • 400 + 61
where n = Chain Link Number

Figure 15. Nth chain link

Before a file can be read or written, it must be opened and made active. Making a file active consists of constructing an entry in the active file table (AFT), which includes a copy of the file's FST entry. An active file table entry contains disk addresses and core addresses for the chain link and data block that are currently in core. Figure 18 shows the format of a single active file table entry, and Figure 19 illustrates the form of the active file table showing several entries.

A file's entry is removed from the active file table when the FINIS command is issued for it. See the section "Active File Table (AFT) Management" for further information.

### Data Block Structure

Files stored on the disk may be made up of either fixed-length or variable-length items. (A mixture of fixed-length and variable-length items within a particular file is not permitted.) Regardless of their format, the items are stored in sequential order in as many data blocks as are required to accommodate them. Each data block, with the exception of the last, is completely filled and, where necessary, items that are started in one data block are continued in the next. Figures 20 and 21 show the data block formats for files containing fixed-length and variable-length items, respectively.

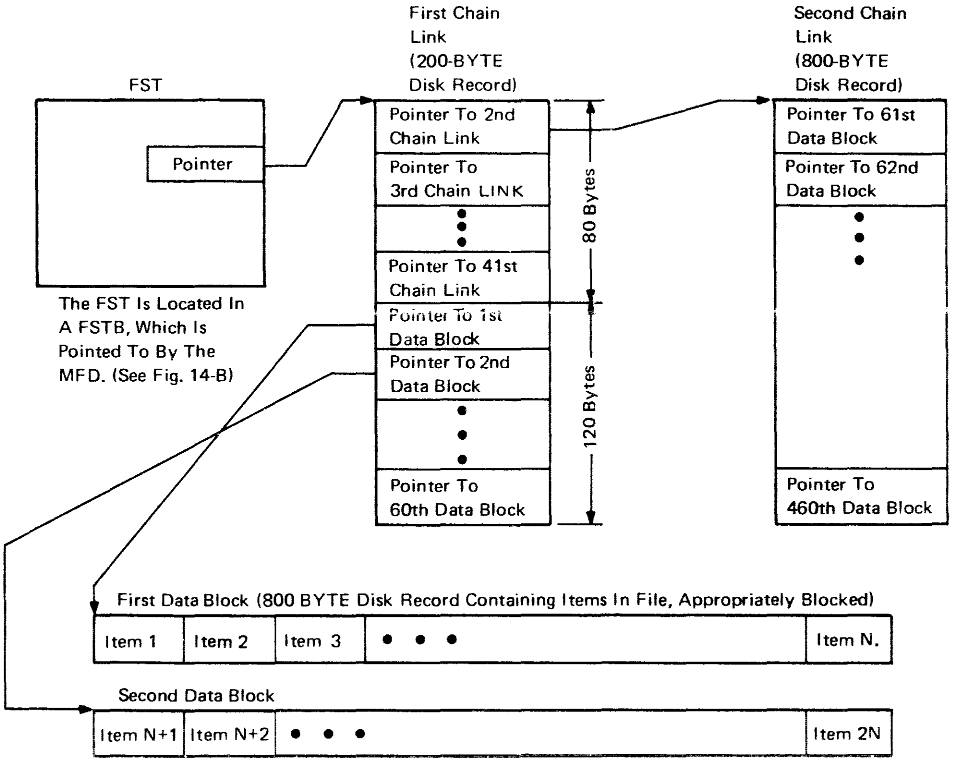
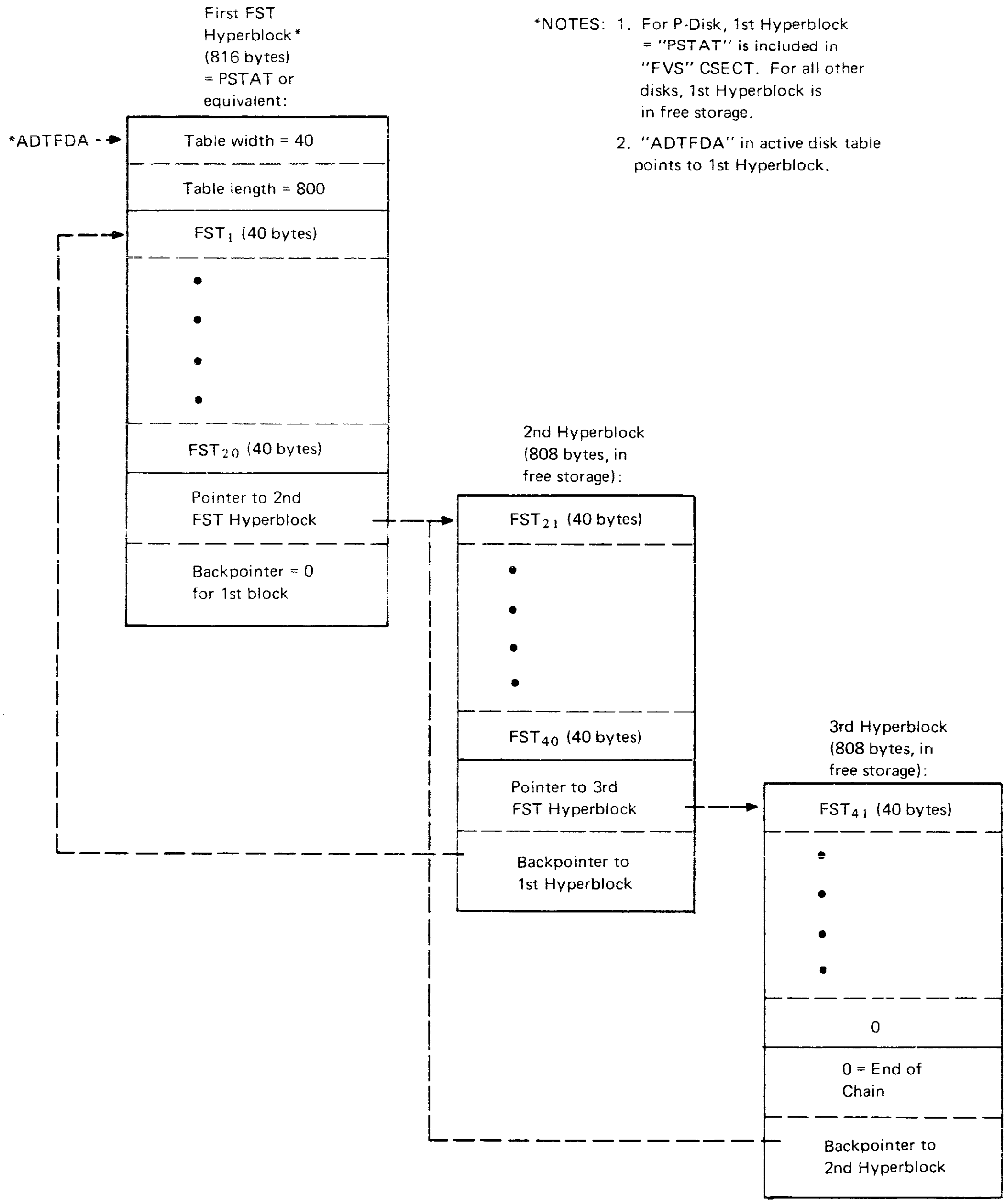

FST
Pointer
The FST Is Located In A FSTB, Which Is Pointed To By The MFD. (See Fig. 14-B)
First Chain Link (200-BYTE Disk Record)
Pointer To 2nd Chain Link
Pointer To 3rd Chain LINK
Pointer To 41st Chain Link
Pointer To 1st Data Block
Pointer To 2nd Data Block
Pointer To 60th Data Block
80 Bytes
120 Bytes
Second Chain Link (800-BYTE Disk Record)
Pointer To 61st Data Block
Pointer To 62nd Data Block
Pointer To 460th Data Block
First Data Block (800 BYTE Disk Record Containing Items In File, Appropriately Blocked)
Item 1
Item 2
Item 3
Item N.
Second Data Block
Item N+1
Item N+2
Item 2N

Figure 16. Relationship of FST, chain links, data blocks, and items of a file



*NOTES: 1. For P-Disk, 1st Hyperblock = "PSTAT" is included in "FVS" CSECT. For all other disks, 1st Hyperblock is in free storage.

2. "ADTFDA" in active disk table points to 1st Hyperblock.

Figure 17. Example of three FST hyperblocks in main storage

```
         CONTENTS OF ONE ACTIVE-FILE-TABLE BLOCK

AFTSECT  DSECT         (DSECT name if referenced via "AFT" Macro)

Name:    Size:         Disp. Contents:

AFTCLD   DS     H          0 DISK ADDRESS OF CURRENT CHAIN LINK
AFTCLN   DS     H          2 NUMBER OF CURRENT CHAIN LINK
AFTCLA   DS     F          4 CORE ADDRESS OF CHAIN LINK BUFFER
AFTDBD   DS     H          8 DISK ADDRESS OF CURRENT DATA BLOCK
AFTDBN   DS     H         10 NUMBER OF CURRENT DATA BLOCK
AFTDBA   DS     F         12 CORE ADDRESS OF CURRENT DATA BLOCK
AFTCLB   DS     XL80      16 CHAIN LINK BUFFER FROM 1ST CHAIN LINK
AFTFLG   DS     X         96 FLAG BYTE

         FLAG BYTE (AFTFLG) DEFINITIONS:

AFTUSED  EQU    X'80'        ACTIVE FILE TABLE BLOCK IN USE
***      EQU    X'40'        *NOT CURRENTLY USED*
AFTICF   EQU    X'20'        FIRST CHAIN LINK IN CORE FLAG
AFTFBA   EQU    X'10'        FULL BUFFER ASSIGNED
AFTDBF   EQU    X'08'        DATA BLOCK IN CORE FLAG
AFTWRT   EQU    X'04'        ACTIVE WRITE
AFTRD    EQU    X'02'        ACTIVE READ
AFTFULD  EQU    X'01'        FULL-DISK SPECIAL CASE

AFTPFST  DS     3X        97 POINTER TO FST-ENTRY in FST HYPERBLOCK
AFTIN    DS     H        100 CURRENT ITEM NUMBER
AFTID    DS     H        102 DISPLACEMENT OF CURRENT ITEM IN DATA BLOCK

AFTFST   DS     0D       104 ACTIVE FST BLOCK (AFTN thru AFTYR):

AFTN     DS     D        104 FILE NAME
AFTT     DS     D        112 FILE TYPE
AFTD     DS     F        120 DATE/TIME LAST WRITTEN
AFTWP    DS     H        124 WRITE POINTER (ITEM NO.)
AFTRP    DS     H        126 READ POINTER (ITEM NO.)
AFTM     DS     H        128 FILE MODE
AFTIC    DS     H        130 ITEM COUNT
AFTFCL   DS     H        132 FIRST CHAIN LINK
AFTFV    DS     C        134 FIXED(F)/VARIABLE(V) FLAG
AFTFB    DS     X        135 FST FLAG BYTE (=0)
AFTIL    DS     F        136 (MAXIMUM) ITEM LENGTH
AFTDBC   DS     H        140 800-BYTE DATA BLOCK COUNT
AFTYR    DS     H        142 YEAR

AFTADT   DS     F        144 POINTER TO ACTIVE DISK TABLE
AFTPTR   DS     F        148 POINTER TO NEXT AFT BLOCK IN CHAIN

AFTFSF   EQU    X'40'        BIT IN AFTPTR INDICATES IN FREE STORAGE
```

Figure 18. Active file table block

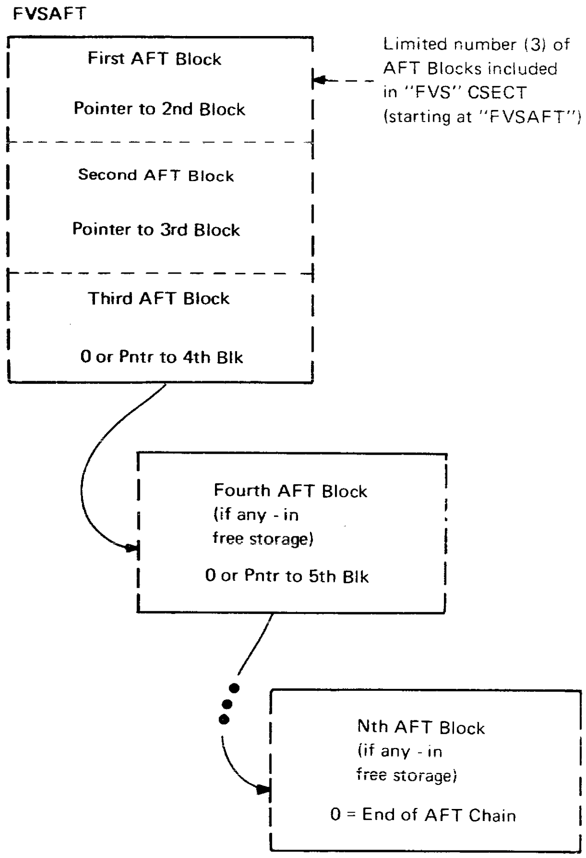

FVSAFT
First AFT Block
Pointer to 2nd Block
Limited number (3) of AFT Blocks included in "FVS" CSECT (starting at "FVSAFT")
Second AFT Block
Pointer to 3rd Block
Third AFT Block
0 or Pntr to 4th Blk
Fourth AFT Block (if any - in free storage)
0 or Pntr to 5th Blk
Nth AFT Block (if any - in free storage)
0 = End of AFT Chain

Figure 19. Active file table (AFT)

## System Disk

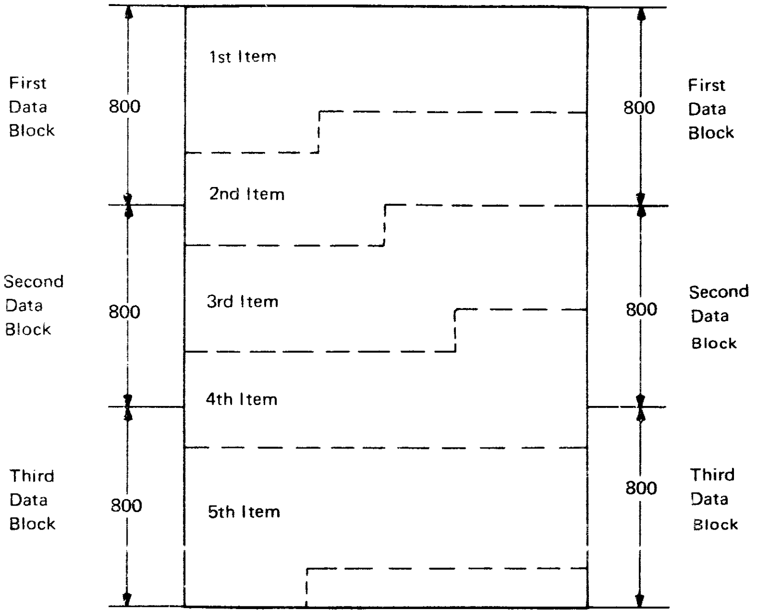
The file management scheme for the system disk is similar to that for any read-only disk. However, the usual file status table blocks are replaced by a system status table (SSTAT), which becomes part of the nucleus.

The system status table contains a file status table for each system disk file (for example, a disk resident command program) that has a filemode of P2 when generated onto the system disk.

SSTAT is created by the SYSGEN routine, as required at IPL time. SSTAT is constructed within the free storage of the nucleus. It contains references to all files of mode P2. When the IPL CMS feature (IPL by name) is used, however, a copy of the nucleus with SSTAT already filled in is obtained, and SYSGEN is not called.

## DISK SPACE MANAGEMENT

Disk space management allocates and keeps track of storage on the primary, temporary, or any other read-write disk. Various CMS programs require disk space in which to write user files and control tables. When a program needs disk space, it calls the

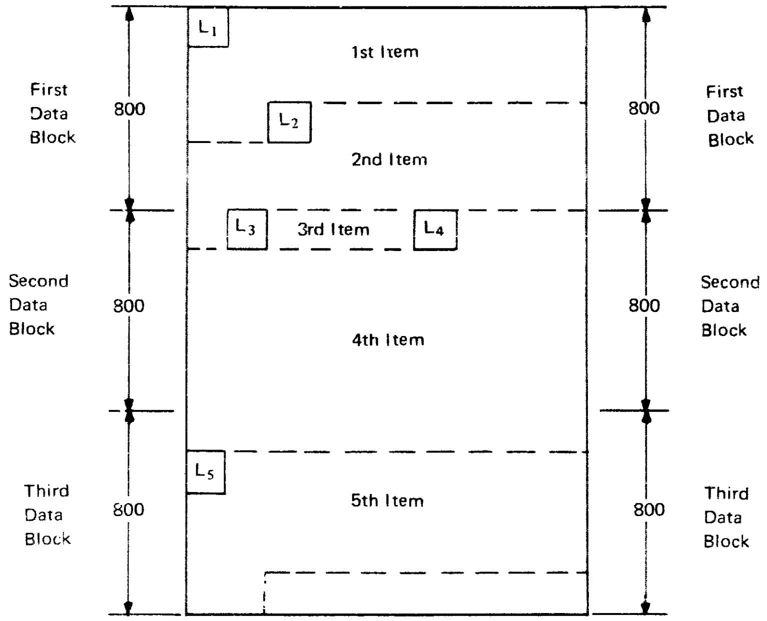


NOTE: The location of an arbitrary Item in a file consisting of fixed-length items is determined by the following formula:

$$\frac{(\text{Item Number} - 1) \times \text{Item Length}}{800}$$

where the Quotient=Data Block Number and the Remainder=Displacement in Data Block

Figure 20. Data block structure for file consisting of fixed-length items



NOTE: Each variable length item is preceded by a two-byte field that contains the length of the item.

Figure 21. Data block structure for file consisting of variable-length items

disk space management routines, which allocate an area of the requested size and return the starting address of the area to the caller.

A disk area of 800 bytes (a quarter-track on a 2311) is requested for a data-block or Nth chain link. A 200-byte area (a sixteenth-track on a 2311) is requested if a first chain link is needed.

(On a 2311, the records are physically grouped four 829-byte records per track, of which CMS uses the first 800 bytes -- hence, the terms "quarter-track" and "sixteenth-track". On a 2314 disk, the records are actually grouped fifteen 829-byte records per two tracks, of which CMS uses the first 800 bytes.)

Read-Write Disks

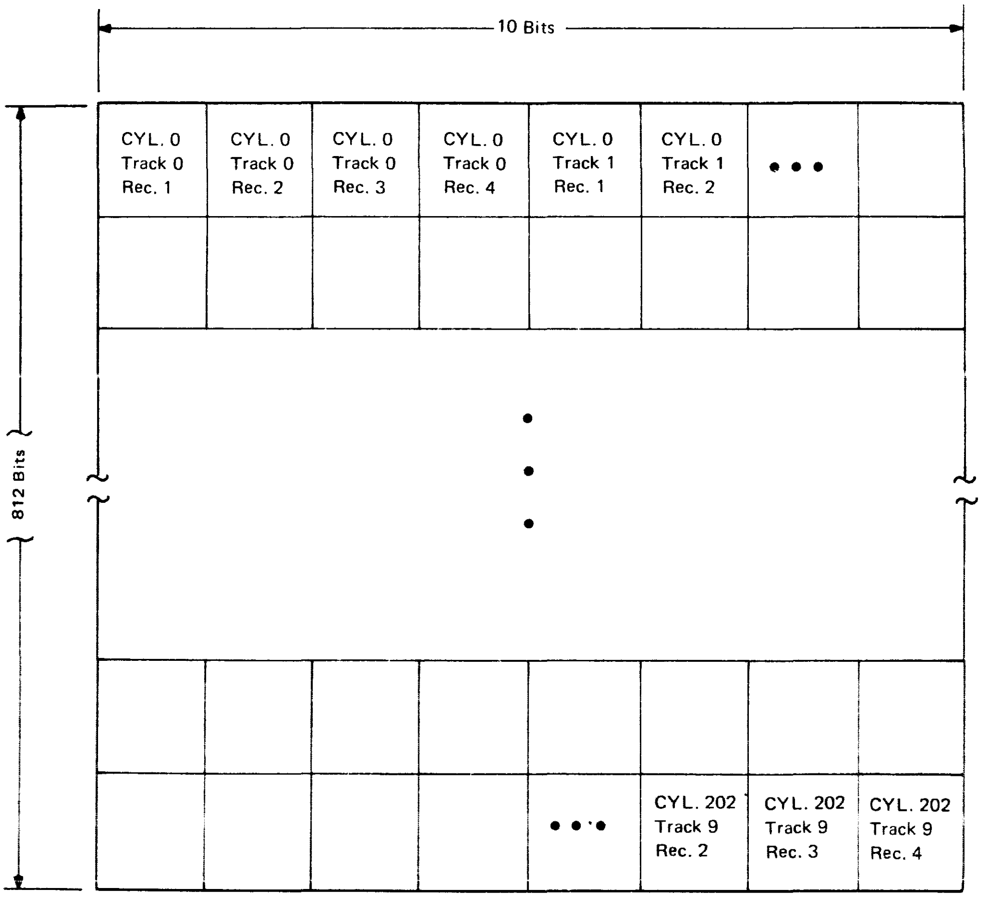
The status of quarter tracks on any read-write disk (which are available and which are in current use) is stored in a table called QMSK.

When the system is not in use, a user's QMSK is disk-resident (in the user file directory); during a session it is maintained on disk, but also resides in main storage. QMSK is of variable-length, depending on how many cylinders exist on the primary disk (2311 or 2314), but is an integral number of bytes. Each bit is associated with a particular quarter track on the primary disk. The first bit in QMSK corresponds to the first quarter track, the second bit to the second quarter track, etc. When a bit in QMSK is set to one, it indicates that the corresponding quarter track is in use and not available for allocation. A zero bit indicates that the corresponding quarter track is available. Figure 22 illustrates the format of QMSK for a 203-cylinder 2311.

Another table, called QQMSK indicates which sixteenth tracks are available for allocation and which are currently in use. QQMSK contains 100 entries, which are used to indicate the status of up to 100 sixteenth tracks. An entry in QQMSK contains either a disk address, pointing to a sixteenth track that is available for allocation, or zero. Figure 23 shows the format of the QQMSK.

Disk space management is implemented as follows. When a program requires a quarter track, it calls the disk space management routine TRKLKP, which scans the QMSK table for the first zero bit. When found, the bit is turned on to indicate that the corresponding quarter track is being used, and the address of the quarter track is returned to the caller. When a program frees a particular quarter track, it passes the address of the quarter track to the disk management routine TRKLKPX, which sets the corresponding bit in QMSK to zero. The quarter track is thereby flagged as being available for allocation to another program.

When a program (namely WRBUF) requires a sixteenth of a track it calls the disk space management routine QQTRK, which scans the QQMSK table, starting from the beginning, for the first entry containing a disk address. If such an entry exists, the disk address is given to the caller and the entry is set to zero. If no such entry exists, QQTRK obtains a quarter track from TRKLKP, normally segments it into four equal parts, places the addresses of the last three-sixteenths of the track into the first three zero entries in QQMSK, and returns the address of the first sixteenth of the track to the caller.



NOTE:

| Bit Value | Meaning |
|---|---|
| 0 | Quarter Track not in use. |
| 1 | Quarter Track in use. |

QMSK for a 2311 Disk (up to 203 cylinders): maximum of 8120 bits (1015 bytes)

For a 2314 Disk, a bit is assigned to each 800-byte record, 15 bits per two tracks, up to 150 bits per cylinder.

For permanent disk, first 215 bytes of PQMSK are kept in MFD; the remaining bytes (if any) on disk (in as many PQMSK extensions as necessary), pointed to by MFD.

Figure 22. Disk quarter-track assignment (QMSK)

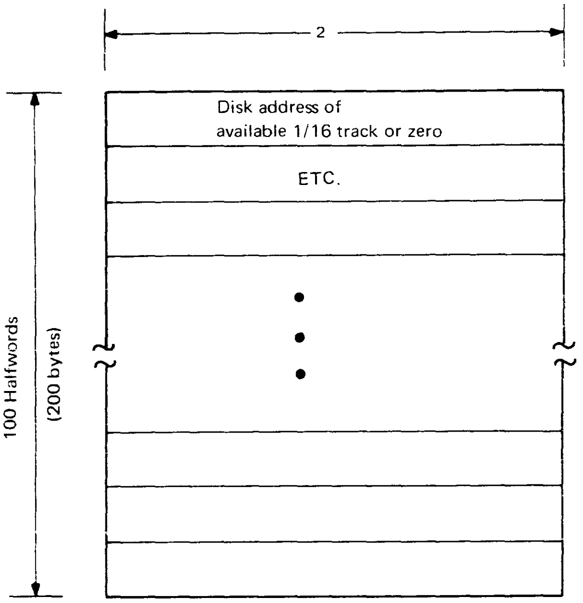


Figure 23. Disk sixteenth track assignment (QQMSK)

When a program (for example, ERASE) frees a sixteenth of a track, it passes its address to the disk space management routine QQTRKX. This routine scans the QQMSK table for the addresses of the three other sixteenths that resulted from the division of the quarter track. If all three addresses are present, indicating that a complete quarter track has been freed, the three addresses are deleted from QQMSK, and the bit in QMSK corresponding to the freed quarter track is set (via TRKLKPX) to reflect its availability for allocation. If all three addresses are not present, the address of the sixteenth to be freed is placed in the first zero entry in QQMSK.

The disk addresses used throughout CMS are in the form of halfword block numbers. Figures 24 and 25 show their format for first chain links (sixteenth-tracks) and other (800-byte) records, respectively.

Some special logic is used for first chain links if a block number obtained from TRKLKP exceeds 8191. The notes on Figure 24 show the algorithms used.

The location of the QQMSK table, and the location and size of the QMSK table are kept for each disk in the active disk table. Pertinent information, with the mnemonics used in the ADT macro, are given below. The complete information is given in Figure 34 under File Management Macros.

| Block Number of 800-byte block containing 200-byte First Chain Link | Format of halfword Disk Address | Binary Representation *** |
|---|---|---|
| 1-8191 * | Bits 0-1 contain 00, 01, 10, or 11<br>Bits 2-15 contain block number | xx0x xxxx xxxx xxxx |
| 8192-16383 ** | Block Number "as is" | 001x xxxx xxxx xxxx |
| 16384-24575 ** | Block Number + 24576 | 101x xxxx xxxx xxxx |
| 24576-32767 ** | Block Number "as is" | 011x xxxx xxxx xxxx |

* For Block Numbers < 8192, first chain links are grouped four per 800-byte block. The first two bits of the disk-address (00, 01, 10, or 11) signify which 200 bytes of the 800-block are used.

** For Block Numbers of 8192 or greater, first chain links occupy the first 200 bytes of an 800-byte block. (Remaining 600 bytes are unused)

*** Note that the various disk routines (RDTK-WRTK, QQTRK, & QQTRKX) can readily distinguish between the two types of first chain link from the third bit of the disk-address with no ambiguity (0 for block-numbers 1-8191, 1 for block numbers from 8192 up).

Figure 24. Disk address format for first chain links

Disk-Addresses for Data-Blocks and N'th Chain Links are in the form of a halfword 'Block Number' from 1 up, as follows:

| |
|---|
| 0xxx xxxx xxxx xxxx |

(The largest possible block-number for a full 203-cylinder 2314 Disk is less than 32767)

Figure 25. Disk address format for Nth chain links and for data blocks

## 1. Disk statistics available on any logged-in disk:

ADT1ST Displacement of first full-word in QMSK containing at least one bit = 0 (maintained in core by TRKLKP and TRKLKPX)

ADTNUM Total number of 800-byte records on disk (formerly called NUMTRKS)

ADTUSED Number of records in use (formerly called QTUSEDP)

ADTLEFT Number of records left = ADTNUM - ADTUSED (formerly called QTLEFTP)

ADTLAST Displacement of last nonzero byte in QMSK (formerly LASTRK)

ADTCYL Number of Cylinders on Disk (formerly NUMCYLP or NUMCYLT)

2. Kept in core only for a logged-in-read-write disk:

ADTMSK Core-address of QMSK table.

ADTQQM Core-address of QQMSK table.

ADTPQM1 Number of bytes (in any) in QMSK exceeding 215
(formerly PQMSIZ)

ADTPQM2 Number of 800-byte records (QMSK extensions) needed for maintaining on disk a QMSK exceeding 215 bytes in length.
(formerly PQMNUM)

ADTPQM3 Number of double words in QMSK
(formerly RONUM)

For any read-write disk, the QMSK and QQMSK tables are brought into core when the disk is logged in by the READMFD routine. They are maintained as described above by TRKLKP, TRKLKPX, QQTRK, and QQTRKX. The updated QMSK and QQMSK tables are written on disk when appropriate by the UPDISK routine, which maintains the User File Directory for any given read-write disk.

Size of QMSK Bit-Mask

As mentioned above, the size of the QMSK is a function of the size of the disk. A one-cylinder 2311, containing only 40 records, would require only five bytes, or one double word, for the QMSK. A 2311 of up to 43 cylinders, or a 2314 of up to 11 cylinders, would require no more than 215 bytes for its QMSK, which would be kept entirely in the Master File Directory (MFD) on disk. A larger disk would require additional QMSK extensions for maintaining the QMSK on disk, as shown in the following table:

| Number of QMSK Extensions Required (If Any) | Number of Cylinders on Disk | |
|---|---|---|
| | 2311 | 2314 |
| None | 1 - 43 | 1 - 11 |
| 1 | 44 - 203 | 12 - 54 |
| 2 | - | 55 - 96 |
| 3 | - | 97 - 139 |
| 4 | - | 140 - 182 |
| 5 | - | 183 - 203 |

## System and Other Read-Only Disks

When any disk (system disk or otherwise) is read-only, its QMSK and QQMSK tables are not brought into core and, like the rest of the UFD for that disk, remain as is on disk until such time as the disk is logged in as a read-write disk.

## ACTIVE FILE TABLE (AFT) MANAGEMENT

When files are being read or written by CMS, the necessary data is kept in the Active File Table (AFT).

There is no specific limit as to how many files can be active at any one time (formerly there was a limit of 8). Since many I/O programs handle at least two or three active files, it is convenient to have a few AFT blocks available at all times; others are created from free storage upon demand and are released when no longer needed.

A limited number of AFT blocks, therefore are provided in the CMS nucleus, referenced at FVSAFT within the FVS storage C-Sect. This number is set to 3 as a practical value, but could easily be changed if desired, by a revision of the FVS table. It must be at least 1, however.

Four routines are used for active file table management - ACTLKP, ACTNXT, ACTFREE, & ACTFRET.

The AFT macro is used by these and other programs for referencing an Active File Table block. The form of this macro is shown in Figure 18.

## ACTIVE DISK TABLE (ADT) MANAGEMENT

For each disk handled by CMS, the pertinent file directory information is stored in an active disk table entry for that disk. The initial active disk table (IADT) in the CMS nucleus contains the active disk table entries for the six disks handled by CMS. The ADT macro is used to reference an entry in this table.

The disks included in the table, not all of which are necessarily attached to a CMS user at any given time, are shown as follows (showing the order of search used if "any" disk is requested):

| | |
|---|---|
| P-Disk | Primary Disk - usually logged in - normally read - write |
| T-Disk | (if any - normally read-write if logged in) |
| A-Disk | (if any - read-write or read-only) |
| B-Disk | (if any - read-write or read-only) |
| S-Disk | CMS System Disk (read-only) |
| C-Disk | (if any - read-write or read-only) |

The file directory for the S-Disk is brought into core at CMS initialization time by SYSGEN and READFST, with the FST entries being stored in the SSTAT system status table.

The file directory for the P-Disk and/or any other disk is brought into core by the CMS LOGIN command, and released when no longer needed by the RELEASE command.

The S-Disk cannot be logged in (via LOGIN) or released by RELEASE.

The A-Disk, B-Disk, or C-Disk can each be logged in as a separate, unique disk (either read-only or read-write), or as read-only extension of another disk higher in the order of search. For example, the A-Disk could be a read-only extension of the P-Disk or T-Disk; the C-Disk could be a read-only extension of any of the other types.

The T-Disk is normally a unique disk, since many CMS commands have the capability of handling a T-Disk if present.

Note that the C-Disk has a lower priority than the S-Disk. This serves two very important purposes, as follows:

1. Any disk (such as an old CMS S-Disk) which has modules on it that have been superseded by newer modules on the CMS system-disk should be logged in (if needed at all) as the C-DISK. This ensures (through the order of search) that the new module from the S-Disk will be used in preference to the obsolete module on the old disk. (This can be particularly important for an old S-Disk having an obsolete copy of certain file management modules such as LOGIN, LISTF, OFFLINE, etc.)

2. A C-Disk can purposely be made as a read-only extension of an S-Disk, for any of several good reasons. This procedure is discussed in the CP-67/CMS Installation Guide.

Two routines in particular are used for active disk table management — ADTLKP and ADTNXT.

## I/O OPERATIONS

CMS input/output operations may be either synchronous or asynchronous. These I/O operations for direct access storage devices, tapes, card readers, card punches, printers, and the console (when under a programmed controlled read) are synchronous. That is, CMS enters the wait state after starting the I/O operation and resumes processing when the initiated I/O operation causes an interrupt.

Those input/output operations for the console when performing either an output or an input operation initiated by an attention interrupt, are asynchronous. For asynchronous I/O, CMS does not enter the wait state, but rather, CMS continues processing while the user enters data from the terminal or while data is typed to the terminal.

User input/output operations can be either synchronous or asynchronous depending on user requirements. Refer to the section "User Input/Output Operations" for more detail.

## CMS Terminal I/O

Terminal handling deals with the way in which CMS controls terminal I/O activity. This activity is managed by three first-in, first-out lists. The first of these, called the read-write stack, contain an entry for each read or write request that has not been satisfied. The first and last entries in the read-write stack are pointed to by fields (FSTRDWRT, LSTRDWRT) in the console constant area (CONGEN). All other entries are chained together. (See Figure 26.) The number of entries in the read/write stack is also stored in a field (NUMRDWRT) in CONGEN. Each entry in the read-write stack is called a CCW package. The CCW package describes the corresponding read or write. When a read or write request is satisfied, the corresponding CCW package is removed from the read-write stack.

The second list, referred to as the pending read stack, contains an entry for each read request that is pending (that is, waiting to be satisfied). It is similar in structure to the read-write stack except that it only contains entries for pending reads. (See Figure 27.)

When a read request is satisfied, the corresponding entry is removed from the pending read stack and placed into a third stack called the finished read stack.

The finished read stack is similar in structure to the other two stacks except that it only contains entries for finished read operations. (See Figure 28.)

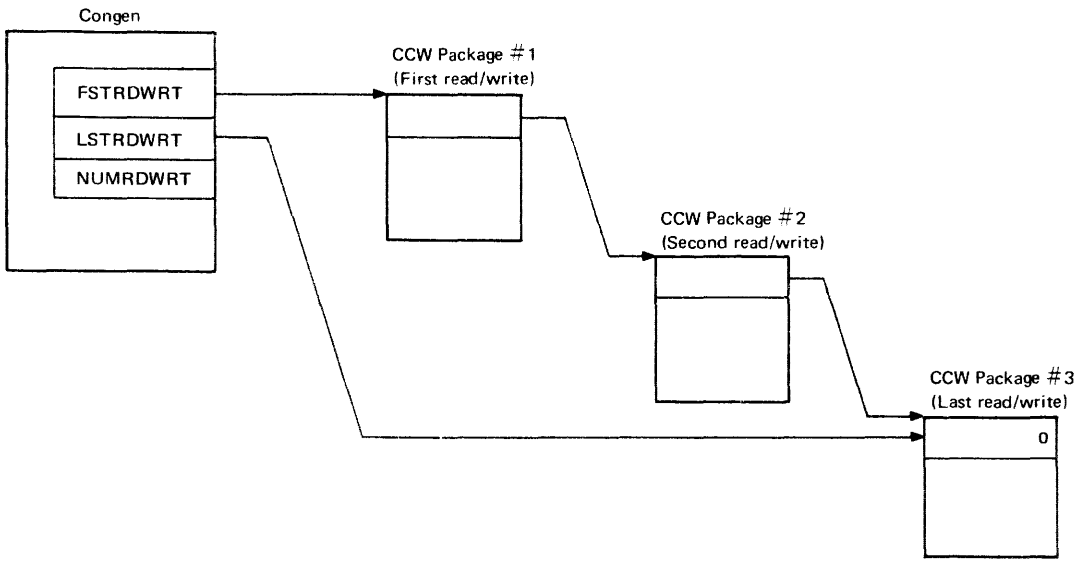


Figure 26. Read/write stack

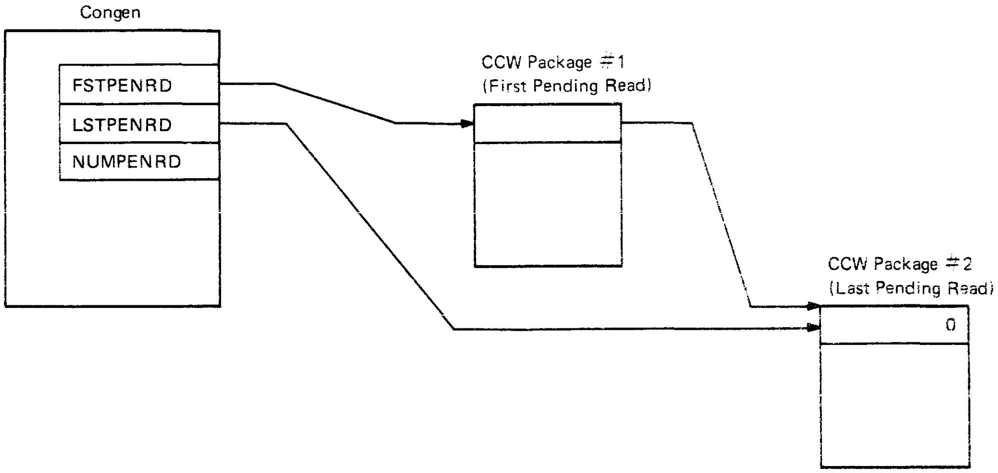


Figure 27. Pending read stack

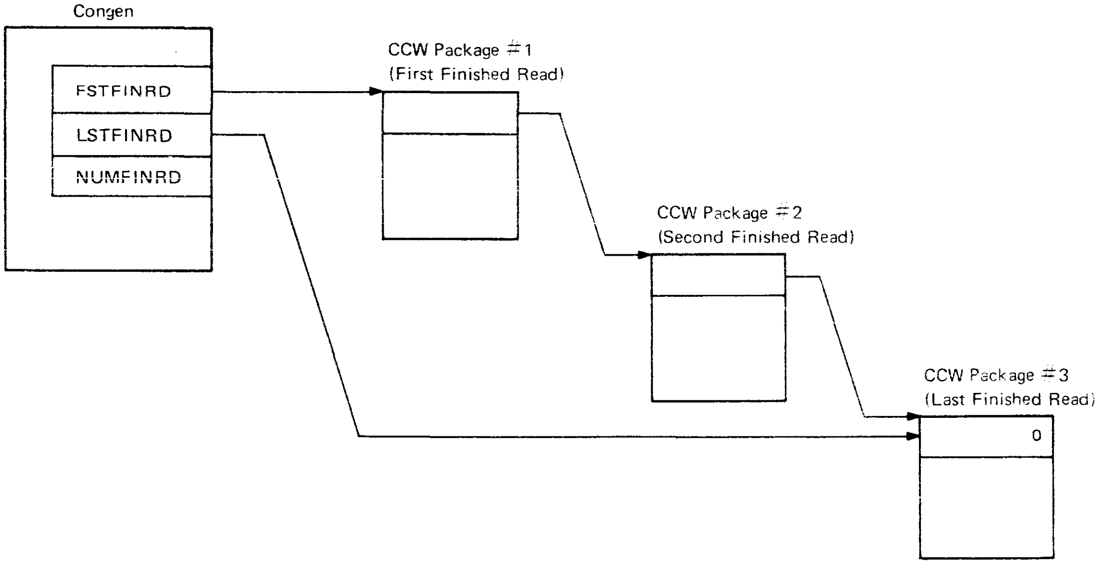


Figure 28. Finished read stack

Detailed information on how terminal operations are handled can be found in the discussions of the WAITRD, TYPLIN, and TYPE function programs and in the discussion of input/output interruptions.

## CMS Nonterminal I/O

As previously stated, non-terminal I/O operations are synchronous. When a CMS program starts an I/O operation on a device other than the terminal, it enters the wait state to wait for an interruption from that device. The program does this by calling a common wait program (WAIT). When WAIT receives control, it loads a PSW that has the wait bit set. This PSW also contains an address pointing to a location within WAIT to which control is to be returned when the interruption being waited for occurs.

When an I/O interrupt occurs, control goes to IOINT as the new I/O PSW points to IOINT. IOINT passes control to the program that handles interrupts for the particular device that caused the interrupt if any such program exists. The program analyzes the interrupt to determine (1) if another interrupt is necessary to complete this particular interrupt sequence (e. g. , channel end, followed by device end), and (2) if an error has occurred during the I/O operation. The program sets GPR15 to 0 for a completed interrupt sequence, and sets GPR15 to nonzero for an incompleted sequence. (The actions taken on an error are discussed in IOERR, the CMS program providing centralized error recovery. ) The interrupt processing program then returns to IOINT which either (1) returns control to WAIT with all interrupts disabled and in the runnable state (i. e. , the wait bit turned off) when the interrupt was caused by the waited-for-device and GPR15 is zero; (2) places the machine in the wait state by loading the old I/O PSW — which had previously been used to place the machine in the wait state — when the interrupt was caused by the waited-for-device and GPR15 is nonzero; or (3) places the machine in the wait state by loading the I/O PSW when the interrupt was not caused by the waited-for-device.

## User Input/Output Operations

CMS allows the user to perform his own input/output operations, i. e. , issue his own start I/O (SIO) instructions anywhere in his CMS machine. This is accomplished by always running CMS in the supervisor state so privileged instructions can be issued by the user.

CMS also provides the user utilizing this facility with an interface enabling him to use the CMS I/O interrupt handling routines WAIT and IOINT and the centralized CMS I/C error recovery IOERR. The interrupt handling interface, HNDINT, is described below. The I/O error interface is described under "Input/Output Service Routines".

System input/output activity to disk may be initiated without using an SIO instruction. This I/O can be performed by using a Diagnose instruction to signal CP-67 to perform certain disk I/O operations. See the section "Disk Handling Function Programs" for further details

## HNDINT

FUNCTION: The HNDINT function sets the CMS I/O interrupt handling routines to transfer control to a given location for an I/O device other than those normally handled by CMS, or to clear such transfer requests.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
        DS      0F
PLIST   DC      CL8'HNDINT'             called routine
        DC      CL4'SET' or CL4'CLR'    function
        IODEV   NAME, NUMBER, ADDRESS,
                  ASAP/WAIT-FLAG, KEEP/CLEAR FLAG
        .               .
        .               .
        .               .
        DC      X'FFFFFFFF'             end of list
```

ERROR CODES:

E(00001) Incorrect parameter list

MACRO IODEV: The IODEV macro sets up the following information in a 12-byte field:

```
NAME      =  Symbolic device name (1st 4 letters)
NUMBER    =  Hexadecimal device address
ADDRESS   =  Symbolic address of interrupt-handler to be invoked.  If
             address = 0, interrupts will be ignored when received.
ASAP/WAIT-FLAG:
    ASAP  =  Invoke interrupt-handler immediately.
    WAIT  =  Invoke interrupt-handler only when WAIT is called.
KEEP/CLEAR-FLAG:
    KEEP  =  Retain interrupt-handling between CMS commands.
    CLEAR =  Clear interrupt-handling after each CMS command.
             CLEAR = DEFAULT OPTION
Example:  IODEV  NEWD, 387, MYCODE, ASAP, KEEP
```

OPERATION: When an interrupt is received and processed by IOINT, it passes control to the interrupt-handler as follows:

| | |
|---|---|
| Register 0, 1 | I/O OLD PSW |
| 2, 3 | CSW |
| 4 | Device address |
| 14 | Return address to IOINT |
| 15 | Address of interrupt-handler |

When processing is complete, the interrupt-handler must return to IOINT via register 14, with Register 15 as follows:

R15 = 0 means SUCCESSFUL HANDLING
R15 = Nonzero means ANOTHER INTERRUPT EXPECTED

The general procedures for CMS I/O handling using HNDINT are as follows:

1. The program must initialize handling to be done via HNDINT SET.

2. When I/O to the appropriate device is to be done, the system-mask must be set OFF (by SSM instruction) and appropriate SIO given.

3. When SIO is performed satisfactorily, the system-mask can be set to allow all interrupts.

4a. If ASAP was specified, the interrupt-handler is invoked as soon as the interrupt is fielded by CMS IOINT. The interrupt-handler returns to IOINT, which returns to user's program.

4b. If ASAP was not specified, IOINT retains needed information until CMS WAIT function is called.

5. When program cannot proceed until the interrupt has been received, CMS WAIT function is called. If interrupt has not yet been received, CMS goes in WAIT state until IOINT fields and processes the interrupt in the normal way.

   If the interrupt has been received and processed (for example, on ASAP), WAIT returns to caller with necessary internal flags cleared.

   If the interrupt has been received but not yet processed (as under WAIT option instead of ASAP), CMS WAIT now calls IOINT to invoke desired interrupt-handler, then clears needed flags and returns to caller.

6. When finished, using program should normally clear the interrupt-handling scheme thru HNDINT CLR call (unless KEEP option is used and the interrupt-handler remains intact in core).

SVC SIMULATION

The SVC interruption handler (INTSVC), in addition to processing the special SVC X'CA' supervisor call instruction (refer to ''Internal Linkage Scheme''), also will transfer control to routines that will simulate various Operating System/360 supervisor functions. The simulation is required to enable the language processing programs, which make extensive use of Operating System/360 macro instructions, to function in the CMS environment. (Many Operating System/360 macro Instructions expand into SVC's).

These functions are simulated to yield the same results as seen from the processing program, as specified by OS program logic manuals. However, they are supported only to the extent stated in CMS documentation and to the extent necessary to successfully execute OS language processors. The user should be aware that restrictions to OS functions as viewed from OS exist in CMS.

The OS Functions that CMS will simulate are:

OS Simulation Under CMS

The OS simulation routines provide the CMS supervisory and file management functions necessary to support Assembler F, FORTRAN G compiler, FORTRAN G text decks, FORTRAN G library routines, PL/1 compiler, PL/1 text decks, and PL/1 library routines.

Since the OS simulation routines are CMS routines (i. e., they are not OS routines), there is no guarantee that jobs, other than those listed above, that run under OS will also run under CMS.

The following Operating System/360 functions are simulated by CMS:

| SVC<br>Number | OS<br>Function | Simulation<br>Routine | Usage |
|---|---|---|---|
| 00 | *XDAP | SOSVCTR | Used to access SYSUT1 |
| 01 | WAIT | SOSVCNU | wait for an I/O completion |
| 02 | POST | SOSVCNU | post the I/O completion |
| 03 | RETURN | SOLINKS | return from a LINK-to routine |
| 04 | GETMAIN | SOMAIN | conditionally acquire user free storage |
| 05 | FREEMAIN | SOMAIN | release user-acquired free storage |
| 06 | LINK/GETPOOL | SOLINKS | link control to another load phase |
| 07 | XCTL | SOLINKS | release, then link control to another load phase |
| 08 | LOAD | SOLINKS | read into core another load phase |
| 09 | DELETE | SOLINKS | delete a loaded phase |
| 10 | GETMAIN<br>FREEMAIN<br>FREEPOOL | SOMAIN | manipulate user free storage |
| 11 | *TIME | SOSVCTR | get the time of day |
| 13 | ABEND | SOABEND | abort processing, and enter DEBUG |
| 14 | *SPIE | SOSVCTR | allow a processing program to decipher program interrupts |
| 18 | *BLDL/FIND | SOSVCTR | manipulate simulated partitioned data files |
| 19 | OPEN | SOOPCL | activate a data file |
| 20 | CLOSE | SOOPCL | deactivate a data file |
| 21 | *STOW | SOSVCTR | manipulate partitioned directories |
| 22 | OPENJ | SOOPCL | activate a data file |

| SVC Number | OS Function | Simulation Routine | Usage |
|---|---|---|---|
| 23 | TCLOSE | SOOPCL | temporary deactivate a data file |
| 24 | *DEVTYPE | SOSVCTR | obtain device-type physical characteristics |
| 35 | *WTO/WTOR | SOSVCTR | communicate with the console |
| 40 | *EXTRACT | SOSVCTR | effective NOP |
| 41 | *IDENTIFY | SOSVCTR | effective NOP |
| 44 | *CHAP | SOSVCTR | effective NOP |
| 46 | *TTIMER | SOSVCTR | effective NOP |
| 47 | *STIMER | SOSVCTR | effective NOP |
| 48 | *DEQ | SOSVCTR | effective NOP |
| 51 | ABDUMP | SOABEND | (same as ABEND) |
| 56 | *ENQ | SOSVCTR | effective NOP |
| 57 | *FREEDBUF | SOSVCTR | release a free storage buffer |
| 60 | *STAE | SOSVCTR | allow processing program to decipher abort condition |
| 62 | *DETACH | SOSVCTR | effective NOP |
| 63 | *CHKPT | SOSVCTR | effective NOP |
| 64 | *RDJFCB | SOSVCT2 | obtain information from FILEDEF command |
| 68 | *SYNAD | SOSVCT2 | handle data set error conditions |
| 69 | *BACKSPACE | SOSVCT2 | backup a record on a tape or disk |
| — | GET/PUT | SOQSAM | access system-blocked data |
| — | READ/WRITE | SOBSAM | access system-record data |
| — | NOTE/POINT | SOCNTRL | access or change relative track address |
| — | CHECK | SOCNTRL | test ECB for completion and errors |

* These SVCs are simulated in SOSVCTR. This routine is loaded into the transient area whenever one of these SVCs is issued.

## Operating System/360 SVC Simulation Routines

CMS provides a number of routines to simulate certain Operating System/360 functions used by programs such as the Assembler and the FORTRAN and PL/I compilers. Some of the SVC simulation routines are located in the disk resident modules SOSVCTR and SOSVCT2. Whenever one of the SVC routines in SOSVCTR or SOSVCT2 is invoked, that routine is loaded into the transient area. The following paragraphs describe how these simulation routines work.

XDAP-SVC 0: Used by OS compilers to read the source code spill file, SYSUT1

WAIT-SVC 1: This routine (WAITX) receives control when a WAIT macro instruction is issued. When it gets control, WAITX tests the completion bit in the ECB. If the bit is on, indicating that the event being waited for is complete, it returns to the calling program. If the bit is off, a wait state PSW is loaded and waits for the bit to be turned on, at which time it returns to the calling program.

POST-SVC 2: POST will set bits in the event control block (ECB) to signify termination of an I/O operation upon receiving an I/O interrupt from the specific device.

RETURN-SVC 3: To return from a load module that was given control by a LINK or XCTL call, SVC 3 may be used. SVC 3 will cause control to be passed to the RETURN entry point in the core resident routine, SOLINKS. If the load module was LOADMOD'ed by CMS, the chaining stack will be updated, the PSW at the time of the LINK or XCTL will be loaded, and the address of the returning phase will be deleted from the stack. If the load module was dynamically loaded by the relocation of the object code, transfer is made to DEL4. All entries, in the load tables (LDRTBL) in high-numbered core, that are a result of the returning phase having been loaded, will be blanked out. The core storage that was obtained to load the phase will be returned to the system pool of free storage by a call to FRET.

GETMAIN-SVC 4: Control is passed to the GETMAIN entry point in the SOMAIN core resident routine. The mode is determined: VU, VC, EC. A call is made to GETBLK to obtain the block of storage. General storage maintenance is described elsewhere in this manual. Control blocks of two fullwords precede each section of available core: (1) the address of the next block, (2) the size of this block. The head of the pointer string is located at the words FSTFRE - initial free block, LENFRE - size of initial block, FRELST - address of first link in chain of free block pointers.

FREEMAIN - SVC 5: Releases a block of free storage. If the block is part of segmented core, a control block of two full words is placed at the beginning of the released area. Adjustment is made to include this block in the chain of available areas.

LINK-SVC 6: Program transfer is controlled by the nucleus routine SOLINKS. The LINK macro causes program control to be passed to a designated phase. If any X'80' bit within the word SWITCH is on, loading will consist of LOADMODing a CMS MODULE into core. If all bits are off, dynamic loading will be initiated. The TEXT deck of the desired phase will be located, and the first ESD card will be scanned (in DYNALOAD) to obtain the length of the CSECT. A GETMAIN will be issued to obtain enough core storage so that the loader (LDR) may relocate the phase into core. If other text decks or library routines are needed to complement the desired phase, a GETMAIN is issued for each segment length. A chain of pointers is built to record the old SVC PSW, the entry point of the new phase, and the loader table entries caused by the new phase.

GETPOOL: GETPOOL routine is entered via a link SVC 6 with an entry point of IECQBFG1 whenever the user issues a GETPOOL macro. The routine gets core and builds the pool of buffers. GETMAIN is issued to obtain the storage necessary for the number and size of buffers requested, and a chain of buffer pointers is built whose address is placed in the DCB passed by the caller.

XCTL-SVC 7: XCTL determines whether LOADMOD or dynamic loading is required for loading the module to which XCTL will transfer control after loading is complete.

LOAD-SVC 8: Control is passed to SVC08 located in SOLINKS when a LOAD macro is issued. Upon entry, SVC08 determines if the CMS overlay structure is in effect. If it is, LOADMOD is called to read a CMS MODULE. If dynamic loading is desired, a CMS TEXT file is loaded. Control then passes back to the user.

DELETE-SVC 9: DELETE removes all references caused by the specified module from the loader tables; and frees acquires main storage if the dynamic loader was in effect.

GETMAIN/FREEMAIN-SVC 10: Control is passed to the SVC 10 entry point in SOMAIN. Storage management is analogous to SVC 4/5 respectively.

TIME-SVC 11: This routine (TIME) located in SOSVCTR receives control when a TIME macro instruction is issued. A call is made to the pseudo timer device, X'OFF'. The real time of day and date are returned to the calling program in a specified form: decimal (DEC) binary (BIN), or timer units (TU).

ABEND-SVC 13: This routine (SOABEND) receives control when either an ABEND macro or an unsupported OS/360 SVC is issued. If an SVC 13 was issued, a check is made to see if there are any outstanding STAE requests. If not, or if an unsupported SVC was issued, TYPLIN is called to type a descriptive error message at the terminal. Next, CONWAIT is called to wait until all terminal activity has ceased, and then, control is passed to DEBUG. If a STAE macro was issued, a STAE work area is built and control is passed to the STAE exit routine. After the exit routine is through, a test is made to see if a retry routine was specified. If so, control is passed to the retry routine. Otherwise control passes back to the user.

SPIE-SVC 14: This routine (SPIE) receives control when a SPIE macro instruction is issued. When it gets control, SPIE inserts the new program interruption control area (PICA) address into the program interruption element (PIE). The program interruption element resides in the program interruption handler (PRGINT). It then returns the address of the old PICA to the calling program, sets the program mask in the calling program's PSW, and returns to the calling program.

BLDL/FIND-SVC 18: See BLDL and FIND under description of BPAM routines.

STOW-SVC 21: See STOW under description of BPAM routines.

OPEN/OPENJ-SVC's 19/22: OPEN will simulate the data management function of opening one or more files. It is a nucleus routine and receives control from SVCint when an executing program issues an OPEN macro instruction. See SOOPCL for a description of its operation.

CLOSE/TCLOSE-SVC's 20 and 23: CLOSE and TCLOSE are simulated in the nucleus routine SOOPCL. It receives control whenever a CLOSE or TCLOSE macro instruction is issued. See SOOPCL for a description of its operation.

DEVTYPE-SVC 24: This routine (DEVTYPE), located in SOSVCTR, receives control when a DEVTYPE macro is issued. Upon entry, DEVTYPE moves Device Characteristic Information for the requested data set into a user specified area, and then returns control to the user.

WTO, WTOR--SVC 35: This routine (WTO), located in SOSVCTR, receives control when either a WTO or a WTOR macro instruction is issued. For a WTO, it constructs a calling sequence to the TYPLIN function program to type the message at the terminal. (The address of the message and its length are provided in the parameter list that results from the expansion of the WTO macro instruction.) It then calls the CONWAIT function program to wait until all terminal I/O activity has ceased. Next, it calls the TYPLIN function program to type the message at the terminal and returns to the calling program.

For a WTOR macro instruction, this routine proceeds as described for WTO; however, after it has typed the message at the terminal it calls the WAITRD function program to read the user's reply from the terminal. When the user replies with a message, it moves the message to the buffer specified in the WTOR parameter list, sets the completion bit in the ECB, and returns to the calling program.

EXTRACT-SVC 40: This routine (EXTRACT), located in SOSVCTR receives control when an EXTRACT macro is issued. Upon entry, EXTRACT clears the first word of the user provided answer area and returns control to the user.

IDENTIFY-SVC 41: IDENTIFY is a NOP located in SOSVCTR.

CHAP-SVC 44: CHAP is a NOP located in SOSVCTR.

TTIMER-SVC 46: TTIMER signals zero time remaining and signals no errors (in effect a NOP). TTIMER is located in SOSVCTR.

STIMER-SVC 47: STIMER is a NOP located in SOSVCTR.

DEQ-SVC 48: DEQ is a NOP located in SOSVCTR.

ABEND-SVC 51: See ABEND-SVC 13.

ENQ SVC 56: ENQ is a NOP located in SOSVCTR.

FREEDBUF-SVC 57: This routine (FREEDBUF) located in SOSVCTR receives control when a FREEDBUF macro is issued. Upon entry, FREEDBUF sets up the correct DSECT registers and calls the FREEDBUF routine in SOBDAM. This routine returns the dynamically obtained buffer (BDAM) specified in the DECB to the DCB buffer control block chain. Control is then returned to the SOSVCTR routine which returns control to the user.

STAE-SVC 60: This routine (STAE) located in SOSVCTR receives control when a STAE macro is issued. Upon entry, STAE creates, overlays or cancels a STAE control block (SCB) as requested. Control is then returned to the user with one of the following return codes in register 15.

| Code | Meaning |
|---|---|
| 00 | An SCB is successfully created, overlaid or cancelled. |
| 08 | The user is attempting to cancel or overlay a non-existent SCB. |

Format of SCB

| | |
|---|---|
| 0 | 0 or pointer to next SCB |
| 4 | exit address |
| 8 | parameter list address |
| 12 | |

DETACH-SVC 62: DETACH is a NOP located in SOSVCTR.

CHKPT-SVC 63: CHKPT is a NOP located in SOSVCTR.

RDJFCB-SVC 64: This routine (RDJFCB) receives control when a RDJFCB macro instruction is issued. When it gets control, RDJFCB obtains the address of the JFCB from the DCBEXLST field in the DCB and sets the JFCB to zero. It then determines if a FORTRAN object program - not the FORTRAN compiler - is being executed. (If a FORTRAN object program is being executed, a switch given by the system reference table (SYSREF) in the nucleus constant area (NUCON) will be set on. This switch is set on by the FORTRAN object-time I/O program, IXCCMS, at the start of execution of the object program.) If a FORTRAN object program is not being executed, RDJFCB returns to the calling program. If such a program is being executed; RDJFCB calls the STATE function program to determine if the associated file exists. If it does, RDJFCB returns to the calling program. If the file does not exist, RDJFCB sets a switch in the DCB to indicate this and then returns to the calling program. RDJFCB is located in SOSVCT2.

Note: The switch set by the RDJFCB is tested by the FORTRAN object-time direct-access handler (DIOCS) to determine whether or not a referenced disk file exists. If it does not, DIOCS will initialize the direct access file.

SYNAD-SVC 68: Located in SOSVCT2, SYNAD simulates the functions SYNADAF and SYNADBLS. SYNADAF expansion includes an SVC 68 and a high-order byte in register 15 denoting an access method. SYNAD will prepare an error message line and swap save areas and register 13 pointers. The message buffer is 120 bytes: bytes 1-40, blank; bytes 41-120, "BSAM/QSAM INPUT/OUTPUT ERROR nn CN FILE: "dsname"; where nn is the CMS RDBUF/WRBUF error code, or the residual count, if an error is encountered.

SYNADRIS expansion includes SVC 68 and a high order byte of X'FF' in register 15. The save area will be returned, and the message buffer will be returned to free storage.

BACKSPACE-SVC 69: Also in SOSVCT2. For a tape, a BSR command is issued to the tape. For a direct access data set, the CMS write and read pointers are decremented by one.

GET/PUT — See SOQSAM for a description.

READ/WRITE — See SOBSAM for a description.

NOTE/POINT — See NOTE and POINT for descriptions.

CHECK — See CHECK for a description.

Notes on using the OS simulation routines:

* CMS files are physically blocked in 800-byte blocks, and logically blocked according to a logical record length. If the filemode of the file is not 4, the logical record length is equal to the DCB LRECL — and the file must always be referenced with the same DCB LRECL, whether or not the file is blocked. If the filemode of the file is 4, the logical record length is equal to the DCB BLKSIZE — and the file must always be referenced with the same DCB BLKSIZE.

* To set the READ/WRITE pointers for a file at the end of the file, a FILEDEF command must be issued for the file specifying the MOD option.

* A file will be erased and a new one created if the file is opened and the following conditions exist:

a. the OUTPUT option of OPEN is specified.

b. the TYPE option of OPEN is not J.

c. the dataset organization option of the DCB is not direct access.

d. a FILEDEF command has not been issued for dataset specifying the MOD option.

## SOOPCL

FUNCTION: To process OPEN and CLOSE macros.

EXIT CONDITIONS: If an OPEN is successful, control is returned to the user with the DCBOFLGS OPEN bit on. If an OPEN fails for one of the reasons listed below, the DCBOFLGS OPEN bit is turned off. The following message is on the console and control is returned to the user.

OPN-CONFLICTING XXXXXXXX PARAMETERS
XXXXXXXX is equal to the
DCB DDNAME of the DCB that failed to open.

REASONS WHY AN OPEN MIGHT FAIL.

1. The data set organization is not physical sequential, partitioned or direct access.

2. No LRECL, BLKSIZE or BUFL is filled in.

3. BLKSIZE conflicts with LRECL and RECFM.

4. Default FILEDEF issued by OPEN failed.

5. RECFM does not agree with the format of the existing file.

6. RECFM is fixed and LRECL does not agree with the record length of the existing file or if filemode is 4 the BLKSIZE does not agree with the record length of the existing file.

CALLS TO OTHER ROUTINES: FILEDEF STATE ERASE
TYPEIN TAPEIO CLOSIO
FINIS

CALLED BY: OS OPEN or CLOSE macro

OPERATION: OPEN (SVC 22) and OPENJ (SVC 19)

Initialization: On entry to SOOPCL, IOTYPE is set to indicate OPEN or OPENJ and the address of the current DCB is obtained from the list pointed to by Register 1.

Determination of Access Method: The data set organization (DCBDSORG) switch is checked to see if it is either physical sequential, partitioned, or direct access (effectively eliminating only ISAM). If none of the above, the DCB will not be opened.

Next the macro format field (DCBMACRF) is checked to see which access method is requested, and the access method indicator (DCBCIND2) is set to signal QSAM or BSAM.

QSAM: If the access method is QSAM, DCBMACRF is tested for a GET or PUT request, and the relevant routine address is placed in the corresponding DCB access field (note that GET and PUT are in the CMS routine SOQSAM).

BSAM: If the access method is BSAM, the address of SOBSAM is placed in DCB access field; the CHECK address is placed in the DCB check field; and if POINT is requested, the SOCNTRL address is placed in the DCBNOTE field.

Setting up DCB fields: After the relevant QSAM or BSAM processing, a check is made to see if CONTROL is specified and if it is, the CNTRL address is placed in the DCBCNTRL field. Next the CMSCB chain is tested to see if there is a CMSCB for this DCB, that is, if a FILEDEF command for the respective data set has been issued.

If one does not exist, the assumption is made that the user has set up the required DCB fields, and FILEDEF is called to create a CMSCB with a filename of FILE, a filemode of P1, and a filetype equal to the DCB DDNAME after a matching CMSCB is found or created, and it is used to fill in vacant entries in the DCB.

The following table shows the CMSCB fields that are used to complete DCB fields not initialized by the user prior to issuing the OPEN call. It also shows the JFCBMASK bit setting which is on if the associated CMSCB field must be used. If BLKSIZE, DSORG, or RECFM are not specified by a FILEDEF command, the defaults of 80, PS (sequential), and F (fixed) are used to fill in the respective fields of the CMSCB.

| DCB | FCB | JFCB DEFAULTS | JFCBMASK |
|---|---|---|---|
| DCBBLKSI | JFCBLKSI | 80 | X'00001000' |
| DCBDSORG | JFCBSORG | PO | X'00000001' |
| DCBLRECL | JFCLRECL | 0 | X'00000002' |
| DCBRECFM | JFCRECFM | F | X'00000400' |
| DCBKEYLE | JFCKEYLE | 0 | X'00000020' |
| DCBOPTCD | JFCOPTCD | 0 | X'00008000' |
| DCBLIMCT | JFCLIMCT | 0 | X'00004000' |

Setting up a new CMSCB: If it is necessary to set up and initialize a new CMSCB for the DCB currently being opened, free storage space is obtained and cleared. The CMSCB chain is updated to reflect the addition and the following fields are filled in:

| FIELD | CONTENT | DESCRIPTION |
|---|---|---|
| FCBSECT | X'08' | indicates OPEN acquired this CMSCB. |
| FCBDEV | X'14' | disk default |
| FCBDSNAM | filename | CMS filename |
| FCBDD | DCBDDNAM | CMS filetype |
| FCBDSTYP | DCBDDNAM | CMS filetype |
| FCBDSMD | file mode | CMS filemode |

Setting Up Control Block Pointers: After the CMSCB is initialized, the address pointers are set to link the various simulated control blocks.

| Control Block | Field | Contents after completion |
|---|---|---|
| DCB | DCBDVTBL | CMSCB address |
| DCB | DCBDEBAD | DEB address |
| DCB | DCBIOBAD | IOB address chained scheduling |
| DCB | DCBIOBA | IOB address normal scheduling |
| DCB | DCBIOBL | Length in double words of IOB |
| ICB | IOBDCBPT | DCB pointer |
| DEB | DEBDCBAD | DCB pointer |
| DEB | DEBDEBID | X'0F' flag to show block is DEB |
| DEB | DEBOPATB | Open option byte |

File Verification: The DEB option byte (DEBOPATB) is checked and if outin has been specified control passes to the user exit processing routine (EXITLIST). If the file device type for the current DCB is not disk, control passes to EXITLIST.

If the device type is disk, its current status is checked. The table below shows the action taken.

| File Condition | Action |
|---|---|
| Non-existent | New file to be written |
| present and read/write pointers set to one | erase old copy, for new file to be written |
| present and write pointer not equal to one | new information will be appended to file |

EXITLIST — User Exit Processing Routine: If the exit list field (DCBEXLST) is empty, control passes immediately to VEROPEN - the verification routine for record format dependent quantifiers.

If DCBEXLST contains a code other than X'05', checking continues until an end-of-list tag is found, at which time control returns to VEROPEN, or until an X'05', is found, in which case the DCBOFLGS are locked on and a branch is taken to the user DCB exit processing routine. On return exit conditions are restored and the possible existence of further requests is checked.

VEROPEN — Validate Contents of Record Format Dependent Fields: Tests DCBRECFM to find the record format, then tests various fields to validate the contents and sets up record description fields in accord with what it finds.

Each of the following tables attempts to show the kind of validation which is required by the particular format. After completion of control block analysis and set up, control passes to BUFFPOOL to handle buffer pool requirements.

DCBRECFM = FIXED, UNBLOCKED

| Fields Filled In | | | | Action |
|---|---|---|---|---|
| DCBLRECL | DCBBLKSI | DCBBUFL | NONE | Assignment is in the arrow direct |
| x | | | | LRECL → BLKSIZ → BUFL |
| | x | | | BLKSIZ → LRECL → BUFL |
| | | x | | BUFL → LRECL → BLKSIZ |
| | | | x | Error Exit, R15=2 |

DCBRECFM = FIXED, BLOCKED

Since blocksize is a multiple of logical record length, either blocksize or buffer length must be specified.

| Fields Filled In | | | | Action |
|---|---|---|---|---|
| DCBBLKSI | DCBBUFL | DCBLRECL | NONE | Assignment is in arrow direction |
| x | x | | | BLKSIZ → BUFL<br>BUFL → BLKSIZ |
| | | | x | Error exits R15=3 |
| No DCBLRECL specified | | | | BLKSIZ → (BUFL) → LRECL |
| DCBLRECL specified but not a multiple of BLKSIZ (BUFL) | | | | Error exit R15=4 |

DCBRECFM = VARIABLE

There must be either a buffer length or a blocksize and the chosen field must be longer than the 4-byte block description word.

| Fields Filled | | | Action |
|---|---|---|---|
| DCBBLKSI | DCBBUFL | NONE | Assignment is in arrow direction |
| x | | | DCBBLKSI → BUFL |
| | x | | BUFL → BLKSIZ |
| | | x | Error Exit R15=5 |
| Blksize < 4 bytes | | | Error Exit R15=6 |

DCBRECFM Variable

| | |
|---|---|
| Is BLKSIZ-4 an integral multiple of lrecl? | NO → BLKSIZ= (N*LRECL)+4 |

DCBBRECFM = UNDEFINED

Takes the largest value of LRECL, BUFL, or BLKSIZ and uses it to set the others.

Final merging occurs as follows for all formats: DCBBLKSI FILLED IN?

```
YES — is LRECL larger than BLKSIZE?
        YES — LRECL → BUFL → BLKSIZ
        NO — BLKSIZ → BUFL → LRECL

NO — Is LRECL filled in?
        YES — LRECL → BLKSIZ → BUFL
        NO — Is BUFL filled in?
                YES — BUFL → LRECL → BLKSIZ
                NO — BSAM, BPAM?
                        NO — Error exit R15, x'01'
```

Buffpool: If user does not supply a buffer pool, parameters for the GETPOOL macro are set up by examining DCBBUFNO and DCBFUFL. If BUFNO is not specified, the default value of one is used. If the length is not specified, and the method is BSAM, the buffer pool acquisition is ignored; if QSAM, an error exit is taken. The GETPOOL macro is issued.

BUCN3: After a buffer pool has been either verified or obtained by GETPOOL, the address of the first buffer in the chain is stored in IOBSTART. The address of the first buffer to be used (same address) is stored in DCBRECAD. If the method is QSAM and the format is variable, the address is adjusted to eliminate the BDW.

The same address as for IOBSTART is placed in IOBNXTAD as initial condition of next buffer and in DCBEOBAD as initial end of block condition. A 1 is inserted in the high order byte of DCBEOBAD as the ID of the next buffer to be used.

If the method is QSAM, put-locate mode, the address of the next buffer is placed in DCBEOBAD. A 2 is set in IOBSTART as ID of next buffer to be used, unless there are no more.

SETEOD: If the user has not specified EOD and SYNAD addresses, the standard EOD and SYNAD address are placed in DCBEODAD and DCBSYNAD respectively.

OPENED: DCBOFLGS is set to indicate that the DCB has been opened successfully and return is set to INTSVC if there are no more DCBs to be processed. Otherwise, return is to COMOPEN.

OPERATION: CLOSE (SVC 20) and TCLOSE (SVC 23)

Initialization: IOTYPE is set to indicate CLOSE or TCLOSE.

COMCLOSE: After checking to make sure that the particular DCB has actually been opened, the address of the CMS Control Block is obtained from DCBDVTBL, FCBIOSW is set to indicate closing in process, and DCBOFLGS are set to ''busy''. If the access method used is QSAM, put-locate, the last record must be outputted and control is passed to SOQSAM-PUT. If the requested file disposition was LEAVE, FCBIOSW is set to indicate this. Then the FCBDEV is checked for device type code and the appropriate device routine is branched to:

TAPE: If the file disposition was LEAVE, the routine goes off to CLOSE2—the common close routine. If not, the tape is rewound before going off to the common close routine.

UNIT RECORD: A CLOSIO is issued to the device—printer, punch, or reader—and a branch taken to common close.

CONSOLE: Go to common close.

CLOSE2: If IOTYPE is T, control is passed to CLOSED. Otherwise, a check is made to see if FCBPOS or FCBKEYS is zero. If not SOSVCTR is called to free and, if necessary, save any PDS or KEY table in core. Next DCBMACR, DCBIFLG, DCBDDNAM, DCBLRECL, DCBRECFM, DCBDSORG, DEBCIND2, DCBKEYLE, DCBDETCD and DCBLIMCT are restored to their status before OPEN and control is passed to CLOSED.

CLOSED: The DCB list pointer is restored and, if this was the last DCB, the routine returns to the user. If not, the routine returns to COMCLOSE and proceeds to close the next DCB.

SOQSAM (CMS Queued Sequential Access Method)

FUNCTION: To analyze record format and set up the buffers accordingly for GET and PUT requests.

CALLS TO OTHER ROUTINES: SOEOB

CALLED BY: GET or PUT macro

OPERATION:

Initialization: IOTYPE and IOBERBPT are set to indicate access method and FCB address respectively. Then a branch is taken to GETTER or PUTTER, depending on whether the request is a GET or a PUT.

GETTER: After FCBIOSW and IOBIOFLG are set to indicate input in process, DCB fields DCBMACF and DCBRECFM are analyzed to determine the type of move desired - move or locate - and the record format. If the mode is MOVE, the user specified 'move to' address is stored in DEBTCBAD. Both modes continue by determining whether end of block conditions exist.

EOB = no: If EOB does not exist, DCBBLKSI is accessed to obtain the record length and SOEOB is called to get a record. On return, if the end of data set has been reached, then DCBEODAD is accessed and control passed to that address. If any other error code is detected, control is passed to the address in the DCBSYNAD field.

On a good return from SOEOB, the following actions are taken:

Locate mode, fixed format — Reset DCBRECAD and return to user.

Locate mode, undefined format — Set DEBLRECL equal to DCBBLKSI, reset DCBRECAD and return to user.

Locate mode, variable format — Set DCBLRECL from RDW field in record, reset DCBRECAD and return to user.

Move mode, all formats — Same procedure as in locate mode, except that just before returning to the user, the record is moved to the user buffer.

EOB = yes: It is necessary to obtain a new buffer from the buffer pool, which has been initialized by SOOPCL.

Upon Entry:
A (DCBBUFCB) = A (BUFCB)
where the buffer number (DCBBUFNO) is the high order byte of the buffer, and the buffer length is the halfword following DCBBUFCB.
A (IOBNXTAD) = A (NEXT VALID BUFFER TO-BE-USED)
A (IOBSTART) = X'ID OF NEXT BUFFER', AL3 (INITIAL BUFFER IN BLOCK)

During:
The ''NEXT'' buffer becomes the ''CURRENT'' buffer.
The ''CURRENT'' buffer + BUFL = ''NEXT'' buffer.
The ID + 1 = the ID of the ''NEXT'' buffer that will be used.
If ID > BUFNO, ID is set = 1; and A (IOBNXTAS)=A (IOBSTART).

On completion processing proceeds as in EOB = No.

SOBSAM (CMS Basic Sequential Access Method)

FUNCTION: The CMS BSAM routine processes sequential READ and WRITE macros. All the OS macro options are supported except those dealing with spanned records.

EXIT CONDITIONS: The SOBSAM routine passes control back to the user with the following error codes in the ECB and a zero in register 15:

| | ECB Code | Register 15 |
|---|---|---|
| Successful Completion | 7F | 0 |
| Unsuccessful Completion | 42 | 0 |
| End of EXTENT | 7F | 8 |

CALLS TO OTHER ROUTINES: SOBDAM, PDSSAVE, SOECB

CALLED BY: OS READ or WRITE macro

OPERATION: The CMS BSAM routine is called by an OS READ or WRITE macro. It checks DCBFDAD to see if the first byte is a P. If so, the contents of the last two bytes of DCBFDAD are incremented by one and stored in FCBITEM. Next the DSORG option is checked.

- If the DSORG option in the DCB is DA (Direct Access), control is given to the SOBDAM routine to convert the record identification into an item number and process any keys used. If SOBDAM completes successfully, control is returned to BSAM. Otherwise, control is returned to the user.

- If the DSORG option in the DCB is PO (Partitioned Organization), and a write is specified, and the FCBPDS entry is zero, control is passed to the PDSSAVE routine to save the directory of the PDS (Partitioned Data Set) and point the FCB file item number to a free member slot. If PDSSAVE completes successfully, control is returned to SOBSAM. Otherwise, control is returned to the user.

- If the DSORG option in the DCB is PS (Physical Sequential) and the MACRF option is WL (create a BDAM data set)*, an eight is put in register 15 and a check is made to see if end of EXTENT has been reached. If so, control is returned to the user. If not, register 15 is set to zero and a check is made of the option specified in the WRITE macro's DECB. If SZ is specified, control is returned to the user with a hex '7F' in the ECB. If SD is specified, SOBDAM is called to write a dummy key and upon return from SOBDAM control is passed back to the user with a hex '7F' in the ECB. If SF is specified, a hex '7F' is stored in the EOB and if the keylength is not zero SOBDAM is called to process a key. If the SOBDAM routine and/or the check for valid options is completed successfully SOBSAM begins filling in the IOB and the WRBUF PLIST. Otherwise control is returned to the user with a hex '42' in the ECB, denoting an error.

- If the DSORG option in the DCB is PS or PO, a write is specified, and the MACRF option in the DCB is not WL, a check is made to see if the keylength is zero. If not, SOBDAM is called. If the SOBDAM routine and or the check for valid options is completed successfully, SOBSAM begins filling in the IOB and the WRBUF PLIST. Otherwise control is returned to the user with a hex '42' in the ECB.

- After the necessary checks and calls to SOBDAM and PDSSAVE are made, SOBSAM fills in the IOBIOFLG bit, the IOTYPE byte, the DCBOFLGS bit, the buffer length if the record format is not fixed, the buffer address, the DECB I/O started bit, the IOB pointer in the DECB and the ECB pointer in the IOB. Control is then passed to the SOEOB routine to do the I/O and fill in the ECB. After control is passed back to SOBSAM from SOEOB, control is passed back to the user.

* If the WL (create a BDAM data set) option is specified, the number of records in the data set extent must be specified using the FILEDEF command. The default size is 50 records.

*NOTE (BSAM and BPAM) — supported for DISK only

FUNCTION: To return in register one the relative position of the last block read from or written into a data set. The return format of register 1 is the same as in OS.

EXIT CONDITIONS: Control is returned to the user.

CALLS TO OTHER ROUTINES: None

CALLED BY: OS NOTE macro

OPERATION: Upon entry to NOTE, a check is made to see if a POINT was just issued. If not, the item number of the next record to be processed is loaded from FCBITEM into register 1, register 1 is decremented by 1, and control is returned to the user. If a POINT was just issued, register 1 is loaded with the value in DCBFDAD and control is returned to the user.

* The NOTE routine is part of the SOCNTRL routine.

*CHECK (BDAM, BSAM, BPAM)

FUNCTION: To check for errors or exceptional conditions on a previous READ or WRITE. If the previous READ or WRITE completed successfully, control is returned to the user. If not, the error analysis (SYNAD) routine is given control, or, if no error analysis routine is provided, the task is abnormally terminated.

EXIT CONDITIONS: If no error flags are set in the ECB or DECB, control is returned to the user. If the ECB input end-of-data (EODAD) flag is set, control is given to the EODAD routine. If other DECB or ECB error flags are set, the error analysis (SYNAD) routine is given control or if no error analysis routine is provided the task is abnormally terminated.

CALLS TO OTHER ROUTINES: EODAD routine, SYNAD routine, DEBUG

CALLED BY: OS CHECK macro

OPERATION: CHECK is called by an OS CHECK macro. Upon entry the DECB or ECB is tested for I/O errors. If there are no errors, control is returned to the user. If an ECB end-of-data-set flag is on, the EODAD routine is given control. If there is an error other than end-of-file, the SYNAD exit routine is called. If no SYNAD exit is given, DEBUG is entered.

* The CHECK routine is part of the SOCNTRL routine.

*POINT (BDAM and BSAM—supported for DISK only)

FUNCTION: The point macro causes processing for a data set to start at a specified block in the data set. The format of the block address must be the same as in OS.

EXIT CONDITIONS: Control is returned to the user.

CALLS TO OTHER ROUTINES: None.

CALLED BY: OS POINT macro

OPERATION: The POINT routine is called by the user. Upon entry, the relative block address is loaded into a register and right adjusted. If the rightmost byte of the block address is not set to one, a one is subtracted from the register. The content of the register is stored in DCBFDAD. Control is then returned to the user.

* The POINT routine is part of the SOCNTRL routine.

## CMS BDAM (CMS BASIC DIRECT ACCESS METHOD)

FUNCTION: The CMS BDAM macro routine is used to access data set records directly by item number. It converts record identifications given by OS BDAM macros into item numbers and uses these item numbers to access records. The CMS BDAM macro routine supports all the Release 20 OS BDAM macro functions except those listed as restrictions.

EXIT CONDITIONS: If ID, KEY, BUFFER, LIMIT, SEARCH or I/O errors occur, they are reflected in the DECB and control is returned to the SOBSAM routine which returns control to the user. The error codes correspond to OS error codes and are listed below.

Error Codes put in DECB + 1

| | | | |
|---|---|---|---|
| DUMMYERR | DC | X'1001' | Key to be added begins hex 'FF' |
| NOTFCUND | DC | X'1000' | The record was not found |
| IDTOBIG | DC | X'1010' | Record ID was more than 2 bytes |
| IOERR | DC | X'0800' | Uncorrectable I/O error |
| BADDCB | DC | X'1020' | DCB and macro entries conflict |
| NOBUFFER | DC | X'0200' | No buffers free |
| NOSPACE | DC | X'2000' | No space found |

CALLS TO OTHER ROUTINES: RDBUF, SVCFREE, WRBUF, FINIS, KEYSAVE

CALLED BY: SOBSAM

OPERATION: For Relative Block, Relative Track and Actual addresses, the low order two bytes of a record identification are used as an item number. For Relative Track Address and Actual Key, the low order byte of the relative track address is used to access a table of keys which, if not already in core, is brought in and searched for the correct key.

CMS does not support actual key I/O so the CMS BDAM routine simulates it. In CMS, all keys are kept at the end of their data file. When the data file is opened, two new files are created with the same filetype, but with filenames of $KEYTEMP and $KEYSAVE. Both these files contain all the keys in the original data file. $KEYTEMP is used for updating keys and $KEYSAVE is used to save all the keys in case of a re-IPL or system crash. For every item in the original file there is a corresponding key space in the $KEYTEMP file. Each item in the $KEYTEMP file is a key table that contains 256 keys. When the data file is closed, the $KEYTEMP file is written at the end of the data file, and the $KEYTEMP and $KEYSAVE files are erased.

The CMS BDAM routine gets control from the CMS BSAM routine which in turn gets control from an OS READ or WRITE macro. Upon entry to SOBDAM, a check is made to see if dynamic buffering is needed. If so, key buffer and or data buffer is acquired or returned depending on whether a read or write is requested. Next the relative or actual address is checked to make sure it does not exceed two bytes. This address is converted into an item number and, if keys are not involved, the feedback option is taken care of and control is passed back to SOBSAM. If keys are accessed and the key table containing the key wanted is not in core, it is brought in and searched or updated. If a search is specified, the item number of the key table containing the key is combined with the position number of the key in the table to form the item number of the data. If the extended search option is not specified, only one key table of 256 keys are searched. If the extended search option is specified, the limit parameter in the DCB is converted to a number of key tables and that number of tables are searched for a matching key. After the key table has been read, updated, or searched, the item number, if feedback is requested, is stored in the correct feedback address and control is returned to SOBSAM. Core for the key table and its control parameters is acquired the first time a key is accessed. The address of this core is stored in the FCB and the core is not freed until the data set is closed.

The format of the disk key table and the in-core key table and control words is described below.

Key Table

| keylength |
|---|
| 1st KEY |
| 2nd KEY |
| |
| nth KEY |

n is 256

In-Core Key Table and Control Words

```
KEYTABL      DSECT
KEYLNGTH     DS        1F                  Key length
ENDDATA      DS        1F                  Last data item in file
KEYOP        DS        2F                  Command Name
KEYNAME      DS        2F                  Filename of key file
KEYTYPE      DS        2F                  Filetype of key file
KEYMODE      DS        1H                  Filemode of key file
KEYTBLNO     DS        1H                  Item number of key table
KEYTBLAD     DC        A (KEYTABLE)        Address of key table
TBLLNGTH     DS        1F                  Byte size of key table
KEYFORM      DC        C'F'                Format of key table
KEYCHNG      DC        X'00'               Byte to signify change in key table
KEYCOUT      DC        X'0001'             Number of tables per item
             DS        1F                  Used by RDBUF for residual counts
KEYTABLE     DS        0F                  Table of keys
```

RESTRICTIONS: The four methods of accessing BDAM records:

1. Relative Block RRR
2. Relative Track TTR
3. Relative Track and Key TT Key
4. Actual Address MBBCCHHR

The restrictions on these methods:

- Spanned records are not supported in CMS.

- Only the BDAM identifiers underlined above can be used to reference records as CMS files have a two-byte record identifier.

- If BDAM methods 2, 3 or 4 are used and the RECFM is U or V, the BDAM user must not update the track indicator until a no space found message is returned on a write. For method 3 (WRITE ADD), this is when no more dummy records can be found on a WRITE request. For methods 2 and 4, this will not occur, and the track indicator will only be updated when the record indicator reaches 256 and overflows into the track indicator.

- Two files with keys and the same filetype cannot be open at the same time. If a program that is updating keys does not close the file it is updating for some reason, e.g., a system crash or a re-IPL, the original keys are saved in a temporary file with the same filetype and a filename of $KEYSAVE. To finish the update, run the program again.

- Once a file is created using keys, the file must not be added to without using keys and specifying the original key length.

*KEYSAV (BDAM or BSAM)

FUNCTION: To build a keys file when a data file using keys is opened and to save the keys at the end of the data file when it is closed.

EXIT CONDITIONS: Control is returned to caller with a zero in register 15 if execution was successful, and a nonzero, if not.

CALLS TO OTHER ROUTINES: FINIS FRET WRBUF RDBUF ERASE STATE

CALLED BY: SOOPCL SOBDAM

OPERATION: KEYSAV gets control from either SOBDAM or the CLOSE routine, SOOPCL.

- If KEYSAV gets control from SOBDAM, a key table and a PLIST for accessing the key table is built in core. Next, two new files with the same filetype as the data file, but with filenames of $KEYTEMP and $KEYSAVE are created, using the keys at the end of the data file. The $KEYTEMP file will be used for updates to the keys, and the $KEYSAVE file will be used in case of a system crash or re-IPL. If a $KEYSAVE file already exists for a data file when it is opened, then the keys from that file rather than the keys from the end of the data file, will be used to create $KEYTEMP. After the two files are created, control is returned to SOBDAM.

- If KEYSAV gets control from SOOPCL, then keys from the $KEYTEMP file are read in and written at the end of the data file. When this is complete, the $KEYTEMP and $KEYSAVE files are erased, the core for the key table and its PLIST is freed up, and control is returned to SOOPCL.

## CMS BPAM (BASIC PARTITIONED ACCESS METHOD)

The CMS BPAM macro routines are used to access and build Partitioned data sets. These data sets are divided into sequentially organized members, each of which has a unique name stored in a directory. The CMS BPAM macro routines support all the OS BPAM macro functions except the OS facility of adding user data to the directory entries.

The functions and operations of the CMS BPAM macro programs are given below.

### *FIND (BPAM)

FUNCTION: When called by the user: Causes the control program to use the address of the first block of a specified partitioned data set member as the starting point for the next READ macro instruction for the same data set.

When called by STOW or BLDL: Finds the directory entry for a member and pass back the in-core address of the entry.

When called by DICTSAVE: Reads in the directory

EXIT CONDITIONS: When control is returned to the problem program or calling routine, the return code in register 15 is as follows:

| Name Provided | Relative Address Provided |
|---|---|
| 00-successful execution | 00-at all times. |
| 04-name not found | If the relative address is bad |
| 08-permanent I/O error reading directory | it is reflected in the next READ. |

CALLS TO OTHER ROUTINES: RDBUF, SVCFREE

CALLED BY: OS FIND macro, BLDL, PDSSAVE, and STOW

STORAGE ALLOCATION:

| | |
|---|---|
| -nucleus | 0 |
| -control blocks | 24 + (12 times no. of entries in PDS) |

OPERATION: Upon entry to FIND, a check is made to determine if a relative address list was provided. If it was, the item number is obtained from the list and stored in FCB ITEM, and control is returned to the user with a zero in register 15. If a relative list was not provided, a search is made for the member name in the directory. If FCDDSTYP in the FCB is MACLIB, the name of the first macro library is moved from the Maclib list to FCBDSNAM and the address of the first macro library directory is loaded. Next a check is made of the FCBPDS entry in the FCB. If it is zero, the directory header record is read into a save area, SVCFREE is called to obtain core for the directory and its control words, the directory is read into core, and the pointer to the in-core directory is stored in FCBPDS. If, when the dictionary header record is read, the eighth character in it is a '$', a one is put in the change byte and the PDS directory is read from a file with the same filetype and a filename of $PDSTEMP. Once in, the directory is kept in core until a BLDL or a CLOSE is issued for the data set. After FIND has the pointer to the in-core directory, it begins searching for a matching member name. If the member name is not found, a check is made to see if any additional directory blocks have been added by STOW. If so, they are searched.

After the directory search is through and the member is either found or not found, a check is made to see from where the search was requested. If it was PDSSAVE, BLDL, or STOW, control is returned to the requesting routine. If it was a successful user request, the item number of the member is moved from the directory into FCBITEM and DCBRELAD, and control is returned to the user with a zero in register 15. If it was an unsuccessful user request and the FCBDSTYP in the FCB macro is not MACLIB, control is returned to the user with a four in register 15. If FCBDSTYP in the FCB macro is MACLIB, the next maclib name in the Maclib list is moved to the FCB, the address of the next maclib directory is loaded, and the search for the member starts again. If the next FCB pointer in the Maclib FCB list is zero, control is returned to the user with a four in register 15.

* ● There are two FIND routines. One is part of SOCNTRL and is used only when a relative address list is provided. The other is part of SOSVCTR.

● The DCBDSORG option in the DCB must always be PO when referencing a BPAM data set.

*BLDL (BPAM)

FUNCTION: To fill a users list in main storage with the relative track addresses (item numbers) for requested members.

EXIT CONDITIONS: When control is returned to the problem program, the return code in register 15 is as follows:

| Code (Hexadecimal) | |
|---|---|
| 00 | Successful completion |
| 04 | List could not be filled. The TTR field of the member not found is filled in as zero. |
| 08 | Permanent input or output error while reading in directory. |

CALLS TO OTHER ROUTINES: FIND, PDSSAVE

CALLED BY: OS BLDL macro

STORAGE ALLOCATION:

-nucleus 0
-control blocks 0

* The BLDL routine is part of SOSVCTR

OPERATION: Upon entry to BLDL, a zero is put in register 15 and a check is made to determine if the DCB DDNAME is TXTLIB. If it is, control is returned to the user.

If it is not, FIND is called to search the directory for a match of the first member name in the user's list. If a match is not found, the TTR field is filled in with zeroes, FIND is called to search for the next member, and a four is put into register 15. If it is found, BLDL fills in the users list with the member's item number and continues calling FIND until the entire BLDL list has been filled in. PDSSAVE is called to free the in-core directory and control is returned to the user. The format of the user's list after calling BLDL follows:

| 2 | 2 | 8 | 3 | 3 | 8 | 3 | 3 |
|---|---|---|---|---|---|---|---|
| FF | LL | NAME | TTR | KzC | NAME | TTR | KzC |

TTR the item number will always be right justified in these three bytes.
KzC These three bytes will always be zero.

## *STOW (BPAM)

FUNCTION: To add, change, replace or delete an entry in a Partitioned Data Set (PDS) Directory.

EXIT CONDITIONS: When control is returned to the problem program, the return code in register 15 is as follows:

| Code (Hexadecimal) | |
|---|---|
| 00 | Successful update |
| 04 | Name already in directory |
| 08 | Name not found |
| 0C | Directory or file full |
| 10 | Permanent input or output error detected attempting to update the directory. |

CALLS TO OTHER ROUTINES: FIND, NOTE, SVCFREE, WRBUFF

CALLED BY: OS STOW macro

* • The STOW routine is part of SOSVCTR

- Files with a filetype of MACLIB must be altered to another filetype before they can be updated. Two files with the same filetype cannot be updated at the same time.

OPERATION:

- If the DELETE option is specified, FIND is called to search the directory for a match to the member in the users list. If the search is successful, the directory entry is zeroed out, a one is put in the change byte, and control is returned to the user with a zero in register 15. If the search is not successful, control is returned to the user with an eight in register 15.

- If the CHANGE option is specified, FIND is called to search the directory for a match to the member in the users list. If the search is not successful, control is returned to the user with an eight in register 15. If the search is successful, FIND is called again to search for the new member name in the directory. If this second search is successful control is returned to the user with a four in register 15. If this second search is not successful, the directory is changed, a one is put in the change byte, and control is returned to the user with a zero in register 15.

- If the REPLACE or ADD option is specified, FIND is called to search the directory for a match to the member in the user list. If a match is found and ADD is specified, control is returned to the user with a 4 in register 15.

If a match is not found, FIND is called to search the directory for a member name of all zeroes. After the search is complete, an end-of-data-set mark (hex 61FFFF61) is written at the end of the member, NOTE is called, and a check is made to make sure there is room for the new member, and, if necessary, a new PDS block on the disk. If there is not enough room, control is returned to the user with a twelve in register 15. If there is enough room and an unsuccessful search for a name of zeroes, SVCFREE is called to obtain enough core for a PDS block and four (4) extra bytes. The PDS block size is then added to the CORESIZE, the item number of the item after the end-of-data-set mark is stored in DICTPTR, and the new PDS block is zeroed out. After a match is found or a new PDS block is added, the directory entry or new PDS block is updated, the pointer to any new PDS block is stored after the last member searched, a two is stored in the change byte and control is returned to the user with a zero in register 15.

- The updated directory is not written to disk until the data set is closed. If an update program does not close a PDS data set for some reason, e.g., a system crash or a re-IPL, the PDS directory for that file will be saved in a temporary file with the same filetype and a filename of $PDSTEMP. To restore the directory to the original file the update program must be run again.

*PDSSAVE (BPAM)

FUNCTION: To ensure that a BPAM PDS directory is not destroyed during an update and is saved after it.

EXIT CONDITIONS: Control is returned to the calling routine with the following code:

| Successful | Calling Routine | FCBPDS entry |
|---|---|---|
| Yes | SOBSAM | address of directory |
| No | SOBSAM | zero |
| Yes | SOOPCL | zero |
| No | SOOPCL | address of directory |

CALLS TO OTHER ROUTINES: FIND, SVCFRET, WRBUF, ERASE

CALLED BY: SOBSAM, SOOPCL, BLDL

* The PDSSAVE routine is part of SOSVCTR

OPERATION: PDSSAVE obtains control from SOBSAM on the first write to a BPAM file after OPEN, and from SOOPCL when an updated BPAM file is closed. When called by SOBSAM, PDSSAVE calls FIND to read the directory. The change byte is checked and, if it is on, control is returned to SOBSAM. If the change byte is not on, a $ is written in the temporary indicator of the directory header record of the original file, FIND is called to read the directory, and a new file is created with the same filetype and a filename of $PDSTEMP. A directory header record and a copy of the in-core directory is written into this file and control is returned to SOBSAM.

When called by SOOPCL or BLDL, PDSSAVE checks the change byte and, if it is zero, frees the directory core, sets FCBPDS to zero, and returns to the caller. If the change byte is not zero, PDSSAVE writes the directory to disk. If there are no errors, the directory header record is written, SVCFRET is called to free the directory core, FCBPDS is set to zero, the $PDSTEMP file is erased, and control is returned to the caller. If there are errors writing the directory to disk, the directory header record is not written and the $PDSTEMP file is not erased.

TABLE/RECORD FORMAT: The format of the directory header record, the directory on disk and the in-core directory with its control words is described below.

| Directory Bytes | Header Record Contents |
|---|---|
| 1 - 6 | Used for MACLIB |
| 7 - 8 | Item pointer to start of directory |
| 11 - 12 | Byte size of directory |
| 13 - 80 | Rest of record not used |

Directory on Disk

| 8 Bytes | 2 Bytes | 2 Bytes |
|---|---|---|
| Name of first member | Item PTR | no. of items |
| Name of second member | Item PTR | no. of items |
| | | |
| Name of nth member | Item PTR | no. of items |

In-Core Directory and Control Words

| | | | |
|---|---|---|---|
| DIRNAME | DS | 3H | Used for MACLIB indicator |
| DIR PTR | DS | 1H | Item pointer to start of directory |
| TEMPEYTE | DS | 1H | TEMP indicator |
| CORESIZE | DS | 1H | Byte size of original in-core directory |
| PDSBLKSI | DS | 1H | Byte size of each PDS block |
| CHNG BYTE | DC | X'00' | Byte used to indicate directory change |
| R15CODE | DC | X'00' | Used to save register fifteen. |
| PDSDIR | DS | 0F | In-core directory<br>At the end of the in-core directory is a full word that is either zero or a pointer to the next PDS block. |

PDS Block
(added to in-core directory by STOW)

| Bytes | Contents |
|---|---|
| 1 to n | block of PDS entries |
| n + 1 to n + 4 | Zero or pointer to next PDS block |

n = number of entries in a block

SOEOB

FUNCTION: Perform actual device I/O.

CALLS TO OTHER ROUTINES: SYSCTL, FINIS, FREE, FRET, RDBUF, WRBUF, STATE

CALLED BY: SOQSAM, SOBSAM

OPERATION:

EOBROUTN: If the BATCH monitor is running, control is passed to BATCHOP for specific data sets: standard processor input and output files; e.g., SYSIN, FORTRAN, PLI, LISTING, TEXT. Otherwise, if FCBPROC contains the address of a user-provided processing routine, control is passed to that routine. If not, control passes to EOB2.

On returning from the user-provided routine, R1 is loaded with the number of bytes actually read or written, R14, with ECB code x'7F', R15 is cleared, and a branch taken to EOBRETRN if the I/O is completed without error. If there is an error, R1 is cleared, R14 is loaded with FCB code x'4F', R15, with the CMS error code, and a branch taken to EOBRETRN.

EOB2: If either I/O is to be performed or there was no address in OSVECTOR, the FCBDEV is obtained and control is passed to the appropriate device dependent code. When device dependent processing is completed, return is via EOBRETRN, URERROR for unit record errors, or DSKERR for disk errors.

BATCHOP: If the operation is a GET, SYSTCTL is called with a read request. If the operation is a PUT and the dsname indicates LISTING, SYSCTL is called with a write request. Any other PUT goes to EOB2 for routing to standard device dependent code.

EOBRETRN: The residual count, if any, is stored in IOBCSW + 6; the contents of R14 are stored in the ECH completion code field (IOBFCBCC); the FCB completion code and CMS error code, if any, are placed in the ECB, and return is to the caller.

URERROR: R14 is loaded with '42' and control is passed to EOBRETRN.

CRT: The ECB completion code x'42' is put in R14, and a x'FF' is inserted in R15.

DUMMY: The ECB completion code x'7F' is put in R14, and R15 is cleared.

CONSOLE: If the desired operation is a READ, a console read is issued and DCBBLKSI is accessed to obtain the length of the desired record. The residual count is calculated and the record moved from the console input buffer to the IOAREA. If FORTRAN execution is in process, R1 is cleared to ensure a zero residual count. In all cases, x'7F' is put in R14, and R15 is cleared.

On output the contents of IOAREA are placed in the console buffer and the record is written.

DISK: A call to either WRBUF (a PUT request) or RDBUF (a GET request) is issued. For a READ request, the residual count is placed in FILEREAD. In either case a x'7F' is placed in R14 and R15 is cleared.

READR: CARDRD is called with the address of the IOAREA in the plist. For a successful return, R14 is loaded with a '7F' before returning to EOBRETRN.
If the error return is end-of-file, a x'7F' is put in R14 and a X'C' is inserted in R15 prior to going to EOBRETRN.

All other errors go to URERROR.

PUNCH: CARDPH is called with the address of the IOAREA in the plist.

PRINT: Requesting blocked records is an error and control goes to URERROR. . If the format is variable, the LRECL is adjusted to eliminate the block descriptor word and the record description word. Truncation length is 133 bytes. PRINTIO is called with the address of IOAREA.

CMSCB

This macro contains the fields from the following OS control blocks that CMS utilizes:

JFCB, DEB, IOB, DECB

```
         MACRO
         CMSCB
*
*    FCB HEADER CONTROL WORDS
*
FCBHEAD   DSECT
FCBFIRST  DC      A(0)                A (FIRST FCB IN CHAIN)
FCBNUM    DC      H'0'                NUMBER OF FCB BLOCKS CHAIN
          DC      H'0'                - NOT USED -
*
*    SIMULATED OS CONTROL BLOCKS
*
FCBSECT   DSECT
FCBINIT   DS      0X                  X'08' = OPEN ACQUIRED THIS CMSCB
FCBNEXT   DS      A                   AL3 (NEXT CMSCB)
FCBPROC   DS      A                   A (SPECIAL PROCESSING ROUTINE)
FCBDD     DS      CL8                   DATA DEFINITION NAME
FCBOP     DS      CL8                 CMS OPERATION
IHAJFCB   DS      0D                  *** JOB FILE CONTROL BLOCK ***
JFCBDSNM  DS      0X                  44 BYTES, DATA SET NAME
FCBTAPID  DS      0X                  TAPE IDENTIFICATION
FCBDSNAM  DS      CL8                   DATA SET NAME
FCBDSTYP  DS      CL8                   DATA SET TYPE
FCBPRPU   EQU     FCBDSTYP + 4          PRINTER/PUNCH COMMAND LIST
FCBDSMD   DS      CL2                   DATA SET MODE
```

```
FCBITEM    DS     H             ITEM IDENTIFICATION NUMBER
FCBBUFF    DS     F             A (INPUT-OUTPUT BUFFER)
FCBBYTE    DS     F             DATA COUNT
FCBFORM    DS     CL2           FILE FORMAT: FIXED/VARIABLE
                                  RECORDS
FCBCOUT    DS     H             RECORDS PER CMS PHYSICAL
                                  BLOCK
FCBREAD    DS     F             N'BYTES ACTUALLY READ
FCBDEV     DS     X             DEVICE TYPE CODE
FCBDUM     EQU    0               DUMMY DEVICE
FCBPTR     EQU    4               PRINTER
FCBRDR     EQU    8               READER
FCBCON     EQU    12              CONSOLE TERMINAL
FCBTAP     EQU    16              TAPE
FCBDSK     EQU    20              DISK
FCBPCH     EQU    24              PUNCH
FCBCRT     EQU    28              CRT
FCBMODE    DS     X             MODE:  1, 2, 3, 4, 5
FCBXTENT   DS     H             NUMBER OF ITEMS IN EXTENT
           DS     F             - NOT USED -
           DS     F             - NOT USED -
           DS     F             - NOT USED -
           DS     F             - NOT USED -
           DS     F             - NOT USED -
FCBR13     DS     F             SAVEAREA VECTOR R13
FCBKEYS    DS     A             A (DDS IN'CORE KEY TABLE)
FCBPDS     DS     A             A (PDS IN-CORE DIRECTORY)
JFCBMASK   DS     8X            VARIOUS MASK BITS
JFCBCRDT   DS     3C            DATA SET CREATION DATE (YDD)
JFCBXPDT   DS     3C            DATA SET EXPIRATION DATE (YDD)
JFCBIND1   DS     X             INDICATOR ONE
JFCBIND2   DS     X             INDICATOR TWO
JFCBUFNO   DS     X             NUMBER OF BUFFERS
JFCBFTEK   DS     0X            BUFFERING TECHNIQUE
JFCBFALN   DS     X             BUFFER ALIGNMENT
JFCBUFL    DS     H             BUFFER LENGTH
JFCEROPT   DS     X             ERROR OPTION
JFCKEYLE   DS     X             KEYLENGTH
           DS     X             - NOT USED -
JFCLIMCT   DS     3X            BDAM SEARCH LIMIT
FCBDSORG   DS     0X            DATA SET ORGANIZATION
JFCDSORG   DS     2X
FCBRECFM   DS     0X            RECORD FORMAT
JFCRECFM   DS     X
JFCOPTCD   DS     X             OPTION CODES
FCBBLKSZ   DS     0H            BLOCK SIZE
JFCBLKSI   DS     H
FCBLRECL   DS     0H            LOGICAL RECORD LENGTH
JFCLRECL   DS     H
```

```
FCBICSW    DS      X                  I/O OPERATION INDICATOR
FCBICRD    EQU     X'01'              READ/SOQSAM
FCBICWR    EQU     X'02'              WRITE/PUT
FCBCLOSE   EQU     X'80'              DURING "CLOSE"
FCBCLEAV   EQU     X'40'              DISP=LEAVE DURING CLOSE
FCBPVMB    EQU     X'04'              PUT-MOVE-VAR-BLK
FCBCASE    EQU     X'08'              ON=LOWER CASE CONSOLE I/O
           DS      1X                 - NOT USED -
DEBLNGTH   DS      0X                 L'DEB IN DBLW WORDS
           DS      F                  - NOT USED -
IHADEB     DS      0D                 *** DATA EXTENT BLOCK ***
DEBTCBAD   DS      A                  A (MOVE-MODE USER BUFFER)
           DS      F                  - NOT USED -
DEBOFLGS   DS      4X                 DATA SET STAUS FLAGS
DEBOPATB   DS      4X                 OPEN/CLOSE OPTION BYTE
IOBICFLG   DS      0X                 (START OF IOB PREFIX FOR
                                        NORMAL SCH)
IOBOUT     EQU     X'40'              "WRITE, PUT" IN PROGRESS
IOBIN      EQU     X'20'              "READ, GET" IN PROGRESS
IOBNXTAD   DS      A                  A (NEXT BUFFER TO BE USED)
IOBECB     DS      F                  ECB FOR QSAM NORMAL SCHEDUL-
                                        ING
IHAIOB     DS      0F                 *** INPUT/OUTPUT BLOCK ***
DEBDEBID   DS      0X                 DEB IDENTIFICATION
DEBDCBAD   DS      A                  A (DATA CONTROL BLOCK)
IOBECBCC   DS      0X                 ECB COMPLETION CODE
IOBECBPT   DS      A                  A (EVENT CONTROL BLOCK)
IOBFLAG3   DS      0X                 I/O ERROR FLAG
IOBCSW     DS      8X                 LAST CCW STORED (I.E., RESIDUAL
                                        COUNT)
IOBSTART   DS      A                  X'ID-NEXT BUFFER', AL3 (INITIAL
                                        BUFFER)
IOBDCBPT   DS      A                  A (DATA CONTROL BLOCK)
IOBEND     DS      0X                 END-OF-INPUT/OUTPUT BLOCK
FCBEND     DS      0D                 END-OF FCB, JFCB, DEB, IOB
                                        BLOCKS
FCBENSIZ   EQU     (*-FCBSECT)/8      SIZE OF FCB ENTRY, DOUBLE-
                                        WORDS
           SPACE   3
*
*    DATA EVENT CONTROL BLOCK
*
IHADECB    DSECT
DECSDECB   DS      F                  EVENT CONTROL BLOCK
DECTYPE    DS      H                  TYPE OF I/O REQUEST
DECBRD     EQU     X'80'              READ SF
DECBWR     EQU     X'20'              WRITE SF
DECLNGTH   DS      H                  LENGTH OF KEY & DATA
DECDCBAD   DS      A                  V (DATA CONTROL BLOCK)
```

```
DECAREA     DS      A                     V (KEY & DATA, BUFFER)
DECIOBPT    DS      A                     V (IOB)
*                   BDAM EXTENSION
DECKYADR    DS      A                     V (KEY)
DECRECPT    DS      A                     V (BLOCK REFERENCE FIELD)
            SPACE   3
*
*   SOME FREQUENTLY USED EQUATES
*
DDNAM       EQU     FCBDSTYP              FILETYPE = DATA SET NAME
BLK         EQU     X'10'                 RECFM=BLOCKED RECORDS
BS          EQU     X'20'                 MACRF=BSAM
DA          EQU     X'20'                 DSORG=DIRECT ACCESS
FXD         EQU     X'80'                 RECFM=FIXED LENGTH RECORDS
IS          EQU     X'80'                 DSORG=INDEXED SEQUENTIAL
LOC         EQU     X'08'                 MACRF=LOCATE MODE
MOV         EQU     X'10'                 MACRF=MOVE MODE
PS          EQU     X'40'                 DSORG=PHYSICAL SEQUENTIAL
PO          EQU     X'02'                 DSORG=PARTIONED ORGANIZATION
PREVIOUS    EQU     X'80'                 OFLGS=PREVIOUS I/O OPERATION
QS          EQU     X'40'                 MACRF=QSAM
UND         EQU     X'C0'                 RECFM=UNDEFIN FORMAT RECORDS
VAR         EQU     X'40'                 RECFM=VARIABLE LENGTH
                                            RECORDS
            MEND
```

## SECTION 3: MONITOR OPERATIONS

The monitor is responsible for the following operations:

- System Initialization
- System Continuity
- Interruption Handling
- Override Handling
- System Restart

### SYSTEM INITIALIZATION

After initial program load (IPL), control passes to the CMS initialization program (INIT), which immediately establishes addressability and sets up the interrupt new PSW's. INIT then calls INITSUB, which performs one time only initialization. INITSUB is covered up once intialization is completed. If the IPL was from the card reader, INITSUB calls CLOSIO PRINTER (via BALR) to ensure printing of the load map generated by the nucleus loader. INITSUB then initializes the interval timer in hex location 50 and calls the IPLDISK function program to write an IPL program and a copy of the CMS nucleus on the system disk. Subsequently, an IPL may be from the system disk (usually 190). On return from IPLDISK, INIT proceeds as if the IPL had been from disk, as described below. (IPL by hexadecimal disk address enters INITSUB at this point.)

INITSUB tests for IPL on a bare machine and calls BAREMACH to redefine device addresses, if this test is positive. INITSYS is called to generate the SSTAT table. INITSUB then branches to hexadecimal location F0 so that a saved version of CMS can be made for IPL by name. (IPL by name (IPL CMS) starts at location F0.) The instructions at location F0 return control to INITSUB, which then proceeds to complete the one time only initialization as follows:

1. Set up all new PSW's.

2. Set BLIP timer interrupt.

3. Clears IPLDEV if IPL by name was the mode of entry into INITSUB. This enables the IPL module to work correctly.

4. Calculate core size and store in NUCON and other places. Calculate and store core size related parameters.

5. Pass control to BATCH if present within the nucleus.

6. Otherwise, return to INIT.

A message that includes the version number and date of the CMS currently in use is then typed, indicating that CMS has been initialized. After that INIT issues a read to the terminal for the user's first command and waits for the command to be entered. If the first command is a carriage return, INIT calls LOGIN UFD to read the P-Disk user file directory into core, tests for the existence of the file PROFILE EXEC (and if the file exists calls EXEC PROFILE), and enters the main control loop (the CMS command environment) at the point where a command has been successfully completed.

If the first user command is not a carriage return, INIT calls the SCAN function program to convert the input line into the standard CMS parameter-list format and checks the first command. If it is FORMAT P or FORMAT P ALL (but not FORMAT P C), INIT calls the FORMAT command via SVC X'CA' and checks for a successful return therefrom. If the FORMAT command was successful, INIT enters the main control loop as described in the above paragraph. If the FORMAT command failed, the read is reissued, and the typed-in command is again analyzed. If the command was LOGIN, INIT calls the LOGIN command via SVC X'CA' and checks for a successful return from the command. If the LOGIN failed, the read is reissued and the typed-in command is again analyzed. If LOGIN NOPROF was not specified, INIT tests for the existence of the file PROFILE EXEC, calls EXEC PROFILE via SVC X'CA' if the file exists, and then enters the main control loop. If LOGIN NOPROF was specified, INIT enters the main control loop directly. If the first command is none of the above, an implied automatic login procedure is invoked via an SVC call to LOGIN UFD (without disturbing the first entered command); upon successful completion of this LOGIN, PROFILE EXEC is tested for and executed if it exists, and the main control loop is then entered to execute the first command entered by the user.

Having thus initialized CMS, handled the first user command, and ensured that the user is properly logged in, INIT has finished the initialization phase and subsequently handles typed-in commands as described under "SYSTEM CONTINUITY."

## SYSTEM CONTINUITY

INIT is responsible for the continuity of operation of the CMS command environment. When a typed-in command has been executed and SVCINT returns to INIT, it passes along the return code from the called command in register 15. A code of zero indicates successful completion of the command; a positive code indicates that the command was completed but with an apparent error; and a negative code returned by SVCINT indicates that the typed-in command could not be found or executed at all.

Upon return from SVCINT, INIT saves this return code briefly and calls the UPUFD function program to update the user file directory (UFD) on the user's P-disk. (If the user had typed in "ko" while the previous command was running, INIT calls CLROVER at this point to ensure completion of the override trace printing.)

Having updated the user file directory, INIT checks the return code that had been passed back by SVCINT. If the code is zero, INIT types a READY message and the CPU time used by the given command. If the code is positive, an error message is typed, including the error code returned (as a five-digit decimal number), along with the CPU time used. If the code is negative, INIT types the message "INVALID CMS COMMAND". INIT then proceeds in the main control loop to call WAITRD to get the next command. When the command is entered, INIT calls SETCLK to initialize the CPU time for the new command and then puts it in standard parameter-list form by calling the SCAN function program. After calling SCAN, INIT checks to see if an exec filetype exists with a filename of the typed-in command. (For example, if ABC was typed in, INIT checks to see if ABC EXEC exists.) If such an EXEC file does exist, INIT adjusts register 1 to point to the same command as set up by SCAN, but preceded by CL8'EXEC', and then issues an SVC X'CA' to call the corresponding EXEC procedure ('ABC EXEC' in the example).

If no such EXEC file exists for the first word typed in, INIT makes one further check using the CMS 'ABBREV' abbreviation-checker. If, for example, the first word typed in had been 'FORT', INIT looks up FORT via the ABBREV routine (if included in the nucleus); if an equivalent is found (for example, 'FORTRAN' for 'FORT'), INIT looks for an EXEC file with the name of the equivalent word (for example, FORTRAN EXEC); if such a file is found, INIT adjusts R1 as described above to call EXEC and substitutes the equivalent word (for example, FORTRAN) for the first word typed in (for example, FORT). Thus if FORT is a valid abbreviation for FORTRAN and the user has an EXEC file called 'FORTRAN EXEC', he invokes this when he merely types in 'FORT' from the terminal.

If no EXEC file is found either for the entered command name or for any equivalent found by ABBREV, INIT leaves the terminal command as processed by SCAN and then issues an SVC X'CA' to pass control to SVCINT, which, in turn, passes control to the appropriate command program. When the command terminates execution, or if SVCINT cannot execute it, the return code is passed in register 15, and the CMS command environment continues as described earlier.

## Stacking of Typed-in Commands

While a CMS command is being executed, it is possible for a user to type in or stack the next command (or commands) which he would like executed. To do this he hits the attention key once if running on an actual 360, or twice if running under the Control Program. This action generates an attention interrupt, which is processed by the CONSI console interrupt routine and the WAITRD function program. The CONSI routine issues a read from the terminal into an area of free storage; when the desired command has been typed in, it is placed in a chained list of finished read commands. Then when WAITRD asks for a line to be typed in, the previously typed-in finished input commands are supplied on a first-in, first-out basis.

For example, while an assembly of PROGRAM is being run, a user might stack his next two commands, which might be OFFLINE PRINT PROGRAM LISTING and OFFLINE PUNCH PROGRAM TEXT. Stacked input commands can be abbreviated the same as other typed-in commands.

When stacked-up input commands are processed by INIT, several READY or error messages are typed in a row before a new read is issued. In the example above, three consecutive READY messages would be given, one from the assembly, one from the OFFLINE PRINT, and one from the OFFLINE PUNCH.

CMS commands (or input data to EDIT) can also be stacked by entering several logical lines on one physical line, separated by the CMS line-end character, which is usually a pound-sign (#).

In the above example, the three commands could be typed in as follows:

```
assemble program#offline print program listing#offline punch program text
```

## Abbreviations for CMS Commands

As mentioned elsewhere, INIT and SVCINT sometimes wish to check if an abbreviation (or synonym) has been substituted for a CMS (or user) command. To implement this feature, the CMS nucleus normally includes the ABBREV abbreviation-checker function program. Use of ABBREV facilitates certain abbreviations for CMS commands. For example, A suffices for ASSEMBLE, E for EDIT, F for FORTRAN, O for OFFLINE, L for LISTF, and the like.

The abbreviations are interpreted based on the number of characters contained in the first word of a command. ABBREV looks this up in a user defined synonym table (if one has been set up in core by calling the CMS 'SYN' command), or in a table of standard cms system abbreviations. If a match is found, and a count in the table indicates that sufficient characters were in the abbreviation to identify it without ambiguity (for example, 'ALTER' requires at least two characters to distinguish it from 'A' for 'ASSEMBLE'), then ABBREV returns the equivalent match, which is then substituted for the given abbreviation by SVCINT or INIT. For example, EDIT is substituted for either E, ED, or EDI; ALTER for AL, ALT, or ALTE.

Note: Abbreviations as described herein are valid for the first word of commands typed in from the terminal under the CMS command environment (including stacked commands), or for parameter lists in existing CMS programs, or for commands handled by the CMS EXEC command. They are not valid, however, for: (1) debug requests, (2) edit requests, or (3) RETURN from the ECHO command. Note that the EDIT and DEBUG requests have their own abbreviation schemes.

## Minimum Abbreviations for CMS System Commands

The complete list of minimum abbreviations for CMS System commands is given in the following table.

| System Command | Shortest Form | System Command | Shortest Form |
|---|---|---|---|
| ASSEMBLE | A | LISTF | L |
| ALTER | AL | OFFLINE | O |
| CLOSIO | CL | PRINTF | P |
| CPFUNCTN | CP | SCRIPT | SC |
| DEBUG | DE | STAT | S |
| EDIT | E | TAPE | T |
| FORTRAN | F | UPDATE | U |
| GENMOD | G | VSET | V |

## INTERRUPTION HANDLING

The monitor processes all SVC, input/output, program, machine, and external interruptions. The following paragraphs describe the processing carried out for each type of interruption.

### SVC Interruptions

Supervisor call (SVC) interruptions are handled by the SVCINT monitor program. Two types of SVC's are processed by SVCINT: internal linkage SVC's (refer to "Internal Linkage Scheme") and certain Operating System/360 SVC's (refer to "SVC Simulation"). Internal linkage SVC's are issued by the monitor, command, and function programs of the system when they require the services of other CMS programs. (Commands entered by the user from the terminal are converted to internal linkage SVC's by INIT.) The Operating System/360 SVC's are issued by the processing programs (for example, Assembler, FORTRAN compiler). The following paragraphs describe how these interrupts are handled by SVCINT.

#### Internal Linkage SVC's

When SVCINT receives control as a result of an internal linkage SVC (that is, an SVC X'CA'), it saves the contents of the general purpose and floating-point registers and the SVC old PSW in the normal save area (NRMSAV) and establishes the normal and error return addresses and stores them along with the name of the called program in the normal save area. It then determines if the called program is in the transient area (TRANSAR) by comparing the first 8 bytes of the parameter list with TRANSRT. TRANSRT is a 12-byte area with the following content:

(1) First 8 bytes (TRANSRT): filename of routine or program currently in TRANSAR.

(2) Next byte (TRANMSK): desired system-mask for routine or program currently in TRANSAR.

(3) Next 3 bytes: DC AL3(TRANSAR) = address of transient area.

If the name of the called program matches TRANSRT, the called program is already in the transient area. If so, SVCINT stacks the register contents and return addresses as the last entry in a last-in, first-out list, and then branches to the called program in TRANSAR.

If the called program does not happen to be in TRANSAR (its name does not match TRANSRT), then SVCINT scans through the function table FUNCTAB. FUNCTAB contains the name, desired system mask, and core address of all programs which are in (or <u>can</u> be included in) the CMS nucleus. Each entry in FUNCTAB is 12 bytes long, in a similar format to that shown above for TRANSRT. If the called program is found in FUNCTAB, and the core address is valid (that is, nonzero), then SVCINT stacks the register contents and return address as above, sets the system mask if necessary, and branches to the called program at its given address.

If the called program is in FUNCTAB but its core address is zero (indicating that the program is actually disk-resident), or if the called program is not found in FUNCTAB at all, then SVCINT stacks the register contents and return addresses as above and attempts to call in a module of this name via the LOADMOD command. If the LOADMOD succeeded, then SVCINT sets the system mask appropriately and transfers control to the called program. (The core address is given by the constant STADDR in the NUCON table, having been placed there by LOADMOD.) If LOADMOD brought the called program into TRANSAR, it stored its name in TRANSRT. SVCINT, in turn, recognizes that the called program is now in TRANSAR, and branches to it there (ignoring STADDR); SVCINT, of course, is now cognizant of the fact that a new program resides in TRANSAR.

(Note: the loading of a transient disk resident program via LOADMOD is completely transparent to the normal LOAD and START commands. For example, a program can be LOADed, then perhaps OFFLINE PRINT or PRINTF called to print the load map, ERASE called to remove the LOAD map, etc., and then the program STARTed. NUCON and the loader tables are unaffected by the calls to the transient disk resident routines OFFLINE or PRINTF.)

If the program was not found in either TRANSRT or FUNCTAB, nor successfully LOADMODed, SVCINT makes one further effort to link to the command. The ABBREV abbreviation checker (if included in the CMS nucleus) is called to determine whether an equivalent match (for example, EDIT) is found for the input command (for example, E, ED, or EDI). If an equivalent is found, it is substituted (not in core, but in registers) for the original command, and the threefold search (TRANSRT, FUNCTAB, and LOADMOD) is initiated once more. If successful this time, SVCINT links to the command in the usual way.

If (1) the abbreviation-checker is not included in the nucleus (a permissible installation option), (2) if no equivalent was found by ABBREV, or (3) if the second threefold search for the equivalent command fails, SVCINT returns a negative error code to the caller. (If the caller was INIT, the message "?CMS:xxxxxxxx" is typed.)

When the called program returns, SVCINT determines whether or not there were any errors encountered during the execution of that program. (A code returned by the called program in register 15 will indicate this.) If there were no errors, SVCINT saves the contents of the general purpose and floating point registers in the normal save area as they exist upon return from the called program. If normal overriding is not in effect (via SETOVER command), SVCINT restores the calling program's registers, deletes the last entry from the last-in, first-out stack, and makes the appropriate normal (errorless) return to the calling program.

If there were no errors and normal overriding is in effect, SVCINT moves the data required by the normal override handling program (HNDLNRM) into the normal override save area NOVSAV, saves the addresses contained in the normal and error override switches (NRMOVR and ERROVR), and sets these switches to zero so that overriding will not occur during execution of the normal override program. It then deletes the last entry in the last-in, first-out stack and passes control to the normal override handling program. This program, after completing its processing, will return to SVCINT, which will, in turn, return to the calling program.

If errors are encountered during the execution of the called program, and error overriding is not in effect (via either SETOVER or SETERR commands), SVCINT compiles data for use by the error return program (either STDERR or the portion of code pointed to by the address constant following the SVC X'CA') in the error save area (ERRSAV), restores the calling program's registers, deletes the last entry in the last-in, first-out stack, and masks the appropriate error return as defined by the calling program. (If the calling program provides an address constant following the SVC X'CA', the portion of code pointed to by that address constant can gain access to the data in ERRSAV through the use of the .RDERR function program.)

If there were errors and error overriding is in effect, SVCINT moves the data required by the error override handling program (HNDLERR) into the error override save area (ERRSAV), saves the addresses contained in the normal and error override switches and sets these switches to zero so that overriding will not occur during execution of the error override program. It then deletes the last entry in the last-in, first-out stack and passes control to the error override handling program. This program, after completing its processing, will return to SVCINT, which will, in turn, make the appropriate error return as defined by the calling program.

Other System/360 SVC's

The general approach taken by SVCINT to process other SVC's supported under CMS is essentially the same as that taken for an internal linkage SVC. (Refer to "SVC Simulations" for a list of the Operating System/360 SVC's supported by CMS.) However, rather than passing control to a command or function program, as is the case with an internal linkage SVC, SVCINT passes control to the appropriate routine. If overriding is not specified, SVCINT will return directly to the program. If overriding is specified, the override program will return to SVCINT, which will return to the program.

In linking to the particular SVC routine, however, SVCINT uses a different procedure than the threefold (TRANSRT, FUNCTAB, and LOADMOD) search used for SVC X'CA' calls.

In handling these other calls, SVCINT uses two tables, a user-defined SVC table (if any - set up by the HNDSVC program), and the table of standard system OS calls supported (to whatever extent feasible) by CMS.

Each of these tables is in the form of several four-byte items, each of which is of the following format:

First byte: SVC number (for example, 19).

Next 3 bytes: Core-address of appropriate routine for that OS-call (for example, OPEN routine to handle SVC 19 calls).

If the user-defined SVC table is present, any SVC number (other than X'CA') is looked for in that table (checking the number against the first byte). If it is found, control is transferred to the routine at that address.

If the SVC number is not found in the user-defined SVC table (or if the table is nonexistent), it is then looked for in a similar fashion in the standard system table of OS calls.

If it is found there, and the routine address is nonzero, control is transferred to that address in the usual way.

If it is found there, but the routine address is zero, this signals SVCINT that the particular routine is included in a transient disk resident module named 'SVCCARE', which handles various OS calls that need not always reside in the CMS nucleus. If SVCCARE happens to be in the transient area (TRANSAR), control is passed to SVCCARE forthwith. If not, SVCCARE is LOADMODed as any other transient disk resident routine, and then receives control.

In any event, SVCCARE then distinguishes which particular OS call has been passed to it, and handles it accordingly.

If the SVC number is not in either table, then it is treated like an ABEND call.

For setting up the user-defined SVC table mentioned above, the HNDSVC initialization program is provided, making it possible for a user to provide his own SVC routines. This function program is described on the following pages.

When SVCINT passes control to any SVC routine, the following conditions exist:

Registers

| | |
|---|---|
| 0-11 | and 15 as they were at SVC time. |
| 12 | address of SVC-handler routine. |
| 13 | address of SVC save area. |
| 14 | return address to SVCINT. |

The SVC save area has the following format:

| Bytes | Contents |
|---|---|
| 0- 63 | caller's registers 0-15 |
| 64- 71 | SVC-old-PSW |
| 72- 95 | floating-point registers 2, 4, 6* |
| 96-175 | 80 bytes for use by SVC-simulation routine |

*FPR 0 is saved by CMS elsewhere.

## HNDSVC

Filename - HNDSVC, module - HNDSVC, Disk Resident, Transient.

### Function

The HNDSVC function initializes the SVC Interrupt Handler to transfer control to a given location for a specific SVC number (other than X'CA' or 202), or to clear such previous handling.

### Calling Sequence

```
         DS    0F

PLIST    DC    CL8'HNDSVC'                     called routine

         DC    CL4'SET' or CL4'CLR'            function

         DC    AL1 (SVC number), AL3 (address) argument(s)

         .     .

         .     .

         .     .

         DC    X'FFFFFFFF'                     end of list
                                               (must be present)
```

<u>Notes:</u>

1. For CLR, the address fields are irrelevant.
2. Individual SVC numbers may be added or cleared before termination of the command.

<u>Error Codes</u>

| | |
|---|---|
| E(00001) | INCORRECT HNDSVC PARAMETER LIST |
| E(00002) | SVC number replaces another of the same number (for HNDSVC SET). |
| E(00003) | SVC number clearing one which was not set (for HNDSVC CLR). |

Operation

HNDSVC processes the parameter list checking for possible errors. An SVC number of 202 (x'CA') is illegal, and two SVC, numbers that are the same are also invalid. The function must, of course, be either SET or CLR.

For the HNDSVC SET call, a check of addresses is also made for reasonableness (each must be an even number greater than 0 and within the user's core size). An error code 1 (with a message) is given if the parameter list is incorrect.

If the parameter list appears valid, for HNDSVC SET, the logic is as follows:

1. If no user-defined SVC table exists, pointers are set up to the part of the caller's parameter list (from his first argument through the last), that forms the user-defined SVC table until it is cleared.

2. If a user-defined table already exists, enough free storage is obtained for both tables, the old one and the new one. The two are then merged into the free storage area, and pointers to the new table are set up for SVCINT. If the new table contains any SVC numbers identical to the old, the new ones replace the old, and an error code 2 is subsequently returned to advise the caller of this situation. This is purposely not treated as an ABEND condition so that a user could start off with one table, then add a new one, replacing only those in the old table that he wishes to supersede. (The error code is given, however, so that the caller is aware of the replacement, in case it was accidental.)

For the HNDSVC CLR call, the parameter list is checked against the table (if any) currently in use. If the numbers match the table, the table is returned via FRET to free storage (if necessary), and the pointers for SVCINT are cleared. If the numbers do not match the table perfectly, those that do match are cleared, the table compacted appropriately leaving only those which have not been cleared, and the pointers for SVCINT revised accordingly. If any number tries to clear an SVC number not in the table, an error code 3 is given subsequently, but processing continues. If the table does not exist at all, an error code 3 is also given.

In any case, after the HNDSVC CLR call, the pointers will be clear if there is no table left, or revised accordingly if there are still some calls left.

In actual practice, the parameter list is set up so that the function can be initialized to CL4'SET' for the set call, and later to CL4'CLR' for the clear call, using the same parameter list.

At the completion of each CMS command, the INIT program clears any remaining user-defined SVC table, in the event that a program loaded and run in user core forgot to issue a HNDSVC CLR call when finished.

If HNDSVC is inadvertently called from a terminal, error code 1 (with message) will be given.

## Input/Output Interruptions

All input/output interruptions are received by the I/O interruption handler (IOINT). IOINT saves the I/O old PSW and the channel status word (CSW). It then determines the nature of the device causing the interruption and passes control to the program that processes interruptions from that device. It does this by scanning the entries in the device table (DEVTAB) until it finds the one containing the device address that is the same as that of the interrupting device. (DEVTAB is a block of storage within the nucleus constant area (NUCON). It contains an entry for each device in the CMS system. The entry within DEVTAB for a particular device contains, among other things, the address of the program that processes interruptions from that device.)

When the interrupt handling program corresponding to the interrupting device completes its processing, it returns control to IOINT. At this point, IOINT tests the wait bit in the saved I/O old PSW. If this bit is off, it usually indicates that the interruption was caused by a terminal (asynchronous) I/O operation. In this case, IOINT returns control to the interrupted program by loading the I/O old PSW.

If the wait bit is on, this usually indicates that the interruption was caused by a nonterminal (synchronous) I/O operation and that the program that initiated the operation called the WAIT function program to wait for a particular type of interruption (usually a device end, signaling the completion of an I/O operation). (Refer to "Nonterminal I/O".) In this case, IOINT determines whether or not an interruption from the interrupting device is being waited for. It does this by checking the pseudo-wait bit in the device table entry for the interrupting device. If this bit is off, the system is waiting for some event other than the interruption from the interrupting device; IOINT returns to the wait state by loading the saved I/O old PSW. (This PSW has the wait bit on.)

If the pseudo-wait bit is on, an interruption from a particular device is being waited for. (The WAIT function program sets this bit when called by a program that is waiting for an I/O interruption from a particular device.) In this case, IOINT determines whether or not the interruption was the one being waited for. (The interruption may or may not be the one being waited for; for example, a program may be waiting for a device-end interruption from the device, but a channel-end or error interruption may occur.) The program that processes the interruption from the interrupting device will inform IOINT of this. If the interruption is not the one being waited for, IOINT loads the saved I/O old PSW. This will again place the machine in the wait state. Thus, the program that is waiting for a particular interruption will be kept waiting until that interruption occurs.

If the interruption is the one being waited for, IOINT resets both the pseudo-wait bit in the device table entry and the wait bit in the I/O old PSW. It then loads that PSW. This causes control to be returned to the WAIT function program, which, in turn, returns control to the program that called it to wait for the interruption.

## CMS Terminal Interruptions

Terminal input/output interruptions are handled by the CONSI program. Upon receiving control, CONSI determines the nature of the interruption. If it is a channel end, CONSI returns control to IOINT and indicates that it is awaiting an interruption other than the channel end.

If it is a device end, indicating the completion of an I/O operation, CONSI checks whether the finished operation was a read or a write. If it was a write, CONSI deletes the corresponding CCW package from the read-write stack, and frees the storage occupied by it. If there are no more requests in the read-write stack, CONSI then exits to IOINT. If there are some requests in the stack, CONSI obtains the next, and starts the new I/O operation. If this operation is started successfully, CONSI then returns to IOINT. If the I/O was not started successfully, it is either because of an error or a pending attention. In the former case, CONSI terminates CMS operation. In the latter case, the user wishes to stack a command for later processing or to enter a request (either KT, KX, or KO). To enable the user to enter his input line, CONSI constructs a CCW package for a read. If, at this time, there are no requests in the read-write stack, CONSI immediately makes the CCW package the first and only entry in both the read-write and pending read stacks, starts the read operation, and exits to IOINT. If there are requests in the read-write stack, but there are no pending reads, CONSI makes the CCW package the first and only entry in the pending read stack and also links it into the read-write stack as the first entry. It then starts the read operation and exits to IOINT. If there are requests in the read/write stack and there are entries in the pending read stack, CONSI makes the CCW package the last entry in both stacks, starts the next I/O request in the read/write stack, and exits to IOINT.

If a device end caused the interruption and the operation just finished was a normal read (that is, a read not triggered by an attention), CONSI links the CCW package for that read into the finished read stack as the last entry and deletes it from the read-write stack and the pending read stack. From this point, CONSI proceeds in essentially the same manner as it does for a device end caused by the completion of a write (see previous paragraph).

If a device end caused the interruption and the operation just finished was a read triggered by an attention (for example, KT, KX, KO, or a stacked input command), CONSI determines whether it was a KT, KX, or KO request by examing the input buffer. If it was not, CONSI handles the input command as it does a normal finished read. If the read was KO, CONSI sets the kill-override flag that is referenced by SVCINT in determining where control is to be returned. If the read was KT, CONSI deletes all write requests from the read-write stack and sets the kill-typing flag that is referenced by the terminal write programs (TYPLIN/TYPE). After performing the special processing for KT or KO, CONSI proceeds in essentially the same manner as it does for a device end caused by the completion of a write.

If the read was KX, CONSI transfers control to the routine KILLEX, which terminates CMS execution. KILLEX performs the following operations:

- Calls DESBUF to remove all console I/O requests from the read-write stack.

- Calls LOGDISK to close any open CMS files and to update the user file directory on any active read-write disk(s).

- Calls CLOSIO to close the printer and the card reader/punch (the Control Program will interpret this as a request to close the spooling files for these devices).

- Calls the CMS 'IPL' command to re-IPL a fresh copy of CMS.

If an attention caused the interrupt that gave CONSI control, CONSI enables the user to stack his command as previously described.

## CMS Reader/Punch/Tape Interruptions

Interruptions from these devices are handled by the programs that actually issue the corresponding I/O operations. (Refer to the discussions of CARDIO and TAPEIO.) When an interruption from any of these devices occurs, control passes to IOINT. Then, IOINT returns control to WAIT, which returns control to the program that issued the I/O operation. This program can then analyze the cause of the interruption.

Since the address of the interrupt processor in the device table of these devices is 0, IOINT assumes the interrupt was correct and resumes as if a successful return was obtained from an actual interrupt processor.

## CMS Printer/Disk Interruptions

Interruptions from either of these devices give control to IOINT, through which control passes to the appropriate interrupt processing program. As described under "CMS Non-Terminal I/O", the interrupt processing program determines if this is the interrupt requested and if an error has occurred. The channel programs for the printer and disk end with a NOP channel command to cause channel end (CE) and device end (DE) to occur together, thereby ensuring the associated interrupt processing program that the interrupt being serviced is the requested one. On an error, the old I/O PSW, the Channel Status Word, and an error indicator are moved to the proper device table (DEVTAB).

The interrupt processor then returns control to IOINT, which returns control to WAIT, which in turn returns control to the program that issued the I/O operation. This I/O handling program then checks the error indicator in the corresponding device table. If an error is indicated, the program calls the CMS centralized error recovery program, IOERR; if there is no error, the I/O handling program continues its normal processing.

### User Controlled Device Interruptions

Interrupts from devices under user control are serviced the same as CMS devices except that WAIT and IOINT manipulate a user created device table, and that IOINT passes control to a user written interrupt processing routine.

### Program Interruptions

The program interruption handler (PRGINT) receives control when a program interruption occurs. When it gets control, PRGINT determines if the executing program has specified a program interruption exit via a SPIE macro instruction. If it has not, PRGINT passes control to the DEBUG command program. This allows the user to determine the cause of the interruption through use of DEBUG requests.

If the executing program has specified a program interruption exit via SPIE, PRGINT completes the construction of the program interruption element (PIE) by storing the program old PSW and the contents of registers 14, 15, 0, 1, and 2 into it. (The routine that handles the SPIE macro instruction has already placed the address of the program interruption control area (PICA) into PIE.) PRGINT then determines if the event that caused the interruption was one of those selected by the SPIE macro instruction. If it was not, PRGINT passes control to the DEBUG command program.

If the cause of the interruption was one of those selected in the SPIE macro instruction, PRGINT picks up the exit routine address from the PICA and passes control to the exit routine. Upon return from the exit routine, PRGINT generates a program interruption to obtain the system mask from the current PSW. (The exit routine may have altered the system mask and PRGINT must restore it before returning to the interrupted program.) It then places the obtained system mask into the appropriate field of the program old PSW it saved on entry and loads that PSW to return to the interrupted program.

### External Interruptions

An external interruption causes control to be passed to the external interrupt handler (EXTINT), which passes control to DEBUG if the interrupt was not a timer interrupt. If the interrupt was caused by the timer, EXTINT resets the timer and types the BLIP character at the terminal. The standard BLIP timer setting is two seconds, and the standard BLIP character is upper case, followed by lower case (it moves the typeball without printing).

### Machine-Check Interruptions

When a machine-check interruption occurs, control is passed to a corresponding interruption handler (MCHINT). MCHINT calls the TYPLIN function program to type a message at the terminal to the effect that a machine error has occurred. Next, it calls the WAIT function program to wait until the message has been typed. When the message has been typed, MCHINT passes control to the DEBUG command program, which enables the user to determine the effects of the interruption through the use of the DEBUG program.

Note: If a machine check occurs while running under CP-67, the following message is typed and CP is entered:

```
MACHINE CHECK INTERRUPT
CP ENTERED, REQUEST PLEASE
```

To reflect this machine check to the CMS virtual machine, issue a BEGIN console function, and CMS will handle the interrupt as described above.

## OVERRIDE HANDLING

Override handling deals with the processing carried out when the normal and/or error override facilities have been activated by the SETOVER or SETERR commands. The override handling programs (HNDLNRM and HNDLERR) are part of the OVERRIDE module. This module is in core only when the normal and/or error override facilities have been activated.

### Normal Override Operations

The normal override handling program (HNDLNRM) gets control from SVCINT when normal overriding has been activated by the SETOVER command and when a program that received control as a result of an SVC executes without error (that is, it returns a zero in general purpose register 15). HNDLNRM calls the FREE function program to obtain a block of free storage for use as a work area. It then calls the .RDERR function program to place the normal override data saved in NOVSAV by SVCINT into the work area. This data consists of the address of the calling program, the name of the called program, the SVC old PSW that resulted from the call, the normal return address, the error return address, the contents of the general purpose and floating-point registers at the time of the call, and the contents of the general purpose and floating-point registers upon return to SVCINT from the called program. The normal return address is either the address of the instruction immediately following the SVC, or the address of the instruction immediately after the address constant, if one is provided by the calling program. The error return address is either the address contained in the address constant (if one is provided) or the address of the standard error routine, STDERR.

If the called program is the WAIT function program and the user has specified NOWAIT in the SETOVER command, HNDLNRM releases the storage previously obtained and calls the .RESUME function program, which returns to SVCINT. SVCINT will, in turn, return to the calling program. If the user has not specified NOWAIT, HNDLNRM sets a switch to indicate that the called program was WAIT.

HNDLNRM then places the standard override information into a buffer and calls the PRINTR function program to print it. (This information is in the work area after the call to .RESUME; it consists of the address of the calling program, the name of the called program, the SVC old PSW, the normal return address, and the error return address.)

If the called program was WAIT and the user specified WAITSAME, or if a program other than WAIT was called, HNDLNRM proceeds as follows. If GPRSB was specified, it moves the contents of general purpose registers 0 through 7 as they existed at the time of the call, from the work area to the print buffer and calls PRINTIO to print the contents of the buffer. HNDLNRM does the same for general purpose registers 8 through 15. Next, if FPRSB was specified, it moves the contents of the floating-point registers, as they existed at the time of the call, to the print buffer and prints them. If GPRSA and FPRSA are specified, HNDLNRM follows similar procedures to print the contents of the general purpose and floating-point register as they exist upon return from the called program. Then, if parameter list printing was specified, HNDLNRM moves successive doubleword entries from the parameter list to the print buffer until it is filled, and prints the contents of the buffer. If a second line of the parameter list was specified, HNDLNRM does the same for the next series of doublewords in the parameter list. Finally, HNDLNRM calls the FRET function program to release first the work area and then the .RESUME function program to SVCINT and eventually to the calling program; HNDLNRM then returns to SVCINT.

If the called program was WAIT and the user specified either WAIT1 or WAIT2, HNDLNRM merely prints the corresponding number of lines of the parameter list, releases the work area, and returns to SVCINT by calling .RESUME.

### Error Override Operations

The error override handling program (HNDLERR) receives control from SVCINT when error overriding is in effect and an error is encountered during execution of the called program. The logic of HNDLERR is essentially the same as that of the normal override handling program. However, as part of initialization, HNDLERR sets switches for maximum printing. Thus, regardless of whether or not the called program is WAIT, each time HNDLERR gets control it prints the standard override information, the contents of the general purpose and floating-point register (both before and after the call), and two lines of parameter list.

## SYSTEM RESTART

CMS operation can be restarted when the user issues a RESTART or IPL request while in the DEBUG environment. When such a request is made, the DEBUG command issues a call to the CMS "IPL" command, which reads into core a clean version of the CMS nucleus. CMS nucleus in-core reinitialization is no longer supported by the DEBUG request RESTART.

Note: The CMS 'IPL' command can also be invoked from the DEBUG environment, or CP can be called upon to re-IPL through IPL CMS or IPL 190. If CMS is being run as a stand-alone program, CMS can be IPL'ed from the 360 operator console.

# SECTION 4: COMMAND PROGRAM DESCRIPTIONS

## FILE CREATION, MAINTENANCE, AND MANIPULATION

This section describes the processing performed by the various CMS command programs. The calling sequence and parameter list for each command are described.

Note:

1. A few CMS programs require alignment of the parameter list to be on a doubleword boundary, notably OFFLINE, MACLIB, and TXTLIB. If any of these programs is called from within a written program, make sure the parameter list is doubleword aligned (for example, preceded by DS 0D).

   Other CMS programs require only that the parameter list be fullword aligned.

   As a precaution, you may, of course, align all parameter lists on a doubleword boundary, if desired.

2. Each parameter list passed to a command should end with the constant X'FFFFFFFF'. For example, see the HNDINT function program described under "Nonterminal I/O".

3. The four bytes following a CMS Supervisor Call (SVC 202, or SVC X'CA') may contain the error return address of the form:

   ```
   DC AL4 (routine)
   ```

   If the high-order byte is nonzero, SVCINT will assume no error return address provided, and will transfer control to SDTERR if an error return from the given program should occur.

### ALTER

FUNCTION: To alter the identification of a file or related group of files on a read-write disk.

CALLING SEQUENCE:

```
LA    R1, PLIST    R1 must point to P-List as usual
SVC   X'CA'
DC    AL4 (ERROR)
```

ENTRY REQUIREMENTS:

R1 must point to ALTER parameter list:

```
        DS    0F
PLIST   DC    CL8'ALTER'
        DC    CL8'        '   Old Filename ('*' means all names)
        DC    CL8'        '   Old Filetype ('*' means all types)
        DC    CL2'  '         Old Filemode ('*' means any read-write disk)
        DC    CL6'        '   Not used
        DC    CL8'        '   New Filename (= or * means no change)
        DC    CL8'        '   New Filetype (= or * means no change)
        DC    CL2'  '         New Filemode (= or * means no change)
        DC    CL8'(       '   Option delimiter
        DC    CL4' option '   either or both TYPE to type the identifier(s)
                              of ALTER'ed files
        DC    CL4' option '   or NOUP to suppress the updating of the
                              user's file directory
```

EXIT CONDITIONS:

Normal Return
R15 = 0 (and condition code = 0)

Error Returns
R15 nonzero (and condition code = 2)

CALLS TO OTHER ROUTINES:

ACTLKP, ADTLKP, FSTLKW, TFINIS, UPDISK

CALLED BY (where known):

Disk resident routines

MACROS USED:

ADT, AFT, FSTB, FVS

ERROR RETURNS: (R15 value at exit):

1. Old specified file cannot be found
2. New specified file already exists
3. Old mode is illegal for a change
4. No changes were made at all
5. Change of mode is illegal
6. New mode is illegal
7. Incorrect ALTER parameter list (for example, insufficient parameters)
8. Specified file is in Active File Table (cannot change a file while it is active)

EXAMPLES:

```
ALTER LOAD MAP P5 SAVED MAP P1
ALTER CMS-NUC ALPHANUM * LATEST LOADMAPS P1
ALTER SPRT * P SCRIPT = =
ALTER * EXEC P1 " EXEC P2
ALTER ** P5 = = P1
```

OPERATION: ALTER checks the parameter list for various types of errors, and gives error returns, with messages, for any error detected.

Only read-write disk(s) are checked for the files specified; read-only disks are ignored.

When the parameter list has been checked and appropriate flag-bits set as needed, ALTER checks for existence of the given file(s), and changes the file identification, as follows:

1. ACTLKP is called to check if the file to be changed happens to be active — that is, in the Active File Table. This is treated as an error — see error 8. In a couple of cases where this error has been known to occur, the calling program either forgot to close the file before ALTER'ing it, or tried to ALTER it first and close it afterwards. Thus, if this error should occur (a message is typed to warn the user), look for this type of bug in the calling program.

2. If the given file is not in the active file table, ALTER checks for the file by a call to FSTLKW. If not found, ALTER exits with a normal return if at least one file was changed, or with an error 4 (with message) if no files at all were changed.

3. If the given file was found by FSTLKW, ALTER checks to make sure that the file identified by the new name and type does not already exist in the active file table for the same disk (via ACTLKP call — error 8 return if found), or in the FST tables for the same disk (via FSTLKW). If it is found, an error 2 is returned.

4. If not, the file identification is altered as specified by the caller's parameter list, and a flag-bit is set if a change was actually made (for the subsequent exit as described above in step 2).

5. ALTER then checks the NOUP flag bit of ALTRFLG to determine if the NOUP option — to prohibit the updating of the file directory — was specified. If so, the flag bit used to signal that the file directory is to be updated via a call to UPDISK is not turned on.

6. ALTER then checks the flag bit in ALTRFLG to determine if the TYPE option — to type the identifier(s) of the file(s) altered to the console — was specified. If TYPE was specified, the PLIST is set up, and a call to TYPLIN types the identifier of the file.

7. Then a call to the TFINIS routine is made (if necessary) to temporarily close all output files for the disk involved, and then UPDISK is called to update the file directory.

8. Finally, if the parameter list specified all names and/or types, the process is repeated, starting at step 1, to alter all appropriate filenames, types, or modes as desired.

See the examples given above for an insight into the kind of legitimate changes that can be made by an ALTER command.

Installation Note:

ALTER is a transient-disk-resident command.

If another transient-disk-resident command should be programmed which required the use of ALTER as a called subroutine, the ALTER program should be included with it in the module of the new program (NEWPROG), and the new program call ALTER (via BALR) when needed. ALTER saves and restores registers, so the calling program would not have to do that.

```
Example: LOAD NEWPROG ALTER (TRANS TYPE)
         GENMOD NEWPROG
```

In any event, the system mask must be 00 when ALTER is invoked, as is generally required for all nucleus and transient-disk-resident routines.

## CEDIT

FUNCTION: To create and/or edit card images files.

ATTRIBUTES: Disk resident

Note: For a detailed explanation of CEDIT, see the write-up on the EDIT command.

## CLOSIO

FUNCTION: To close out a file and cause an EOF on an output file to the card punch or printer. The output file may now be spooled out to the real device.

ATTRIBUTES: Nucleus resident

CALLING SEQUENCE:

```
LA    1, PLIST
SVC   X'CA'
.
.
.
```

```
PLIST   DC     CL8'CLOSIO'
       [DC     CL8'READER']
       [DC     CL8'PRINTER']
       [DC     CL8'PUNCH']
       [DC     CL8'OFF']
       [DC     CL8'ON']
```

OPERATION: CLOSIO determines whether the first entry in the parameter list is for a card reader, a printer, or a card punch, and issues a start I/O (SIO) to that device. In each case, the CCW used for the operation is invalid and is interpreted by the control program (CP67) as a signal that I/O operations on the corresponding virtual device are complete. CLOSIO repeats this for each parameter in the list. When all parameters are processed, it returns (via SVCINT) to the calling program. If no parameters are supplied, CLOSIO closes all three devices.

If the parameter OFF is entered, any subsequent calls to CLOSIO will be ignored until the ON parameter is supplied.

## COMBINE

ATTRIBUTES: Filename — COMBINE, disk resident module

FUNCTION: To concentrate one or more files into a new file.

CALLING SEQUENCE:

```
        LA     1,PLIST
        SVC    X'CA'
         .
         .
         .
PLIST   DC     CL8'COMBINE'
        DC     CL8'        '   new filename (=means retain old name)
        DC     CL8'        '   new filetype (= means retain old type)
        DC     CL8'        '   new filemode (= means retain old mode)
        DC     CL8'        '   old filename1 (* means all filenames)
        DC     CL8'        '   old filetype1 (* means all filetypes)
        DC     CL8'        '   old filemode1
        DC     CL8'        '   old filename2
        DC     CL8'        '   old filetype2
        DC     CL8'        '   old filemode2
                .
                .
                .
```

```
DC     CL8'            '  old filename n
DC     CL8'            '  old filetype n
DC     CL8'            '  old filemode n
DC     CL5' (TYPE'        option to type file identifiers
                          of files being COMBINE'd
```

Entry Requirements:

R1 must point to the COMBINE parameter list.

EXIT CONDITIONS:

Register 15 contains the error code, if any, to the user. If no errors were encountered, register 15 contains zero.

ERROR RETURNS: (R15 value at EXIT)

1. INVALID PARAMETER LIST
2. OUTPUT DISK TO BE WRITTEN ON NOT LOGGED IN
3. OUTPUT DISK NOT IN WRITE STATUS
4. INPUT FILE DOES NOT EXIST
5. ERROR WHILE TRYING TO WRITE
6. ERROR TRYING TO ALTER OUTPUT FILE
7. INPUT FILE DISK, NOT LOGGED IN
8. DISK SPECIFIED NOT IN ACTIVE DISK TABLE
9. ATTEMPT TO COMBINE FIXED AND VARIABLE LENGTH FILES

Calls to Other Routines:

ADTLKP, FSTLKP, STATE, ALTER, ERASE, RDBUF, WRBUF, FINIS

OPERATION: After checking that all necessary parameters are present; COMBINE checks for either a '=' or '*' in each argument, setting an appropriate flag for each specified if the (TYPE option is specified, a flag is also set. COMBINE then checks to see if the output disk is available in write status. If so, COMBINE checks that the input disk is logged in.

COMBINE then checks the flag it set to see if either a '*' or '=' was specified.

If not, the input filename is moved into a STATE parameter list which verifies that the file exists. If the file is found, the location of the file status table is obtained and the input file is read. Data is written into a temporary file called (TEMP) (FILE). When an END of FILE is encountered FINIS is called to close the input file and determines if another file is to be combined repeating the above procedure to this point. The output file is then FINIS'ed. COMBINE calls ERASE in the event there already exists a file with the same name and type as the new file is to have, then ALTER's the (TEMP)

(FILE) to the new filename and filetype. After successful completion and prior to returning to the user or caller, COMBINE references NUCON and turns the page release flag on.

When the program returns to INIT, this flag is checked and, if it is on, INIT issues a diagnose X'10' to CP to release the user pages from X'12000' up to the value of LOWEXT. COMBINE then calls FINIS to close all active files and branches to the caller.

If a '=' or '*' was specified, COMBINE sets the input filename, filetype, and filemode in the usual parameter lists and calls FSTLKP to determine if the file exists. If the file exists and an '*' was specified in either the filename, filetype, or both, a negative sign bit is moved into the pointer to the parameter list signifying that the disk table is to be searched again for another file. With a file found COMBINE moves into the output parameter list, the name, type or mode depending on whether a '=' was specified in either of the fields. A check is then made to determine if the (TYPE option was specified and if it was, the name and/or type, depending on the field in which an asterisk was specified, is moved into a TYPLIN plist and typed to the terminal. COMBINE then calls RDBUF to read the input file and calls WRBUF to write into the (TEMP) (FILE). FINIS is called to close the input file and output file. ERASE is called to erase the original file, if any, that has the same name and type as the new output file will have and ALTER's the (TEMP) (FILE) to the appropriate name. If an '*' was present for filename, or filetype, COMBINE returns to recall 'FSTLKP' to continue where it had previously left off. If another correct file is found, the above procedure is repeated. If not, COMBINE releases the user pages as described in the preceding paragraph and exits to the caller.

## EDIT

FUNCTION: To create and/or edit card image files.

CALLING SEQUENCE:

```
LA     1,PLIST
SVC    X'CA'
.
.
.
          {'CEDIT}
DC     CL8{'EDIT' }
DC     CL8'        '   filename (optional)
DC     CL8'        '   filetype
```

OPERATION: The operation of the Editor programs consists of initialization, input environment processing, and edit environment processing. The functional difference between EDIT and CEDIT is that EDIT will place the entire file into core, whereas CEDIT will READ into core only the current line and WRITE to an intermediate file when finished with each line.

Note 1: When in the input and edit environments, a line entered from the terminal is read into a buffer called DDLIN. The input line is formatted according to tab settings in a buffer called TABLIN. The line to be written into the output file is located in a buffer called LINE. The current line always resides in the buffer called LINE.

Note 2: During editing, the file being read from and updated is an intermediate file designated as (INPUT)(FILE) P1. The new file being created or written into (that is, the updated version is an intermediate file designated as (INPUT)(FILE) P1). These two files exchange roles whenever the pointer is positioned to the top of the file.

INITIALIZATION: EDIT will warn the user if the intermediate files (that is, (INPUT) (FILE) P1 and (INPUT1)(FILE) P1) exist. It then sets the TABPNT storage cell to point to the tab settings to be used. (The tab settings to be used are determined according to the filetype of the file. TABPNT always contains a pointer to the tab settings that are currently in effect.) If the user did not provide a filename, EDIT immediately enters the input environment. If the user did provide a filename, EDIT calls the STATE function program to determine if the file exists. If it exists, EDIT enters the edit environment. If the file does not exist, EDIT types a message at the terminal to that effect and enters the input environment to enable the user to create a file.

INPUT ENVIRONMENT PROCESSING: The input environment of the EDIT command program is entered when the user wishes to create a new file or to add records to an existing file. When this environment is entered, EDIT types the message INPUT at the terminal. Next, it reads the user's first input line. It then spreads the line according to the tab settings that are currently in effect and writes the previous input line into the file being created. Finally, EDIT moves the input line into the output buffer (LINE) from where it will be written into the file being created when the user enters the next input line from the terminal. EDIT repeats this procedure of reading a line from the terminal, spreading it, and inserting the previous line into the file until the user enters a line with only a carriage return. The carriage return indicates that the user wishes to enter the edit environment and EDIT passes control to the portion of code that controls execution in the edit environment.

EDIT ENVIRONMENT PROCESSING: The edit environment of the EDIT command program is entered when the designated file exists or when the user switches to it from the input environment. The various requests for editing functions are processed by correspondingly named programs.

Control Within the Edit Environment: When the edit environment is entered, EDIT types the message 'EDIT:' at the terminal. It then reads the user's first request. Next, EDIT determines the nature of the request through a table-lookup procedure and branches to the program that is responsible for satisfying the request. When that program is done, it returns control to the control element, which obtains the next request. This request is satisfied in a similar manner.

Delete Request: EDIT reads successive lines from the file being updated (that is, (INPUT) FILE P1). It does not transmit these lines to the file being created. EDIT then returns for the next request. (If an end-of-file occurs during reading, EDIT signals the condition via terminal message and returns for the next request.)

Insert Request: If this request does not provide a line, EDIT enters the input environment. If there is a line, it places the current line into the file being created. It then spreads the line provided with the request according to the current table settings. When the line is spread, EDIT makes it the current line by moving it to the LINE buffer. EDIT then returns for the next request.

Retype Request: EDIT spreads the line supplied as part of the request according to the current tab settings. It then overlays the current line (in the LINE buffer) with the spread input line and returns for the next request.

Serial Request: EDIT checks for the (NO) parameter. If present, it sets a switch to indicate that no serialization is to take place. If not present, EDIT saves the first three characters following the request name for use in columns 73-75 of the output lines (that is, card images). EDIT next obtains the increment field (if any) and saves it for future use. (If an increment field is not provided, EDIT assumes an incremental value of 10.) It then returns for the next request.

Backspace Request: EDIT saves the backspace character for subsequent use. It then returns for the next request.

Tab Definition Request: EDIT saves the tab definition character for use in spreading subsequent input lines. It then returns for the next request.

Tabset Request: EDIT saves each of the tab settings supplied in the corresponding entry in the tab setting table (DEFTAB). These will be used during the spreading of subsequently processed lines. It then returns for the next request. (A pointer to the current tab settings is always kept in a storage cell called TABPNT.)

Quit Request: EDIT calls the FINIS command program to close both the file being changed or created and ERASE any work files. In effect, the file has remained unchanged. It then returns (via SVCINT) to the calling program, which is usually INIT.

Verify Request: EDIT sets the verify mode switch on. This switch is referred to by various other request processing programs. EDIT then returns for the next request.

Brief Request: EDIT sets the verify mode switch off to indicate that brief mode is in effect. It then returns for the next request.

Input Request: EDIT goes directly into the input environment.

Overlay Request: EDIT spreads the line entered as part of the request according to the current tab settings. It then scans the spread line for non-blank characters. Upon encountering one, it replaces the character located at the same relative position on the current line (in LINE) with that non-blank character. When the entire input line has been scanned, EDIT determines if verify mode is in effect. If it is, EDIT types the updated current line at the terminal. If verify mode is not in effect, EDIT does not type the updated current line. EDIT then gives control to the REPEAT request routine to determine if OVERLAY processing is to be repeated for the next line in the file. If not, EDIT returns for the next request.

Blank Request: EDIT spreads the line entered as part of the request according to the current tab settings. It then scans the spread line for non-blank characters. Upon encountering one, it replaces the character located at the same relative position in the current line (in LINE) with a blank. When the entire input line has been scanned, EDIT determines if verify mode is in effect. If it is, EDIT types the updated current line at the terminal. It then gives control to the REPEAT request routine to determine if BLANK processing is to be repeated for the next line in the file. If not, EDIT returns for the next request.

Repeat Request: EDIT stores the specified number of repeats and decrements this count each time the REPEAT routine is entered. If the count is not zero, the current line is written into the file being created (that is, (INPUT) FILE P1), and the next line is read from the file being updated (that is, (INPUT1) FILE P1). Control is then given to the routine that processes the request being repeated (either OVRLAY or BLKOUT). If the repeat count is zero, EDIT returns for the next request.

Next Request: The current line pointer is moved ahead n lines within the file. EDIT then returns for the next request.

Print Request: EDIT types the current line of the file onto the online terminal. EDIT repeats this n times and returns for the next request. (If the user specifies either L or LINENO, serial numbers are included with each typed line. If the user does not specify either of these, serial numbers are not included. Also, if an end-of-file is encountered during the reading of the file being updated, EDIT signals the condition via terminal message and returns for the next request.)

Top Request: If an end-of-file has been reached on the file being updated, CEDIT switches that file with the file being created and returns for the next request. After the switch, the file that was being used for output is used as input; thus, the newly created file becomes the one to be updated on the next pass. If an end-of-file has not been reached, CEDIT transfers the records remaining in the file being updated to the file being created. It then switches the roles of the two files and returns for the next request.

The switching of files is done in the following manner: The FINIS command program will close the newly created file and then the POINT function program will position the read-write pointers to the first item in the file. It does the same for the file that was being updated. Next, the file-names in the corresponding read and write parameters lists are switched. (Refer to the discussions of the RDBUF and WRBUF function programs.) Finally, the old file is erased.

For EDIT, the incore line pointer is placed at the top of the file.

Backup Request: The line pointer is moved backward n lines within the file.

Bottom Request: The line pointer is positioned at the end of the file — last line.

Locate Request: (If an end-of-file is in effect when this request is issued, EDIT switches the two files in the same manner as it does for a TOP request. If an end-of-file is not in effect, no switch is made.) EDIT reads a line from the file and scans across it for a string of characters that match those supplied with the located request. If no such string exists in that line, EDIT will obtain the next line in the file. EDIT scans this line for a match. If a match does not occur, it repeats the procedure. If a match is found, EDIT determines whether verify mode is in effect. If it is, EDIT types the line having the matching string at the terminal and returns for the next request. If an end-of-file is encountered before a match, EDIT signals this via a terminal message and returns for the next request.

Find Request: EDIT spreads the line supplied with the request according to the tab settings currently in effect. (If an end-of-file is pending when this request is made, EDIT switches the two files in the same manner as it does for a TOP request. If an end-of-file is not pending, no switch is made.) It then reads a line from the file being updated. Next, EDIT scans this line to determine if it contains the same characters supplied in the input line in the same relative positions. If the line does not, EDIT will obtain the next line from the file and scan this line in a similar fashion. EDIT repeats this procedure until either a match or an end-of-file is encountered. If a match occurs, EDIT determines whether verify mode is in effect. If it is, EDIT types the line containing the matching character string at the terminal and returns for the next request. If verify mode is not in effect, it merely returns for the next request. If an end-of-file occurs before the match, EDIT signals this via terminal message and returns for the next request.

Change Request: (If an end-of-file is in effect when this request is issued, EDIT switches the two files in the same manner as it does for a TOP request. If an end-of-file is not in effect, no switch is made.) EDIT scans the line supplied with the CHANGE request and retrieves the character string to be replaced. It then determines whether the current line contains this string. If it does not, EDIT returns for the next request. If the current line contains the string of characters to be replaced, EDIT retrieves the replacement characters from the line supplied with the CHANGE request. It then makes the requested replacement of characters in the current line. (The length of the current line will be appropriately adjusted to accommodate the replacement characters.) If the global option is specified, EDIT continues to scan the current line for a second occurrence of the string of characters to be replaced. If there is a second occurrence, EDIT makes the requested replacement. EDIT continues to scan the current line until all such occurrences have been replaced. (If the global option is not specified, EDIT only replaces the first occurrence in the current line.) Then, if verify mode is in effect, EDIT types the updated current line at the terminal. If more than one line is to be considered for change, EDIT writes the updated current line into the file being created and reads the next line from the file being updated. This line then becomes the current line. EDIT then scans the current line to determine whether it contains the character string to be replaced. If it does, EDIT makes the requested replacement (more than one replacement if the global option is specified and there is more than one occurrence of the string in the line), types the updated line at the terminal if verify mode is in effect, and determines if another line is to be considered for change. If the current line does not contain the character string to be replaced, EDIT merely determines whether there is another line to be considered for change. If there is, EDIT replaces the current line into the file being created and obtains the next line from the file being updated. EDIT repeats this procedure until n lines have been considered. (The default value if n is not specified on the CHANGE request is one.) If an end-of-file on the input file (that is, the file being updated) is encountered before n lines have been considered, EDIT types a message at the terminal to that effect and returns for the next request.

File Request: If the user did not provide a filename on the EDIT command and also did not provide one on the FILE request, EDIT prompts the user (via a terminal message) to reissue the FILE request and supply a filename. When the user complies, EDIT determines if an end-of-file on the file being updated is pending. If it is not, EDIT transfers the remaining lines in the file being updated to the file being created. It then calls the FINIS command program to close both the file being updated and the file being created. EDIT then calls the STATE function program to determine whether the new file already exists. If it does, EDIT calls the ERASE command program to erase that file. After the file has been erased or if no such file exists, EDIT calls the ALTER command program to change the designation of the file just created to the designation supplied by the user. It then erases the old file (that is, the one being updated), if necessary, and returns (via SVCINT) to the calling program, which is usually INT. (The old file need not be erased at this time if a top was done just before the FILE request was issued because it is erased as part of the top operation.)

Save Request: The current contents of the file are written on disk, FINIS is called to close the file, and a return to the INPUT mode is made.

## ERASE

FUNCTION: To delete a file or related group of files from the permanent, temporary, and/or other read-write disk(s).

ATTRIBUTES: Nucleus resident

CALLING SEQUENCE:

```
            LA      R1, PLIST        R1 must point to P-List as usual
               then either
            SVC     X'CA'            Call ERASE via SVC
            DC      AL4(ERROR)       Error-return (for example, if file not found)
                   or
            L       R15, AERASE      Where AERASE = V(ERASE)
            BALR    R14, R15         Call ERASE via BALR within Nucleus
            BNZ     ERROR            Transfer if error (for example, file not found)
```

ENTRY REQUIREMENTS:

```
            R1 must point to ERASE parameter list:
            DS      0F
PLIST       DC      CL8'ERASE'       (Note - immaterial if called by BALR)
            DC      CL8'     '       Filename
            DC      CL8'     '       Filetype
            DC      CL2' '           Filemode
                  or
            DC      X'FFFFFFFF'      Delimiter (necessary if filemode omitted)
```

EXIT CONDITIONS:

Normal Return (File successfully erased):
R15=0 (and condition - code = 0)
Incorrect ERASE Parameter List (Error 1)
R15=1 (and condition - code = 2)
File(s) not Found (Error 2)
R15=2 (and condition - code = 2)
File Faulty (but erased) - Error 3
R15=3 (and condition - code = 3)

CALLS TO OTHER ROUTINES:

ACTFRET, ACTLKP, ACTNXT, DISKDIE, FREE, FRET, FSTLKW, QQTRKX, RDTK, TFINIS, TRKLKPX, UPDISK

CALLED BY (where known):

DISK, FINIS, GENMOD, LISTF, LOAD, OFFLINE, plus disk resident routines.

MACROS USED:

ADT, AFT, FSTB, FVS

OPERATION: ERASE checks the parameter list for errors by the caller. The filename and filetype must each be given, or else a single asterisk to indicate all names and/or types. The filemode may be omitted (that is, =X'FF'), in which case the first read-write disk is assumed. If not omitted, the filemode must be alphabetic, or a single asterisk. If alphabetic, a mode number is acceptable. If the mode is '*', all read-write disk(s) are searched by ERASE.

For example, a call of ERASE * TEST P5 would erase all text files on the P-Disk that had a mode number of 5. All other text files on any disks would remain intact, and all other P5 files would remain also.

If any errors are detected in the parameter list, the message INCORRECT ERASE PARAMETER-LIST is given, error 1 is returned, and nothing is erased.

After checking the parameter list and setting flagbits as needed, ERASE checks for a given file and deletes it if found using the following procedure:

1. ACTLKP is called to determine if perchance the file to be erased is still active — that is, in the Active File Table (AFT). If it is (only a file on a read-write disk is acceptable, of course), then it is temporarily closed via a special EFINIS call to the TFINIS routine, which performs just enough of the normal closing steps ordinarily performed by FINIS to permit the file to be successfully erased. Proceeds then to step 3 below.

2. If not found by ACTLKP, then ERASE calls FSTLKW to find the file. If not found, exit is made from ERASE as described in step 14 below.

3. When the file has been found either by ACTLKP (and EFINIS called), or by FSTLKW, then TFINIS is called to temporarily close all output files for this particular disk (unless this was already accomplished by an earlier excursion through this procedure for another file on the same disk).

4. ERASE then checks the TYPE option flag bit to determine if the user specified that the identifier(s) of the file(s) being erased are to be typed to the console. If the bit is on, the PLIST is set up, and a call is made to TYPLIN.

5. Before releasing any tracks belonging to the file that has been found, ERASE calls a special entry in the UPDISK routine (see description of entry (2) in UPDISK routine for details) to reserve enough disk records for a new file directory, to be updated when the file has been erased. This procedure is part of CMS's double directory scheme, and ensures that the file directory for the disk from which the file is being erased is updated when and only when the erase has been completed. (If any system malfunction or user intervention interrupts the process before completion, the old file directory and the file being erased are <u>both</u> still intact.)

6. Then (unless it is already available), 1000 bytes of free storage are obtained via FREE, for use in reading in the first and other (if any) chain links of the file.

7. Next the first chain link of the file is read into core, into the first 200 bytes of the free storage area, via RDTK.

8. The data blocks pointed to by the first chain link are then released via TRKLKPX, and the first chain link itself via QQTRKX (the first chain link remaining in core, however).

9. If any data blocks remain, according to the FSTDBC data-block-count in the FST entry, then additional chain links are read into core, as pointed to by the first chain link. For each of these Nth chain links, the data blocks pointed to thereby are released via TRKLKPX, and then the chain link itself. This process continues, with a count of data blocks returned being decremented, until there are none left, or all available chain links have been exhausted.

10. At this point, all data blocks and chain links have been given back to the QMSK and QQMSK via appropriate calls to TRKLKPX and the one call to QQTRKX. Now a check is made to see if perchance the file being erased happens to be contained in STATEFST. If so, the 48 bytes at STATEFST are cleared to reflect the deletion of the given file. (Note -RDBUF utilizes the STATEFST information in some circumstances; thus it must be either correct or null.)

11. Next provisions are made to keep the FST hyperblocks compacted, for the disk on which the file was found and erased. In this process, the last FST entry for the disk involved is moved to where the FST entry was for the file that we just erased, and the place from which it was moved is cleared. A check is made of the active

file table via ACTNXT in case an active file entry points to the file moved, in which case the pointer is corrected; the pointer following STATEFST is also checked, and corrected if necessary. In any event, the compacting is carefully accomplished, with all pointers, displacements, block counts, etc., being corrected as necessary.

12. Finally, a call to the other special entry of UPDISK (entry 3 in the UPDISK description) is made to complete the updating of the file directory for the disk involved.

13. At this point, if the entire FST hyperblock and the last FST entry in the preceding hyperblock have all become clear, the last hyperblock is returned to free storage, and all pointers and counts corrected accordingly. (This is done to avoid keeping a number of empty hyperblocks in core in case a large number of files are erased.)

14. Finally, the entire procedure is repeated starting at step 1, if the parameter list specified all names, types, or modes.

15. When all appropriate erasing (if any) has been completed, ERASE returns the 1000-byte buffer to free storage, and exists to the caller with the appropriate error code.

If no files at all were erased, ERASE returns an error code 2, but without an error message. (Several system programs call ERASE to eliminate old listings, old text files, etc., in case they might exist, so that an error message for FILE NOT FOUND in ERASE itself would be impractical.)

Several error conditions are detected by ERASE. On one of these, a permanent I/O error in reading in a chain link due to hardware disk errors, ERASE purposely invokes the code at DISKDIE (within the FINIS command) to leave the file directory intact until the disk error can be corrected.

On all others, when the error is detected, ERASE ceases to give back records using TRKLKPX and/or QQTKRX, but deletes the files and compacts the directory as usual. An error 3 is given on exit, when ERASE is finished.

This feature makes it feasible to ERASE a faulty file from one's directory without endangering the integrity of other files on the same disk.

## FILEDEF

FUNCTION: To allow the user to specify, in a manner similar to the OS data definition card, I/O devices and certain file characteristics which will be used by a program at execution time. Can also be used to modify, delete and list previously defined file descriptions.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
        LA      R1, PLIST
        SVC     X'CA'
        DC      AL4 (error)
```

ENTRY REQUIREMENTS:

```
        R1      must point to FILEDEF parameter list:
        DS      0F
PLIST   DC      CL8 'FILEDEF'
        DC      CL8 'fileid'
        DC      CL8 'device'
        DC      CL8 '       '       parameter pairs
        DC      CL8 '       '       parameter pairs comb'd
        DC      CL8 '('             option delimiter
        DC      CL8 '       '       options
        DC      8X  'FF'            fence
```

ENTRY POINTS:

FILEDEF

EXIT CONDITIONS:

```
        Normal Return
              R15 = 0
              RØ  = address of FCB
                    positive if already exists
                    negative if obtained or modified by this call
        Error Return
              R15   non zero (See Error Returns)
```

CALLS TO OTHER ROUTINES:

FREE, FRET, CONWRITE

EXTERNAL REFERENCE:

SYSREF

CALLED BY:

SOIOMAN, LANGUAGE PROCESSORS,
Execution interface — PLI, FORTRAN

MACROS USED:

TYPE, CMSTYPE, CMSCB, CMSYSREF, CMSREG

TABLES AND WORK AREAS:

| | | |
|---|---|---|
| COPYLIST | plist copy | internal |
| FCB | file control block | free storage |

REGISTER USAGE:

| | |
|---|---|
| R0 | — Address Return |
| R1 | — Plist on entry |
| R2 | — Temporary |
| R3 | — FCBLEAD |
| R4 | — FCBSECT |
| R5 | — Plist — working copy |
| R6 | Working temporary |
| R7 | |
| R8 | |
| R9 | |
| R10 | — internal linkage |
| R11 | — conversion |
| R12 | — Base |
| R13 | — Save area |
| R14 | External linkage |
| R15 | |

ERROR RETURNS:

| | | |
|---|---|---|
| 1. | FIL001: | Parm 1 invalid |
| 2. | FIL002: | Mode number missing |
| 3. | FIL003: | Parm missing after xxxxxxxx |
| 4. | FIL004: | Invalid Parm after xxxxxxxx |
| 5. | FIL005: | Bad LRECL/BLKSIZE values |
| 6. | FIL006: | Illegal clear request |
| 7. | FIL007: | Filename/filetype required |
| 8. | FIL008: | Unknown device type |
| 9. | FIL009: | Bad opted parameters |
| 10. | FIL010: | Invalid option |

OPERATION: The starting address of the chain of FCB's is obtained from the nucleus. The PLIST is then analyzed to determine if there is enough space in the transient area for a working copy of the PLIST; if not, free storage is obtained and the PLIST copy is placed there.

The PLIST is then examined for options. If either PERM and/or NOCHNG is specified, appropriate flags are set. Any other options are invalid, and cause an exit with error code 10.

Subsequent processing depends on the operands specified. The first operand is checked, and depending on its contents, operation continues as described below.

No Operand. FILEDEF with no operand requests a list of current file definitions. FCBNUM contains the number of entries in the chain of FCB's. This is used to loop through the chain. For each, FCBDD and FCBDEV is typed to the terminal. For definitions to DSK, FCBDSNAM (the CMS filename) and FCBDSTYP (the CMS filetype) are also typed.

*CLEAR. All FCB's on the chain are released except those flagged PERManent. These are released only when specifically cleared.

Numeric Fileid. The number is converted to a data set reference number (i.e., FTxxFnnn). Processing continues as described under alpha fileid below.

Alpha Fileid. FCB is used to loop through the FCB chain in free storage looking for the specified FCB. If no match is found, the new FCB flag is set, free storage is obtained, and the address of this is placed in the first word of the last FCB on the chain. The address of the new FCB is put in register 0 as a negative quantity and saved to be passed back to the user when parameter processing is complete. If the PERM flag is set, the high order byte of the new FCB is flagged PERManent.

If a matching FCB is found, and the NOCHNG flag is set, FILEDEF returns to the user with the address of the FCB in register 0.

If a matching FCB is found and the NOCHNG flag is not set, the old FCB is saved in case of an error, the old entry flag is set, and the address of the FCB is negatively stored in register 0. If the PERM flag is set, the FCB is flagged PERManent.

Processing is then dependent on the device type and related parameters specified.

Device DUMMY. For device DUMMY an FCB is created with a ddname of dummy ' and a device type of X'00'.

Device Batch. For device BATCH an FCB is created with a device type of disk.

The following chart shows the RANGE of each option and the device types to which it applies.

| Parameter | Range | Default | Disk (1) | Tape | Reader | Punch | Printer | Console | FCB Field |
|---|---|---|---|---|---|---|---|---|---|
| KEYLEN | ≤61439 | --- | ✓ | | | | | | JFCKEYLE |
| XTENT | ≤61439 | 50 | ✓ | | | | | | FCBXTENT |
| LIMCT | ≤61439 | --- | ✓ | | | | | | JFCLIMCT |
| OPTCD | E, F, A, R | --- | ✓ | | | | | | JFCOPTCD |
| DSORG | PS, DA, IS, PD | --- | ✓ | | | | | | FCBDSORG |
| DISP | MOD | --- | ✓ | | | | | | FCBIND2 |
| RECFM | FB, V, VS, VB, VBS, U | --- | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | FCBRECFM |
| BLKSIZE | ≤61438 | --- | ✓ | ✓ | ✓ | | | | FCBBLKSZ |
| LRECL | ≤61438 | --- | ✓ | ✓ | ✓ | | | | FCBLRECL |
| (2) AUXPROC | ABSOLUTE ADDRESS | --- | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | FCBPROC |
| MODE | C'16' | --- | | ✓ | | | | | FCBMODE |

(1) If there are no entries after disk, a default dsname of file and dstype equal to the ddname will be established.

(2) Auxproc is the auxiliary processing routine address which is primarily used by language processors for special handling routines. Referenced by SOEOB.

## FINIS

FUNCTION: To allow the user to close one or more selected files.

ATTRIBUTES: Nucleus resident

CALLING SEQUENCE:

```
         LA     1,PLIST
         SVC    X'CA'
         .
         .
         .
PLIST    DC     CL8'FINIS'
         DC     CL8 {'Filename'}
                    {'*'       }
         DC     CL8 {'Filetype'}
                    {'*'       }
                CL2 {'Filemode'}
         DC         {'*'       }
```

OPERATION: Refer to the description of FINIS under "File Management Function Programs".

Note: Since INIT closes all files after each command, FINIS as a terminal command would not normally be issued. For FINIS from the terminal, error 6 "NO FILES OPEN" would occur.

## LISTF

FUNCTION: To list the names of the files that exist on one or more of the CMS disks.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
         LA     R1,PLIST        R1 must point to P-List as usual
         SVC    X'CA'
         DC     AL4(ERROR)
```

ENTRY REQUIREMENTS:

```
         R1 must point to LISTF parameter list:
         DS     0F
```

```
PLIST DC     CL8'LISTF'
      DC     CL8'        '    filename  or '*' or omitted
      DC     CL8'        '    filetype or '*' or omitted
      DC     CL8'        '    filemode or '*' or omitted
      DC     CL8'(       '    precedes options (if any)
      DC     CL8'        '    option 1 (if any)
             .
             .
             .
      DC     CL8'        '    option N (if any)
      DC     X'FF'            signals end of P-List
```

EXIT CONDITIONS:

Normal Return
R15 = 0
Error Returns (R15 values, with messages as shown):
R15 = 1: INVALID LISTF PARAMETER LIST.
R15 = 2: FILE NOT FOUND
R15 = 3: NO R/W DISK LOGGED IN

CALLS TO OTHER ROUTINES:

ADTLKP, ADTNXT, ERASE, FINIS, WRBUF

CALLED BY:

User

MACROS USED:

ADT, FVS

OPERATION: The disk(s) searched for the given file(s) are determined by LISTF as follows:

1. If filemode is given, ADTLKP is called to reference the given disk; if found, LISTF searches the directory to find the given file(s). If not found by ADTLKP or if the disk is not logged in, error 2 (FILE NOT FOUND) is returned.

2. If the filemode is omitted, ADTNXT is called (repeatedly if necessary), and LISTF searches all read-write disk(s) currently logged in, to find the given file(s). If no read-write disks are logged in, error 3 (NO R/W DISK LOGGED IN) is returned.

3. If the filemode was given as asterisk (*), then ADTNXT is called as above, and all disks, read-write and read-only, are searched by LISTF for the given file(s).

When LISTF, in scanning a particular FST table as obtained from ADTLKP or ADTNXT, finds an FST entry whose filename and filetype satisfy the parameter list, it moves the filename, filetype, filemode, and number of data blocks in the file from that file status table to the buffer. If the EXEC option is not requested, it then calls the TYPLIN function program to type the contents of the buffer at the terminal. (The output lines are preceded by an appropriate heading.) LISTF repeats this procedure for each file status table whose filename and filetype fields satisfy the listing requirements. When the scan of all participating file status tables is completed, LISTF returns to the caller.

If the EXEC option is requested, the contents of the buffer are not written to the terminal. Instead, a CMS EXEC P1 file, containing the dummy arguments "&1 &2" followed by the buffer contents, is created. This file may later be accessed by the EXEC program, which will replace the dummy arguments.

Along with the statistics of file: name, type, mode, and number of records, the date and time that the file was last opened for writing will be obtained from the FST, and made available to the printed line.

If the SORT option is specified, the printed output will group together all identical filetypes.

Several options are available in the LISTF command. See the CMS User's Guide for full information.

Note:

If the P-Disk is read-only and LISTF is given with the filemode omitted, an error 3 (with message) is returned.

For listing files on a read-only P-Disk, therefore, be sure to include the mode letter P.

## OFFLINE

FUNCTION: To perform the necessary conversion between unit record files and disk files and vice versa.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
LA   1,PLIST
SVC  X'CA'
.
.
.
DS   0D
```

```
PLIST DC    CL8'OFFLINE'
                ⎧READ
                ⎪PUNCH
                ⎪PUNCHCC
      DC    CL8'⎨PUNCHOT
                ⎪PRINT
                ⎪PRINTCC
                ⎪PRINTUPC
                ⎩PRINTVLR
      DC    CL8'        '    filename or *
      DC    CL8'        '    filetype
      DC    CL2'  '          filemode
```

Note: Asterisks can be used in place of the filename and filetype if special read mode is desired; that is, if OFFLINE READ filename filetype control cards precede each logical deck.

OPERATION: OFFLINE calls the SVCFREE function program to obtain a block of free storage for use as a work area. Next, it calls the STATE function program to locate the file status table for the specified file. (This file status table will exist only if a file identically named exists in the system.) From here on, the operation of the OFFLINE command program depends on which function was invoked.

READ: The operation of the read portion of the OFFLINE command program depends on whether or not the caller selects special read mode.

Special Read Mode (*): OFFLINE reads the first card from the card reader via a call to the CARDRD function program. If this card is not an OFFLINE control card, OFFLINE signals the error and will then assume a control card of the form "OFFLINE READ . . NAME. . . . TYPE. .". (The user may then ALTER the file identification to what was intended.) If an asterisk was specified only in the filename field, OFFLINE will take the filename, filetype and filemode from the OFFLINE READ card and place that information in the parameter list. If asterisks were placed in the filename and filetype fields, and no mode specified, the filename and filetype from the OFFLINE READ card will be placed in the parameter list and a default mode of P will be placed into the parameter list. If '* * fm' was specified, the filename and filetype are taken from the OFFLINE READ card and put into the parameter list and the mode specified is the mode that is placed into the PLIST.

OFFLINE calls the STATE function program to locate the file status table entry for the indicated file. It will erase the copy of the old file once verification of the input file is obtained. Next, it reads a block of cards as described below. If neither another OFFLINE control nor an end-of-file is encountered during reading, OFFLINE calls the WRBUF function program to place the card images into a disk file. It repeats this procedure for the next block of cards in the reader.

If another OFFLINE control card appears in the input stream, OFFLINE calls SCAN to format it and TYPLIN to type it at the terminal. It then writes the remaining image of the previous file into the disk file, calls the FINIS command program to close that file, and returns to process the new file of cards following the second OFFLINE control card in the prescribed manner. OFFLINE repeats this procedure for each logical file of cards in the reader.

When an end-of-file is encountered during the reading of cards, OFFLINE places the remaining images into the last disk file, calls FINIS to close that file, calls CLOSIO to close card reader operations, calls the SVCFRET function program to release the storage previously obtained, and returns to the calling program. Thus, during processing, OFFLINE converts each card file in the card reader to a correspondingly named disk file.

No Special Read Mode: OFFLINE calls the ERASE command program to erase the identically named file (if one exists and if there are cards to be read). Next, it reads a block of cards as described below. It then calls the WRBUF function program to write the images into a disk file. OFFLINE repeats this procedure for each block of cards until an end-of-file occurs. At this time, it writes the remaining images into the disk file, closes that file (vis FINIS), closes card reader operations (via CLOSIO), releases the storage previously obtained (via SVCFRET), and returns to the calling program. Thus, if the special read mode is not selected, OFFLINE creates a single file from the cards in the reader.

In either case, (Special Read Mode (*) or not), OFFLINE will read the first data record and compare its record length against the specified length. If it gets an incorrect length, it checks to see if an error has been encountered and branches out with the appropriate error code. If there is no error, OFFLINE checks to see if the record length is 132 bytes; if affirmative, it types the message "RECORD LENGTH = 132 BYTES" on the console and continues to read and write as described in the two read mode descriptions. If the record length is not 132 bytes, OFFLINE assumes the file record length to be equal to the length of the first record read and continues to read and write as described in the Read mode descriptions depending on which was specified.

PUNCH: OFFLINE calls the RDBUF function to read a card image from the named disk file into an I/O buffer and then calls the CARDPH function to punch that image on the card punch. It repeats this process for each image in the disk file. When an end-of-file is detected during a disk read operation, OFFLINE closes the disk file (via FINIS), closes punch operations (via CLOSIO), releases the storage previously obtained (via SVCFRET), and returns to the calling program.

PUNCHCC: Prior to punching the specified card file as described above under PUNCH, a control card of the form "OFFLINE READ filename filetype" is punched preceding the normal punched output.

PUNCHOT: Prior to punching the specified card file as described under PUNCH, a control card of the form "OFFLINE READ filename filetype filemode date-last-written time-last-written" is punched preceding the normal punched output.

PRINT: OFFLINE first calls the PRINTIO function program to print a page heading and then to double space. Next, it calls the RDBUF function program to read a line image from the named disk file and the PRINTIO function program to print the line on the printer. OFFLINE repeats the process of reading an image and printing it for 55 lines. At this time, it ejects the printer to a new page, prints a page heading, double spaces, and prints the next 55 line images. When an end-of-file is detected during a disk-read operation, OFFLINE closes the disk file, closes printer operations, releases the storage previously obtained, and returns to the calling program.

PRINTCC: For PRINTCC, OFFLINE operation parallels that for PRINT, except that OFFLINE does not directly control printer facilities (for example, spacing, ejection), but rather, allows the first character of the print line image to be used for this purpose.

PRINTUPC: The function PRINT is performed after each print line image has undergone a translation on each character to uppercase representation. Each character is OR'ed with a value of X'40'.

PRINTVLR: The first four bytes of the print record are scanned for the effective length of the data record. The length is then passed to the PRINT function.

## PRINTF

FUNCTION: To print all or a specified part of a given file on the user's console typewriter.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
         LA    1,PLIST
         SVC   X'CA'
         .
         .
         .
PLIST    DC    CL8'PRINTF'
         DC    CL8'        '  filename
         DC    CL8'        '  filetype
        {DC    CL8'        '} * or starting item number
        {DC    CL8'        '} * or ending item number
        {DC    CL8'        '} print line width
```

OPERATION: PRINTF checks the filename and filetype to ensure they are both present and not asterisks. Then the STATE function program is called to verify the existence of the given file and to determine the number of items, fixed or variable filetype, etc. If STATE cannot find the file, a message 'FILE NOT FOUND' is given, with error code 3.

If an error in the parameters is detected (for example, filename or filetype omitted, or the ending item number less than the starting item number), the message "INCORRECT 'PRINTF' PARAMETER-LIST" is given, and error code 1 is returned.

Several possible file types are checked for, and certain default values are chosen for, various line lengths, as follows:

| Filetype | Line-Length |
|---|---|
| MEMO | 80 |
| SCRIPT | 120 |
| LISTING | 121 |
| Any Other | 72 |

If the starting item number is greater than the number of items in the file, the message *EOF* is printed, and return is made (without an error indication).

If the starting and/or ending item number has been supplied, PRINTF sets the RDBUF parameter list as needed to read the items desired.

PRINTF then calls RDBUF to read several items at once into a very large (3200-byte) buffer included with the program (unless the file is variable, in which case, one item at a time is read). The items are then printed online one at a time via calls to TYPLIN, with the desired or actual line-length used, until the buffer is exhausted, at which time it is refilled if necessary, etc., until printing is complete.

To enhance overall CP/CMS performance in case several users are using PRINTF at the same time (each in his own virtual machine), PRINTF is deliberately designed to take as few pages in core as possible while running, and to minimize disk reading by reading several items at once. (For example, a PRINTF of the first 40 items of a fixed file of 80-byte records would require just one call to RDBUF to fill the 3200-byte buffer, and very few actual SIO's performed by RDTK to satisfy the RDBUF call.)

## SCRIPT

FUNCTION: To print a file of English text at the terminal or on the printer.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
        LA     1,PLIST
        SVC    X'CA'
        .
        .
        .
PLIST   DC     CL8'SCRIPT'
        DC     CL8'        '  filename
        DC     CL8'        '  option 1
        .
        .
        .
        DC     CL8'        '  option N
```

OPERATION: The operation of SCRIPT consists of initialization and text processing.

INITIALIZATION: SCRIPT determines if the user has provided a filename and saves the name if it is specified. If it is not specified, SCRIPT signals the error and returns to the calling program. SCRIPT then processes the parameter list (of PARMROUT) to determine specified options. If the PAGE option was specified, PARMROUT sets an indicator (at SWON). If the user has provided a filename, SCRIPT determines if the file to be typed at the terminal exists. If it does not, SCRIPT signals the error and returns to the calling program. If the file exists, it then types the message 'SEM---VERSION 2' at the terminal and waits for the user to reply with a carriage return (unless the NOWAIT option was specified). SCRIPT then tests for the OFFLINE option and ejects a page on the offline printer if it was specified. When the reply is received, SCRIPT sets the top margin by skipping five lines. This completes initialization.

TEXT PROCESSING: When initialization is complete, SCRIPT reads, in turn, each line in the specified file until the end-of-file is encountered. It then returns to the calling program. The processing performed on each line read depends upon the nature of the first character in the line. This character may be either a nonblank character, blank, or a period.

First Character Nonblank: If the first character in the line read from the file is neither a blank nor a period, SCRIPT determines whether or not fill mode is in effect. If it is not, SCRIPT types the line, as is, at the terminal. (In this case, the line will be truncated if the length to be typed is greater than the current line length.) If fill mode is in effect, SCRIPT types the line, appropriately justified, at the terminal or on an offline printer, if the OFFLINE option was specified. (One or more right-justified lines may be typed at the terminal, depending on how the length of the line to be typed compares with the line length that is currently in effect. Also, some characters at the end of the line to be typed may remain in the output buffer and be merged with characters from the next line read to produce the next right-justified line typed at the terminal. If the TRANSLATE option was specified, the line will be printed in uppercase letters.)

First Character Blank: If the first character in the line read from the file is a blank, indicating that a new paragraph is to be started, SCRIPT types the remainder of the previous line at the terminal or on an offline printer, if the OFFLINE option was specified. It then determines whether or not fill mode is in effect. If it is not, SCRIPT types the line just read, as is, at the terminal or offline printer. If fill mode is in effect, SCRIPT types the line, appropriately justified, at the terminal or offline printer. The line is typed or printed in upper case, if the TRANSLATE option was specified.

First Character Period: If the first character in the line read from the terminal is a period, a control word is contained in the line, and control is passed to a corresponding program to carry out the required processing.

Break (. BR): SCRIPT types the remainder of the previous line at the terminal or offline printer and returns to read the next line from the file.

Page Eject (. PA): SCRIPT types the remainder of the previous line at the terminal or offline printer and skips to the bottom of the page. If the STOP option has been specified, it waits for the user to enter a carriage return. When the carriage return is received, or if the STOP option is not specified, SCRIPT skips all but two lines of the top margin, types the page heading and page number, skips one line, and returns to read the next line from the file.

Space (. SP): SCRIPT types the remainder of the previous line at the terminal or offline printer, skips n lines by typing n null lines, and returns to read the next line from the file. (If the bottom of the page is reached during skipping, a page is ejected and skipping continues. Also if double spacing is in effect, 2n lines will be skipped.)

Heading (. HE): SCRIPT saves the heading for future use and returns to read the next line from the file.

Line Length (. LL): SCRIPT types the remainder of the previous line at the terminal or offline printer, saves the specified line length for future use, and returns to read the next line from the file.

Center (. CE): SCRIPT types the remainder of the previous line at the terminal or offline printer and reads the next line from the file. This is the line to be centered. It then centers that line in an output buffer, types the line, and returns to read the next line from the file.

Page Length (. PL): SCRIPT types the remainder of the previous line at the terminal or offline printer, saves the specified page length for future use, and returns to read the next line from the file.

Top Margin (. TM): SCRIPT types the remainder of the previous line at the terminal or offline printer, saves the top margin size specified for future use, and returns to read the next line from the file.

Bottom Margin (.BM): SCRIPT types the remainder of the previous line at the terminal or offline printer, saves the bottom margin size specified for future use, and returns to read the next line from the file.

Fill (.FI): SCRIPT sets a switch to indicate that fill mode is in effect, types the remainder of the previous line at the terminal or offline printer, and returns to read the next line from the file.

No Fill (.NF): SCRIPT sets a switch to indicate that "no-fill" mode is in effect, types the remainder of the previous line at the terminal or offline printer, and returns to read the next line from the file.

For details on the philosophy of the SCRIPT System and certain algorithms used, see "SCRIPT: An Online Manuscript Processing System", Form 320-2023 from the IBM Cambridge Scientific Center, Cambridge, Massachusetts.

## SPLIT

FUNCTION: To copy a portion of the one file into another.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
         LA    1,PLIST
         SVC   X'CA'
         .
         .
         .
PLIST    DC    CL8'SPLIT'
         DC    CL8'        '   filename1
         DC    CL8'        '   filetype1
         DC    CL8'        '   filename2
         DC    CL8'        '   filetype2
         DC    CL8'        '   1st delimiter
         DC    CL8'        '   2nd delimiter (optional)
```

OPERATION: SPLIT first performs a series of tests to ensure that the parameter list is valid. If it is not valid, it signals the error and returns to the calling program. If the parameter list is valid and the first and second delimiters are numeric, SPLIT stores the first delimiter in the calling sequence to RDBUF. This causes RDBUF to start reading from the appropriate place in the file being copied from. It also saves the second delimiter. SPLIT then reads the specified number of records from the file being copied from and transmits them to the file that is to receive them. Next, SPLIT types the message 'FILE MODIFIED' at the terminal. It then calls the FINIS command program to close both files and returns to the calling program.

If the first delimiter is numeric and the second is symbolic, SPLIT stores the first delimiter in the calling sequence to RDBUF. It then reads the first record to be copied and determines if the label field of that record matches the symbolic second delimiter. If it does not, SPLIT writes that record into the file that is to receive it and reads the next record from the file being copied from. SPLIT repeats this process until a match of label field and symbolic delimiter occurs. At this time, it types the message 'FILE MODIFIED' on the terminal, closes both files, and returns to the calling program.

If the first delimiter is symbolic and the second is numeric, SPLIT reads successive records from the file being copied from until it encounters the one containing a label field that matches the symbolic delimiter. At this point, SPLIT transfers the specified number of records from the file being copied from into the file that is to receive them. It then types the message 'FILE MODIFIED' at the terminal, closes both files, and returns to the calling program.

If both delimiters are symbolic, SPLIT reads successive records from the file being copied from until it encounters the one containing a label field that matches the symbolic first delimiter. It then transfers that record to the file that is to receive it and reads the next record from the file being copied from. If this record does not contain a label field that matches the symbolic second delimiter, SPLIT writes it into the file that is to receive it and reads the next record from the file being copied from. SPLIT repeats this procedure until it encounters the record containing the label field that matches the symbolic second delimiter. At this time, all specified records have been copied, and SPLIT types the message 'FILE MODIFIED' at the terminal, closes both files, and returns to the calling program.

Note: If the second delimiter is not given, SPLIT copies records until the end-of-file is reached.

## STATE

FUNCTION: To determine if a given file exists on P-Disk, T-Disk, S-Disk, or any other available read-write or read-only disk.

ATTRIBUTES: Nucleus Resident

CALLING SEQUENCE:

```
        DC    CL8'        '    filename
        DC    CL8'        '    filetype
        DC    CL2'        '    filemode
```

OPERATION: STATE (when entered from the terminal, or as part of an EXEC file) is identical to the STATE function program, except that from the terminal (or as part of an EXEC procedure) it is used only to determine if a file exists on the specified (or any, if '*' was given) disk.

If a mode-letter was specified, only the FST hyperblocks for the disk specified for that letter (for example, PSTAT for P-Disk) are searched for the given file. (If a file is on a read-only extension of a disk — for example, an A-Disk as an extension of a P-Disk — it will be found if either mode-letter — A or P in the example — is specified.)

If the filemode is * (or omitted entirely), all logged-in read-write and read-only disk(s) will be searched (if necessary) to find the file.

If the file is not found, an error code 1 (with no message) is returned.

See also STATEW command.

## STATEW

FUNCTION: To determine if a given file exists on P-Disk, T-Disk, or any other available read-write disk.

ATTRIBUTES: Nucleus resident

CALLING SEQUENCE:

```
        DC    CL8'      '    filename
        DC    CL8'      '    filetype
        DC    CL2'      '    filemode
```

OPERATION: STATEW (when entered from the terminal, or as part of an EXEC file) is identical to the STATEW function program, except that from the terminal (or as part of an EXEC procedure) it is used only to determine if a file exists on the specified ( or any, if '*' was given) read-write disk.

If a mode-letter was specified, only the FST hyperblocks for the disk specified for that letter (for example, PSTAT for P-Disk) are searched for the given file.

If the filemode is * (or omitted entirely), all logged-in read-write disk(s) will be searched (if necessary) to find the file.

If the file is not found, an error code 1 (with no message) is returned.

STATEW is similar to the STATE command (see description), except that only read-write disk(s) are searched (read-only disks being ignored).

## UPDATE

FUNCTION: To resequence, insert, replace, or delete records on a file.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
        LA      1,PLIST
        SVC     X'CA'
        .
        .
        .
PLIST   DC      CL8'UPDATE'
        DC      CL8'        '   filename 1
        DC      CL8'        '   filetype 1
        DC      CL8'        '   filename 2
        DC      CL8'        '   filetype 2
        DC      CL8'('          separator for option
        DC      CL8'        '   options
```

OPERATION: UPDATE first scans the parameter list for errors. If an error exists, a message is generated and UPDATE returns to the caller. If the parameter list is valid, it enters the specified filenames and filetypes in the SYSIN and UPDATE file control blocks. UPDATE then determines if an intermediate file (INTER), containing changes from a previous update, exists. If an intermediate file does exist, a message is generated and UPDATE returns to the caller; the user either erases it or combines it with the original file and reissues the UPDATE command. If an intermediate file does not exist, the update log (UPDLOG) is erased. During update file processing, a record of control cards in the update file (UPDATE), items added to or deleted from the original file, and error messages are stored in the UPDLOG file.

Cards are then read from the UPDATE file. When a control card is read (identified by a '//' or './' in columns 1 and 2), the FORMAT routine checks to see that it is a valid card, saves sequence numbers, and checks for valid numerics. When a sequence (//S or ./S) control card is read, the file is sequenced on columns 76 through 80 of each card image. When a delete (//D or ./D) control card is read, the DELETE routine finds the specified sequence numbers on the SYSIN file, writes the associated cards into the INTER file and UPDLOG file, and deletes them from the SYSIN file. When an insert (//I or ./I) control card is read, the INSERT routine does the following: finds the specified sequence numbers in the SYSIN file, resequencing the file (via the RESEQ routine) if necessary; writes SYSIN into the INTER file; inserts the cards into the INTER file; and writes the inserted cards into the UPDLOG file. When a replace (//R or ./R) control card is read, the REPLACE routine performs a Delete and Insert operation.

There are three possible options to UPDATE. SEQ8 specifies that sequencing is to be done on all eight characters in columns 73 to 80, rather than the default of five characters. INC specifies that the sequence number in the update card is to be placed in the updated deck rather than the default of eight asterisks to distinguish updated cards. P specifies that the original file is to be erased and the updated file altered to its filename and filetype rather than the default which retains the original file and alters the updated file to a filename of . plus the first seven characters of the original filename.

## EXECUTION CONTROL

The commands that control the execution of programs under CMS are EXEC, GENMOD, GLOBAL, LOADMOD, REUSE, START, USE, and $. These are described in the following section.

### EXEC

FUNCTION: To execute the commands stored in a specified file of filetype EXEC.

ATTRIBUTES: Nucleus resident

CALLING SEQUENCE:

```
        LA      1, PLIST
        SVC     X'CA'
        .
        .
        .
PLIST   DC      CL8'EXEC'
        DC      CL8'        '  filename
        DC      CL8'        '  argument1
        DC      CL8'           argument2
        .
        .
        .
        DC      CL8'        '  argumentN
```

OPERATION: The EXEC command proper is a short nucleus-resident bootstrap program that (when entered initially) obtains main storage using the FREE function program, then RDBUF's the main (much larger) disk-resident portion of EXEC (called "EXECTOR MODULE") into that free storage. (When the last EXEC call has been completed, indicated by a level-counter returning to zero, that free storage is returned via FRET.) This procedure makes it unnecessary to keep the entire EXEC code in the CMS nucleus at all times.

When the EXECTOR MODULE has been loaded, orif it is already in core, it calls FREE as needed for working storage. It then determines via the STATE function program whether the specified file exists. If the file does not exist, EXECTOR calls the TYPLIN function program to type a message to that effect at the terminal, calls the FRET function program to release the previously obtained storage, and returns (via the EXEC bootstrap program) to the calling program, which is usually INIT.

If the specified file exists, EXECTOR saves the arguments that are to replace the dummy arguments in the commands. Next, it reads the first command to be executed from the specified file. Then, EXECTOR calls the SCAN function program to place the command to be executed into parameter list format. Subsequently, EXECTOR replaces the dummy arguments in the parameter list for the command to be executed with the substitutes provided for them in the EXEC command. Next, as a rule EXECTOR calls the TYPLIN function program to type the command to be executed at the terminal. It then executes the command by issuing a SVC X'CA' with register 1 pointing to the parameter list for the command. When execution of the command is complete, EXECTOR reads the next command from the file and executes it similarly. When the last command has been executed (i. e. , when an end-of-file is encountered), EXECTOR calls the FINIS command program to close the file, calls the FRET function program to release the storage previously obtained, and returns via the EXEC bootstrap program to the calling program.

EXECTOR, in addition to processing CMS commands in an EXEC file, also handles several exec command words, which are not CMS commands at all, but directions to EXEC as to how and/or where to proceed if errors occur in execution of the various commands, typing or non-typing of the commands, etc.

Note: To ensure maximum possible file integrity, and to be compatible with INIT in its running of CMS commands, EXECTOR calls LOGDISK to update the user file directly after each execution of a CMS command in an EXEC file.

## GENMOD

FUNCTION: To create a file in nonrelocatable core-image form on the user's P-Disk.

ATTRIBUTES: Nucleus resident

CALLING SEQUENCE:

```
        LA      R1,PLIST        R1 must point to P-List as usual
        SVC     X'CA'
        DC      AL4(ERROR)
```

ENTRY REQUIREMENTS:

```
        R1 must point to GENMOD parameter-list:
        DS      0F
PLIST   DC      CL8'GENMOD'
        DC      CL8'        '  entry 1 (= filename of module)
       [DC      CL8'        '] entry 2
       [DC      CL8'(       '] additional options, namely
                               NO and/or P2 (preceded by left-paren)
       [DC      CL8'        ']
```

EXIT CONDITIONS:

Normal Return
R15 = 0
Loader Tables empty, or Entry 1 or Entry 2 not found
R15 = 1 (Error 1)
Error Writing Module
R15 = value returned by WRBUF or FINIS

CALLS TO OTHER ROUTINES:

START, ERASE, WRBUF, FINIS

CALLED BY (where known):

User, and by various EXEC procedures which generate modules.

MACROS USED:

FVS

OPERATION: The GENMOD program obtains the entry addresses specified in the GENMOD command. If a second entry is not specified in the command, it uses the pointer LOCCNT (established by LOAD, LOADMOD, USE, or REUSE) to the next available load location. GENMOD also calls "START (NO)" to resolve any establishment of common storage, undefined names, etc., as left by the loader.

Before creating the new module on disk, GENMOD erases any old module on the P-Disk that may exist with the same name.

If the P option was specified in the caller's parameter list, the module is created with a mode of P2. Otherwise, a mode of P1 is used.

The new module is then created on the P-Disk with appropriate calls to WRBUF, and then FINIS, to close the file.

The module created by GENMOD (to be readable by LOADMOD) is a variable file consisting of two or more records (the last may be omitted) as follows:

1. A 44-byte record containing vital information from NUCON table, length of second record, indicator of presence or absence of last record, etc.

2. A Core-image of program from entry 1 to entry 2 (or LOCCNT).

3. Loader-tables. (Omitted for transient disk-resident routines, or if the (NO) option was specified in GENMOD parameter list.)

If the entry 1 or entry 2 (if present) is not found in the loader tables, an error message is printed of the following form:

NO "XXXXXXXX" MODULE

and an error code 1 is returned (no module is written).

Notes:

1. GENMOD itself is called only via SVC, but when calling ERASE, WRBUF, and FINIS, GENMOD calls them via BALR, for maximum speed.

2. As a debugging aid, like LOADMOD, GENMOD leaves the following meaningful information in registers 1 — 4 upon exit (which can be displayed by running with SETOVER GPRS or with a suitable breakpoint using DEBUG):

   R1: Starting Address of GENMOD'ed Region

   R2: Ending Address of GENMOD'ed Region

   R3: Starting Address of Loader Tables (if written)
   (R3 meaningless if R4 = 0)

   R4: 00, or Size in Bytes of Loader Tables (if written)

3. See Figure 29 for details on the content of a CMS "MODULE" file.

4. The loader tables (if written) include the entire loader tables as in existence at the time of the GENMOD call (not just the entry points included between entry 1 and entry 2, or between entry 1 and the value of LOCCNT).

5. LOADMOD is also included with the GENMOD program.

1. First Record (44 bytes):

| Bytes | Number of Bytes | NUCON VALUES or other quantity | DETAILED BREAKDOWN Bytes | NUCON VALUE |
|---|---|---|---|---|
| 0-17 | 18 | USFL thru first 2 bytes of TBLNG | 0- 3 | USFL |
| | | | 4- 7 | USLL |
| | | | 8-11 | STADDR |
| | | | 12-15 | LDRTBL |
| | | | 16-17 | TBLNG (first two bytes) |
| 18-19 | 2 | Last 2 bytes of TBLNG, or 0 if loader tables omitted from module | | |
| 20-23 | 4 | 0 if loader tables omitted from module; nonzero if loader tables present in module | | |
| 24-35 | 12 | LOCCNT thru first 4 bytes of "PSW" | 24-27 | LOCCNT |
| | | | 28-31 | LDADDR |
| | | | 32-35 | "PSW" (first four bytes) |
| 36-39 | 4 | Starting Address of Loaded Region | | |
| 40-43 | 4 | Ending Address of Loaded Region | | |

2. Next record(s) (maximum of 65535 bytes)

Core image to be LOADMOD'd (broken into 65535-byte records if necessary)

3. Last record (if present)

Loader Tables (multiple of 16 bytes)

Figure 29. Contents of a CMS MODULE file (variable records)

## GLOBAL

FUNCTION: To allow the user to specify alternate MACRO libraries to be searched during assembly and alternate TXTLIB libraries to be searched during loading.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
        LA     1,PLIST
        SVC    X'CA'
        .
        .
        .
PLIST   DC     CL8'GLOBAL'
                   ( MACLIB )
               CL8'{ TXTLIB }'
                   ( PRINT  )
        DC     CL8'        '     libnamel
        .
        .
        .
        DC     CL8'        '     libnameN
```

OPERATION: GLOBAL first determines whether MACLIB, TXTLIB, or PRINT was specified. If neither was specified, it signals the error (code 1) and returns to the calling program. If MACLIB was specified, GLOBAL moves the specified library names into the macro library list (MACLIBL) and returns to the calling program. This list will be referred to by the routine that simulates the FIND macro instruction. (The assembler issues a FIND when it attempts to locate a particular macro instruction in the specified macro libraries.)

Note: If the user does not supply any library names with a GLOBAL MACLIB command, no MACLIB file will appear in the macro library list.

If TXTLIB was specified, GLOBAL moves the specified library names into the text library list (TXTLIBS) and returns to the calling program. This list is referred to when the loader is searching for subroutines to resolve cross references.

Note: If the user does not supply any library names with a GLOBAL TXTLIB command, no TXTLIB file will appear in the text library list.

If PRINT was specified GLOBAL prints at the terminal a list of the macro and text libraries currently being searched.

The existence of each libname specified will be verified by a call to STATE. If the file "libname MACLIB" or "libname TXTLIB" is not found, it is not included in the respective library list, and GLOBAL will issue a message (error 3) and return. Also, if more than five (5) macro libnames or more than eight (8) text libnames are specified, an error message is typed (error code 2) and a return is executed.

## LOAD

FUNCTION: To read specified programs from disk into core, establish proper linkages, and initiate execution when specified.

ATTRIBUTES: Nucleus resident

CALLING SEQUENCE:

```
       LA    1,PLIST
       SVC   X'CA'
       .
       .
       .
PLIST  DC    CL8'LOAD'
       DC    CL8'(      '  filename1
       .
       .
       .
       DC    CL8'       '  filenameN
       DC    CL8'(      '  separator for options
       DC    CL8'       '  option1
       .
       .
       .
       DC    CL8'       '  optionN
       .
       .
       .
       DC    CL8'       '  libname1
       .
       .
       .
       DC    CL8'       '  libnameN
```

OPERATION: When the command scanner detects a LOAD command, it gives control to the CMS loader (LDR). The CMS loader will load the program at location 12000 (unless an SLC address was specified), search specified libraries for missing subroutines, and establish proper linkages. The operation of the loader is described in Section 5 under the heading "CMS Loaders".

If the SLC option is used, it must not immediately follow a left-parenthesis, resulting in nine consecutive nonblank characters. A blank in between the left parenthesis and the SLC will solve this problem.

Examples:

```
No Good:  LOAD SOMEPROG (SLC13000
OK:       LOAD SOMEPROG ( SLC13000
```

If the program is to be loaded into the transient area, the old module (if any) is to be erased first, then the program loaded with a TRANS option. For example, if a new version of STAT were to be loaded and GENMOD'ed, the sequence might be as follows:

```
ERASE STAT MODULE P
LOAD STAT (TRANS TYPE)
GENMOD STAT
```

## LOADMOD

FUNCTION: To load a nonrelocatable core image file into core.

CALLING SEQUENCE:

```
        LA   R1, PLIST       R1 must point to P-List as usual
        SVC  X'CA'
        DC   AL4(ERROR)
```

ENTRY REQUIREMENTS:

```
        R1 must point to LOADMOD parameter list:
        DS   0F
PLIST   DC   CL8'LOADMOD'
        DC   CL8'        '  filename of module
        DC   CL2'  '        mode (optional)
        DC   X'FF000000'    delimiter if mode omitted
           or
        DC   X'FFFFFFFF'    (see OPERATION)
```

EXIT CONDITIONS:

Normal Return
R15 = 0
Module Not Found (Error 1)
R15 = 1
Module will not fit in core (Ending Address higher than LOWEXT)
R15 = 8 (Error 8)
Error Reading Module
R15 = value returned by RDBUF

CALLS TO OTHER ROUTINES:

STATE, RDBUF, FINIS

CALLED BY (where known):

LINKAGE, SVCINT, $ Command

MACROS USED:

FVS

OPERATION: The LOADMOD program checks to ensure that a module of the given filename exists; then it reads that module into the locations at which it had been generated. If the module was brought into the TRANSAR transient area, the name of the module is stored in the appropriate place (TRANSRT) for SVCINT, and reading is terminated. Otherwise, the starting address of the module is placed in STADDR within the NUCON table, and other pertinent information (but not including actual core size) is stored in the NUCON table. If the module was generated using the (NO) option for GENMOD, reading is terminated. Otherwise, the loader tables are read into high-numbered core, and are thus restored to their value at the time the module is generated. Note that the loader tables are restored in high-numbered core depending on core-size at LOADMOD time, not at the time the module was generated. Thus, for example, a module generated on a 64-page (40000 hex) machine would have its loader tables ending just prior to location 40000 hex; if this module were LOADMOD'ed into a 128-page machine, the loader tables would be stored just prior to location 80000 hex. This feature of LOADMOD makes it possible to generate modules on, say, a standard size machine of 64 pages and to loadmod these modules on any machine big enough to run the programs (either smaller or larger than the standard machine).

If LOADMOD cannot find the specified module, an error message is normally printed of the following form:

NO "XXXXXXXX" MODULE

and an error code 1 is returned.

(If the delimiter in the parameter list was X'FF000000', as it is when LOADMOD is called from SVCINT to load a module that may or may not exist, the above typeout is omitted.)

Before reading the core-image file, LOADMOD checks to ensure that it will fit in core as of the moment, that is, that it will not overlap live data in high-numbered core as given by the LOWEXT value in NUCON. If an overlap would occur, the core-image is not read in, and an error 8 from LOADMOD is given, with the error message mentioned below.

If a read error from RDBUF occurs, an error message is printed, for example, BAD "XXXXXXXX" MODULE (with the name filled in), and the error-code obtained from RDBUF is returned.

Notes:

1. LOADMOD itself is called only via SVC, but when calling STATE, RDBUF, and FINIS, LOADMOD calls them via BALR, for maximum speed.

2. As a debugging aid, LOADMOD leaves the following meaningful information in registers 1-4 upon exit (this information can be displayed by running with SETOVER GPRS or with a suitable breakpoint using DEBUG):

   R1: Starting Address of Loaded Region
   R2: Ending Address of Loaded Region
   R3: Starting Address of Loader Tables (if any)
   (R3 meaningless if R4 = 0)
   R4: 00, or Size in Bytes of Loader Tables (if any)

3. See Figure 29 for details on the content of a CMS "MODULE" file.

4. LOADMOD is an entry point included in the GENMOD program. If a user abbreviation should be set up for LOADMOD using the SYN command, it must begin with the letter L (for example, LQ for load quickly).

## USE — REUSE

FUNCTION: To load programs into core and establish linkages with previously-loaded programs.

ATTRIBUTES: Nucleus resident

CALLING SEQUENCE:

```
       LA   1,PLIST
       SVC  X'CA'
       .
       .
       .         ( USE  )
PLIST  DC   CL8' (REUSE) '
       DC   CL8'          ' filename1
       .
       .
       .
       DC   CL8'          ' filenameN
       DC   CL8'  ('          separator for options
       DC   CL8'          ' option 1
       .
       .
       .
       DC   CL8'          ' optionN
       DC   CL8'          ' libname1
       .
       .
       .
       DC   CL8'          ' libnameN
```

OPERATION: USE picks up the address of the nucleus constant area (NUCON) and passes control to the loader at entry point OVRLD; the program is then loaded at the next high core location above the point at which the last load was terminated.

If REUSE was specified, USE zeroes STADDR (the address at which execution of the user program is to start) before it passes control to the loader; the default entry point will be the entry point of the first file specified with REUSE.

After loading, USE determines if there were any errors, saves the error code in register 15, and returns to the calling program.

## START

FUNCTION: To set undefined symbols to zero, define COMMON, then determine where to start execution of a program that has been loaded into core, and transfer control.

ATTRIBUTES: Nucleus resident

CALLING SEQUENCE:

```
         LA    1,PLIST
         SVC   X'CA'
         .
         .
         .
PLIST    DC    CL8'START'_
        ┌DC    CL8'     '  ┐  entry point
        │DC    CL8'     '  │  argument 1
        │.                 │
        │.                 │
        │.                 │
        └DC    CL8'     '  ┘  argument N
```

OPERATION: START obtains the parameter list and the return address and links to the start execution routine (XEQQ) in the loader to define any undefined symbols and determine the start execution address. (Refer to "Start Execution Routine", in the CMS Loader description.) A message is returned to the user if any error is encountered. Register 15 is set to the entry address. START does not operate on arguments specified on the START statement; they are passed to the user's program via a pointer in general purpose register 1. If an asterisk, '*' is specified as the first argument, control is passed to the location specified at STADDR in NUCON.

When control is passed to the program being started, the message "EXECUTION BEGINS. . ." is typed (without waiting for completion). Also, a zero-filled 18-word save area (STRTSAV) is provided, with its address in register 13.

If a call to START is made when nothing has been loaded, an error of code will be returned to the caller. This may also occur if a program which was GENMOD'ed with no loader tables (using the (NO) option of GENMOD) is LOADMOD'ed and then START'ed.

A special call START (NO) causes the loading process to be completed without actually transferring to a program. Thus undefined names are handled, common (if any) is assigned, undefined names are handled, free storage used by the loader returned, etc.

This call, in particular, is used by GENMOD.

## $

FUNCTION: To execute a file of filetype EXEC, MODULE, or TEXT.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
         LA    1,PLIST
         SVC   X'CA'
         .
         .
         .
PLIST    DC    CL8'$'
         DC    CL8'        '    filename
         DC    CL8'        '    argument 1
         DC    CL8'        '    argument 2
         .
         .
         .
         DC    CL8'        '    argument-n
```

OPERATION: $ calls the FREE function program to obtain a block of storage for use as a work area. Next, it calls the STATE function program to determine if a file designated as 'filename EXEC' exists. If it does, $ proceeds as described below. If it does not, $ calls STATE to determine whether a file designated as "filename MODULE" exists. If it does, $ proceeds as described below. If it does not, $ again calls STATE to determine whether a file designated as "filename TEXT" exists. If it does, $ proceeds as described below. If it does not, the desired file does not exist, and $ signals this by terminal message (using the TYPLIN function program), releases the storage previously obtained (using the FRET function program), sets a code to indicate the error, and returns (via SVCINT) to the calling program, which is usually INIT.

EXEC FILE EXISTS: If a file designated as "filename EXEC" exists, $ releases the storage previously obtained and passes control to the EXEC command program that will process the list of commands. EXEC will return control (via SVCINT) to the program that called $.

MODULE FILE EXISTS: If a file designated as "filename MODULE" exists, $ calls the LOADMOD command program to load the core image file into main storage. Next, it calls the FRET function program to release the storage previously obtained. Finally, $ passes control to the START command program, which will begin execution of the program. START will return control (via SVCINT) to the program that called $.

Note: No module that was loaded at TRANSAR before being GENMOD'ed, or was GENMOD'ed with the (NO) option should be initiated by the $ command, as START will be unable to find the entry point. This, however, is not a problem, because programs of this type should be invoked directly via an SVC call.

TEXT FILE EXISTS: If a file designated as "filename TEXT" exists, $ calls the LOAD command program to load the relocatable module into main storage and to resolve any external references. Next, it releases (using FRET) the storage previously obtained. Finally, $ passes control to the START command program to begin execution of the file (that is, the program). START will return control (via SVCINT) to the program that called $.

Note: In the case of an EXEC file, the arguments are passed to the EXEC command program for processing. General purpose register 1 is used for this purpose. In the case of a MODULE or TEXT file, the arguments are passed to the loaded program. General purpose register 1 is again used for this purpose.

## DEBUGGING COMMAND PROGRAMS

The debugging command programs allow the user to access and modify registers and core storage from his terminal. They may also be called directly by a user or CMS program.

### CLROVER

FUNCTION: To negate normal and error overriding activity.

CALLING SEQUENCE:

```
        LA     1, PLIST
        SVC    X'CA'
        .
        .
        .
PLIST   DC     CL8'CLROVER'
```

OPERATION: SVCINT passes control to OVERNUC, the core resident part of the OVERRIDE program. OVERNUC then branches to code for SETOVER, SETERR, or CLROVER as determined by the command that was in the parameter list. A flag called STATUS is set to O when the OVERRIDE module is not in core, and is set to I when the OVERRIDE module is in core.

CLROVER returns immediately to the calling program (via SVCINT) if the OVERRIDE module is not in core. If the OVERRIDE module is in core, CLROVER sets an offset for dispatching in SETLOC and branches to the OVERRIDE module.

OVERRIDE does some entry initialization and then dispatches to code for the CLROVER, SETERR, or SETOVER command as determined by the offset in SETLOC.

The code for CLROVER in the OVERRIDE module calls the STNOV function program to set the normal override switch (NRMOVR) in SVCINT off (that is, to zero), calls the .STEROV function program to set the error override switch (ERROVR) in SVCINT off, calls the PRINTIO function program to print a message to the effect that normal and error overriding have been cancelled, performs a CLOSIO to the printer, and returns to WATCHDOG in the nucleus.

OVERNUC tests for CLROVER (as determined by SETLOC) and if that function is being performed, resets the STATUS flag to O (OVERRIDE not in core), calls FRET to release the core required for the OVERRIDE module, and then returns (via SVCINT) to the calling program, which is usually INIT.

## DEBUG

FUNCTION: To enable the user to debug his program from the terminal.

ATTRIBUTES: Nucleus resident

CALLING SEQUENCE:

```
      LA    1, PLIST
      SVC   X'CA'
      .
      .
      .
PLIST DC    CL8'DEBUG'
```

OPERATION: The discussion of the DEBUG command program will be divided into two parts: processing on entry, and request environment processing.

PROCESSING ON ENTRY: The processing performed by the DEBUG command program when it receives control depends on whether it was given control because of either a DEBUG command entered from the terminal, an external interruption, or a program interruption.

DEBUG Command: DEBUG saves the contents of the general purpose registers and saves the CSW and CAW. It then types the message 'DEBUG ENTERED' at the terminal and enters the request environment.

External Interruption: DEBUG saves the external old PSW, saves the contents of the general purpose registers, and saves the CSW and CAW. It then types the message 'DEBUG ENTERED' at the terminal. Next, it types the message 'ĖXTERNAL INT.' at the terminal and enters the request environment.

Program Interruption: DEBUG saves the program old PSW, saves the contents of the general purpose registers, and saves the CSW and CAW. It then determines if the program interruption occurred at a break point. (If the address of the instruction that caused the interruption matches the address in an entry in the break point table (refer to "Break Request", later in this section, the interruption occurred at a break point.) If not, DEBUG types the message 'PROGRAM INT.' at the terminal and enters the request environment. If the program interruption occurred at a breakpoint, DEBUG moves the absolute address of the breakpoint to the last three bytes of the saved PSW and restores the operation of the instruction located at the break point. It then types the message 'BREAKPOINT XX AT YYYYYY' (where XX is the breakpoint number and YYYYYY the core-address of the breakpoint reached) and enters the request environment.

On any entry, DEBUG will save lowcore locations 0-256; that is, a dump of low core will reflect their value at the time of entering DEBUG.

REQUEST ENVIRONMENT PROCESSING: When this environment is entered, the user is given the opportunity to make DEBUG requests from the terminal. For each such request, DEBUG determines its nature through a table-lookup procedure and passes control to a corresponding program to implement the request. When the execution of that program is complete, it returns control to the control element, which obtains the next request. This request is processed similarly.

Addressing: An address may be specified two ways: (1) as a symbolic address if previously defined, (2) as a hexadecimal constant. The current value of the origin will be added to the address if it was hexadecimal.

Origin Request: DEBUG converts the origin value supplied on the request to binary, saves it for future use, and returns for the next request. The origin value may be a symbolic address or a hexadecimal address. The previous origin value is not added into the hexadecimal address.

Define Request: DEBUG converts the hexadecimal address to binary, makes the address absolute by adding the current origin value to it, and stores the resultant absolute address in the temporary symbol table (TSYM). It then retrieves the symbol being defined and places it into the temporary symbol table. Next, DEBUG retrieves the length value for the symbol (if any) supplied on the request and places it into the temporary symbol table. (If a length value is not provided, a default value of four is assumed.) Finally, DEBUG moves the contents of the temporary symbol table into the next available entry in the defined symbol table (SYMTBG) and returns for the next request.

Examine (X) Request: DEBUG uses the address specified to determine the locations to be examined (see the foregoing description under "Addressing"). If the length is specified, that is used; otherwise, the length is obtained from the symbol table if the address was symbolic or is assumed to be the default value of four if the address was hexadecimal. Finally, DEBUG moves the number of bytes specified by the length starting from the location of the first byte to an output buffer, types them at the terminal, and returns for the next request.

Break Request: DEBUG uses the address specified to determine the breakpoint location (see "addressing"). This address is stored in the break point table entry corresponding to the break point number supplied with the request. DEBUG saves the operation code (the first byte) located at the break point, replaces the operation code located at the break point with an invalid operation code, and returns for the next request. (When the invalid operation code is encountered during execution of the program containing the break point, a program interruption occurs and control is passed to DEBUG, which types a message at the terminal indicating that the break point has been reached.)

Store Request: DEBUG uses the address specified to determine the absolute location where the data is to be stored (see "Addressing"). It then converts the data to be stored to binary, moves it to the absolute core locations, and returns for the next request.

Change to Debug

Dump Request: Debug determines from the command line the absolute limits of the main storage to be dumped and places the appropriate values into the DUMPLIST plist. Note that the DUMPLIST plist can be located in the routine GENSECT. Also placed into the plist area are the addresses of the general register save area, the floating-point register save area, and the address of a low core save area. Description of the plist and its use can be found in the routine DEBDUMP, which is the dump executioner. DEBUG then BALR's to the DEBDUMP routine, the dump is executed and the next command may be issued.

Set Request: If the PSW is to be set, DEBUG converts the data to binary, overlays the PSW it saved on entry with the converted data, and returns for the next request. DEBUG sets the CSW, CAW, and the contents of the specified register in a similar manner.

PSW Request: DEBUG moves the PSW it saved on entry to an output buffer, types it at the terminal, and returns for the next request. (The PSW saved by DEBUG on entry may have been modified by a SET command.)

CSW Request: DEBUG moves the CSW it saved on entry to an output buffer, types it at the terminal, and returns for the next request. (The CSW saved by DEBUG on entry may have been modified by a SET request.)

CAW Request: DEBUG moves the CAW it saved on entry to an output buffer, types it at the terminal, and returns for the next request. (The CAW saved by DEBUG on entry may have been modified by a SET request.)

GPR Request: DEBUG determines the first register specified. It then moves the contents of that register (saved upon entry) to an output buffer and types it at the terminal. DEBUG repeats this process for each register to be considered. It then returns for the next request. (The contents of the registers saved by DEBUG on entry may have been modified by a SET request.)

Go Request: If an address was specified (see addressing), its absolute value is stored into the saved PSW. DEBUG restores the CSW and CAW it saved on entry to their corresponding locations in lower main storage, restores the registers with the contents it saved on entry, and loads the PSW it saved on entry. (The contents of the registers and the CSW and CAW saved by DEBUG on entry may have been modified by a SET request.) If the GO address was not specified, loading of the PSW causes control to be returned to the interrupted program at the point of interruption, or passed to the location specified if the user modified the address portion of the PSW with a SET request.

Return Request: DEBUG restores the registers with the contents it saved on entry, clears register 15, and branches unconditionally through register 14. (The contents of the registers saved by DEBUG on entry may have been modified by a SET request.) Return is valid only if DEBUG was entered via the DEBUG command.

Restart Request: RESTART is equivalent to the IPL request, discussed below. See IPL.

IPL Request: If the IPL request is given, DEBUG branches directly to the CMS IPL command, which brings in a fresh copy of CMS from the system disk.

KX Request: If KX is given as a request to DEBUG, the kill execution logic is invoked as if KX had been entered via the CMS command environment. The kill execution program closes all open files and updates the user file directory before going to the CP environment. Thus, if DEBUG is reached and the user wishes his files to be closed and updated at their current status, he can issue the KX request. The KX request causes a fresh copy of CMS to be IPL'ed from the system disk.

## SETERR

FUNCTION: To activate error overriding facilities.

CALLING SEQUENCE:

```
        LA      1, PLIST
        SVC     X'CA'
        .
        .
        .
PLIST   DC      CL8'SETERR'
```

OPERATION: SVCINT passes control to OVERNUC, the core resident part of the OVERRIDE program. OVERNUC then branches to the code for the SETERR command. SETERR (within the JOINT subroutine in OVERNUC) tests for the OVERRIDE module loaded into core (STATUS set to I). If the OVERRIDE module is not in core, FREE is called to obtain free storage, and then the OVERRIDE module is copied into core and STATUS is set to I. Within the OVERRIDE module SETERR calls the .STEROV function program to set the error override switch on by placing the address of the error override handling program (HNDLERR) into the ERROVR field within SVCINT, calls the PRINTR function program to print a message to the effect that error overriding has been activated, and returns (via OVERNUC and then via SVCINT) to the calling program, which is usually INIT.

## SETOVER

FUNCTION: To activate normal and error overriding facilities.

CALLING SEQUENCE:

```
        LA      1, PLIST
        SVC     X'CA'
        .
        .
        .
PLIST   DC      CL8'SETOVER'
       [DC      CL8'SAMELAST'
       ┌              ┌ GPRS      ┐       ┐
       │DC      CL8'  │ GPRSB     │   '   │
       └              └ GPRSA     ┘       ┘
       ┌              ┌ FPRS      ┐       ┐
       │DC      CL8'  │ FPRSB     │   '   │
       └              └ FPRSA     ┘       ┘
       ┌              ┌ NOPARM    ┐   '   ┐
       │DC      CL8'  └ PARMI     ┘       │
       └                                  ┘
       ┌              ┌ NOWAIT    ┐       ┐
       │DC      CL8'  │ WAITSAME  │   '   │
       │              │ WAIT 1    │       │
       └              └ WAIT 2    ┘       ┘
       [DC      CL8'  DEFAULT'            ]
```

OPERATION: SVCINT passes control to OVERNUC the core resident part of the OVERRIDE program which causes the OVERRIDE module to be read into free storage if it is not already in core. SETOVER checks for the presence of the SAMELAST option. If provided, it scans the remainder of the parameter list and sets switches to indicate the options specified therein, calls the PRINTIO function program to print a message to the effect that both normal and error overrides are in effect, calls the .STEROV function program to set the error override switch on by placing the address of the error override handling program (HNDLERR) into the ERROVR field within SVCINT, calls the .STNOV routine to set the normal override switch on by placing the address of the normal override handling program (HNDLNRM) into the NRMOVR field within SVCINT, and returns (via OVERNUC and then via SVCINT) to the calling program, which is usually INIT.

If the SAMELAST option is not present, SETOVER sets all option switches to their default values and proceeds as described above from the point where the parameter list is scanned.

(If, during the scan, SETOVER encounters a DEFAULT parameter, it sets all option switches to their default values and processes the remainder of the parameter list in the normal fashion.)

## LANGUAGE PROCESSING COMMAND PROGRAMS

The language processing command programs perform all required initialization functions in preparation for language compilations, and transfer control to the appropriate compiler. Languages supported by CMS include: ASSEMBLER F, PL/I F, and FORTRAN IV G. Two additional language processors are available as Type III programs: SNOBOL and BRUIN.

### ASSEMBLE

FUNCTION: To assemble one or more files.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
      LA      1, PLIST
      SVC     X'CA'
      .
      .
      .
PLIST DC      CL8'ASSEMBLE'
      DC      CL8'        '   filename1
      .
      .
      .
      DC      CL8'        '   filenameN
      DC      CL8'(       '   separator for options
      DC      CL8'        '   option1
      .
      .
      .
      DC      CL8'        '   optionN
```

OPERATION: ASSEMBLE first places the address of the auxiliary assembler dictionary in SSTATEXT and then sets a bit in SWITCH to indicate assembler mode is running.

ASSEMBLE next scans the options specified and uses the information thereby obtained to set up the option list for the assembler and the FILEDEF plists for the calls to FILEDEF. If a particular option is not selected, the corresponding default value appears in the list, which is then compacted to eliminate blanks before passing it to the assembler.

After all the options have been processed, if more than one filename was specified with the command, ASSEMBLE types a message at the terminal giving the name of the file about to be assembled. It then calls STATE to verify the existence of this file. If it does not exist, ASSEMBLE issues an error message (code 1) and returns to the caller. If it does exist, ASSEMBLE checks the item length, issues an error message if item length is incorrect and returns to the caller.

If the length is correct, ASSEMBLE calls ERASE to delete any existing TEXT, LISTING, and utility files for the current SYSIN file, and sets up storage by calls to STRINIT and GETMAIN.

It then calls ADTLKW to obtain the mode of the read-write disk with most available space and uses it to set up the FILEDEF plist for the SYSUT files. CMS control blocks (CMSCB's), which reflect the selected option, are set up for the TEXT, LISTING, SYSIN and utility files. After each successful return from FILEDEF, ASSEMBLE sets a clear switch to indicate which CMSCB's are to be cleared at the end of assembly and finally branches to IEUASM.

On return from the assembler, ASSEMBLE saves the returned error code if it is larger than the one returned for any previous assembly, erases the utility and clears the FCB which it had set up. If there are more files to be assembled, it goes back to the point after option scan to repeat the processing loop. At the completion of all assemblies, ASSEMBLE sets the release page bit, clears the SSTATEXT extension, clears SWITCH, places the error code in register 15 and returns to the user.

SPECIAL OUTPUT HANDLING ROUTINE:

The system routine SOEOB interfaces with LISTING and SYSUT2 files during the assembly.

SYSUT2 — If the file is not in "close" and it is being read in Phase 1, ASMHAND accesses the utility control table to ascertain the length and location of the record to be moved and moves it to the specified location. If the file is being read, but not in Phase 1, fixed length is forced and ASMHAND returns to the user.

If the file is being written in Phase 1, the utility control table is first set up by a call to GETMAIN and then updated to reflect the numbers of records read. If the file is being written but is not in Phase 1, ASMHAND forces a write of 4000 bytes and returns to the caller.

LISTING – If online diagnostics have been requested; ASMHAND checks each line for an error flag and prints these on the terminal along with a summary of errors at the end.

## FORTRAN

FUNCTION: To provide interface functions between the FORTRAN IV compiler and the CMS nucleus.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
      LA 1, PLIST
      SVC X'CA'
      .
      .
      .
PLIST DC CL8'FORTRAN '
      DC CL8'         '       '  filename1
      .
      .
      .
      DC CL8'         '       '  filenameN
      DC CL8'('                  separator for options
      DC CL8'         '       '  option 1
      .
      .
      .
      DC CL8'         '       '  option N
      DC XL8'xx'                 fence
```

ENTRY REQUIREMENTS:

R1 must contain address of plist referred to in calling sequence.

EXIT CONDITIONS:

Return to SVCINT.

CALLS TO OTHER ROUTINES:

FORTRAN compiler, SOEOB

CALLED BY:

SOEOB

OPERATION:

INITIALIZATION: FORTRAN saves the names of all requested program names and verifies that no more than 32 compilations are requested.

Then the option line is analyzed and the default option list modified to reflect user requests. An internal control switch is set to flag certain requests such as listing to printer, BATCH, NODIAG, LIST and such options as TDECK which require a call to the CMS routine FILEDEF for their implementation.

The auxiliary directory address is stored in SSTATEXT and the CMS compiler control switch SWITCH in NUCON is set to indicate FORTRAN compilation in process.

Initial housekeeping is completed by a series of STATE's to verify requested file existence, and ERASE's to eliminate existing LISTING and TEXT files for those FORTRAN files whose compilation is requested.

FILEDEF: This procedure, which is executed for each compile requested, starts with a series of calls to CMS FILEDEF to set up the FCB for the LISTING, TEXT, and FORTRAN files to be compiled, followed by a call to STRINIT to initialize free storage for the use of the compiler. Upon return from FILEDEF, R15 is tested: a positive return signals an error and causes cancellation of the job; a negative return indicates that the user has already set up his own FCB for this program so the FILEDEF clear switch is not set to prevent clearing of user specified FCB's; a zero return indicates a successful FCB initialize and the FILEDEF clear switch for that file is set.

On return from the compiler those FCB's for which the clear switch is set are cleared. If additional files remain to be compiled, FORTRAN returns to FILEDEF; otherwise CMS SWITCH SSTATEXT are cleared, the printer and punch are closed, and return is made to the user.

FORTHELP: This routine is entered from SOEOB whenever a listing line is processed. Its purpose to check for diagnostics and to print them on the terminal if the user has not specified NODIAG. Return is to SOEOB in all cases.

## PLI

FUNCTION: Compile PL/I source programs.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
        LA     R1, PLIST
        SVC    202
        .
        .
        .
PLIST   DS     0F
        DC     CL8'PLI'
        DC     CL8'filename 1'
        DC     CL8' ('          option delimiter
        DC     CL8'options'
        DC     CL8')'           option terminator
```

OPERATION:

1. Uses the value of register 15 as its base address.

2. Saves all registers in SAVEREG.

3. Sets SSTATEXT pointer to the address of PLIDIRT.

4. Sets the PL/I byte in the system SWITCH to signal PL/I Compilation in progress.

5. Sets the "OS language processor I/O pointer" to the address of PLIHAND - the PL/I LISTING file processor.

6. By using HNDSVC, sets pointers to alternate simulation routines for SVC's 19 (OPEN), 00 (XDAP), 47 (STIMER), 40 (EXTRACT).

7. Using the PLIOPTST section, isolates and scans all compiler and CMS options to be in use during its compilation - an option list (OPTIONS) is presented to the compiler phase IEMAB; this list consists of all defaulted values, unless an alternate option has been entered which matches one of its acceptable values from the list, OPTKEYWD.

8. A call to STRINIT initializes free storage pointers.

9. A GETMAIN macro is used to reserve core for the longest branch within the compiler overlay structure.

10. Sets the two flag bytes to reflect compiler information:

| | SWITCH+2 | PLSW |
|---|---|---|
| X'80' | Compilation | LISTING on Printer |
| X'40' | LISTING on Printer | No printing |
| X'20' | DIAG not wanted | DIAG not wanted |
| X'10' | No printing | Punch TEXT deck |
| X'08' | Execution | Not used |
| X'04' | Typing begun | Not used |
| X'02' | Not used | Not used |
| X'01' | Not used | Not used |

11. Uses STATE to verify existence and correct format of the source file.

12. Uses ERASE to remove all utility and output files for this source file (SYSUT1, SYSUT3, TEXT, LISTING).

13. Initializes file maintenance by placing the filename into the system I/O Plist - DCMSOP.

14. Uses the LOAD macro, SVC 8, to read into core IEMAA, the compiler basic phase.

15. Sets registers R13 - R1 and branches to IEMAA:

    R13 - A(SAVEAREA), R14 - A(return), R15 - V(IEMAA),
    R0 - (0), R1 - A(parameter list)

16. Upon return from IEMAA, the error code, if any, is stored into ERRCODE; the utility files are erased; the PRINTER is CLOSED, if the LISTING file was printed; and the TEXT deck is punched if desired.

17. Before returning to CMS, clears SWITCH, resets STRINIT storage pointers, zeroes out SSTATEXT, destroys the alternate SVC simulation list, performs FINIS on all files, places the ERRCODE into 15, and returns to INIT.

During compilation, when a PUT to the LISTING file is executed, control will reach PLIHAND. It will be determined whether the LISTING file is to be output to the printer, left on the user's P-Disk, or destroyed because the no print option was specified. Also, if NODIAG was not specified, a search for the listing record "-ΔΔΔΔ COMPILER Δ DIAGNOSTICS.Δ" is made. When found, all subsequent lines are also typed to the user's terminal with a call to the TYPLIN routine. Records are put onto the printer with a call to PRINTIO routine.

## SNOBOL*

FUNCTION: To compile programs written in SNOBOL into SPL/1, a more basic string-processing language, and to execute SPL/1 programs interpretively.

CALLING SEQUENCE:

```
       LA      1, PLIST 1
       SVC     X'CA'
       .
       .
PLIST  DS      0D
       DC      CL8'SNOBOL'
       DC      CL8'        '     filename
       DC      CL8' (option 1'   option 1 preceded by (
       DC      CL8' option N)'   option N ending with )
```

*SNOBOL is available as Type III program number 360D-03.2.016.

OPERATION: The CMS SNOBOL system operates in two passes: compilation and assembly-interpretation. The SNOBOL-to SPL/1 compiler was itself written in SNOBOL, and consists of three SPL1 subprograms on the system disk: MONITOR, COMPILER, and CRUNCH. The output from the compiler--the user program in SPL/1--is input to the SPL/1 assembler-interpreter, which is a group of programs written in assembler language. The components of the assembler-interpreter are: SPL1, containing the main control routine and some I/O routines for other phases; SPL1ASM, the assembler phase; SPL1INT, the interpretive execution phase; SPL1IOS, a general I/O handler for all phases; and SPL1FRE, a storage management routine.

During the compilation phase, a file--filename LISTING P1--is generated, unless the option NOLIST is included in the parameter list. If OFFLINE or PRINT is specified, this information is also printed offline. ONLINE specifies that the information is to be printed at the terminal. The SPL1 option specifies that the file named in the parameter list has already been compiled into SPL/1, and the compilation phase may be skipped.

When SNOBOL is issued, the main SPL1 control routine is entered. SPL1 branches to SPL1IOS, to interpret the parameter list, and to SPL1FRE, for free storage, then passes control to SPL1ASM to assemble the SNOBOL compiler. The compiler is subsequently executed by SPL1INT, producing a user program in SPL/1, which in turn is passed again to SPL1ASM. For details on the compiler itself, and the operation of the assembler-interpreter, see SPL/1:A String Processing Language, (320-2005) available from the IBM Cambridge Scientific Center, Cambridge, Massachusetts.

## BRUIN*

FUNCTION: To provide an interactive algebraic desk calculator/interpreter facility within CMS.

CALLING SEQUENCE:

```
        LA      1, PLIST
        SVC     X'CA'
        .
        .
        .
PLIST   DC      CL8'BRUIN'
```

OPERATION: BRUIN accepts commands from the terminal that fall into two classes: (1) direct commands that are executed immediately; (2) indirect commands that are stored and may be executed as part of a program at a later time. Values of variables defined by direct or indirect commands are also stored in core.

*BRUIN is available as Type III program number 360D-03.3.013.

BRUIN can store and retrieve indirect commands and values of variables on the disk. Files created by BRUIN have the file name defined by the user and a filetype of BRUIN.

While BRUIN is in the process of creating or replacing a file, the file name .TEMP. BRUIN IS USED.

BRUIN is a module loaded at 12000 that contains the following text decks:

| | | |
|---|---|---|
| BRUIN | - | must be first - contains CALCIO |
| BOIL | - | BRUIN interprets except for I/O and functions |
| ABS | | |
| SQRT | | |
| LOG2E10 | | |
| EXP | | |
| SINCOS | | |
| TAN | | |
| ATAN | | |
| SINCOSH | | |
| TANH | | |
| ATNH | | Arithmetic function subroutines |
| ERF | | |
| IPFP | | |
| RAND | | |
| GAMMA | | |
| UPPR | | |
| LENGTH | | |

Note: BOIL and all of the arithmetic functions are combined into a deck called BRUINTXT TEXT.

The only program that was modified to place BRUIN into CMS instead of an OS environment was CALCIO. This subroutine (called BRUIN in the CMS version) has two functions: (1) entry is from CMS and return is back to CMS; (2) all input/output functions, input/output calls are made with explicit CMS SVC calls rather than with simulation of OS access methods.

To assemble BOIL and the arithmetic functions, BRUINLIB MACLIB, consisting of one macro called BOILSECT, must be made accessible with a GLOBAL command.

## UTILITIES

The utilities available to the CMS user are CNVT26, COMPARE, CVTFV, DISK, DUMPD, DUMPF, DUMPREST, ECHO, FORMAT, MAPPRT, MODMAP, OSTAPE, SORT, STAT, TAPE, TAPEIO, TAPRINT, TPCOPY, and WRTAPE. These are described in detail on the following pages.

### CNVT26

FUNCTION: Convert a BCDIC (026) file to an EBCDIC (029) file.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
       LA    1,PLIST
       SVC   X'CA'
       .
       .
       .
PLIST  DC    CL8'CNVT26'
       DC    CL8'filename'
       DC    CL8'filetype'
```

SOURCE LANGUAGE:

CNVT26 was originally coded in the AED-0 language. The BAL source is the output produced by the AED-0 compiler. All procedures uses are members of the AED program library AEDLIB TXTLIB.

OPERATION:

1. If the filename and/or filetype are missing, types 'FILENAME(S) MISSING' and returns error code 2.

2. If the file is not present, types 'FILE NOT FOUND' and returns error code 1.

3. Erases the work file BCDEBC UTILITY if present.

4. For each record:

   a. Reads a record via procedure RDCMS.
   b. Expands and translates character string via procedure SPRBCD.
   c. Contracts character string via procedure GLUE.
   d. Writes record into work file BCDEBC UTILITY via procedure WRCMS.

5. Closes both files, erases original file, and alters name of work file to that of original file.

## COMPARE

FUNCTION: To compare two disk files.

ATTRIBUTES: Disk resident.

CALLING SEQUENCE:

```
        LA      1, PLIST
        SVC     202
        .
        .
        .
PLIST   DC      CL8'COMPARE'
        DC      CL8'filename1'
        DC      CL8'filetype1'
        DC      CL8'filemode1'
        DC      CL8'filename2'
        DC      CL8'filetype2'
        DC      CL8'filemode2'
        DC      CL8'(NOSEQ' option to omit comparison on last 8 bytes of each record.
```

EXIT CONDITIONS:

Exits to user. R15 = 0 of no errors

ERROR RETURNS:

| | |
|---|---|
| E(00001) | Parameter error |
| E(0002) | First and second files are the same file |
| E(0003) | At least one record differs |
| E(0004) | Fatal error |

CALLS TO OTHER ROUTINES:

STATE, RDBUF, FINIS

CALLED BY:

User

OPERATION: The COMPARE parameter list is checked for errors, and if an error exists COMPARE exits with an Error 1. If all parameters are present, COMPARE checks to see if the user specified the option (NOSEQ). If this option was specified, the last eight bytes of each record will not be compared. A STATE is then done of each file in the parameter list, and, if they both exist, the following is checked. Are they the same file? If so, exit with Error 2. Is one file a fixed number of bytes per record and the other variable? If so, exit with Error 4. If there are no errors, COMPARE now RDBUF's each file comparing a record of one file against the corresponding record of the second file on a byte to byte comparison. If a discrepancy is found the records that were being checked are typed on the terminal to the user. The checking is finished.

After successful completion, and prior to returning to the user or caller, COMPARE references NUCON and turns the page release flag on. When the program returns to INIT, this flag is checked and, if it is on, INIT issues a diagnose X'10' to CP to release the user pages from X'10' 12000 Hex up to the value of LOWEXT.

COMPARE then calls FINIS to close the files and exit to the user.

## CVTFV

ATTRIBUTES: Disk resident, module name = CNVTFV

FUNCTION: To convert a fixed-length record file to a variable-length record file. Option T specifies deletion of trailing blanks.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
         LA     1, PLIST
         SVC    X'CA'
         .
         .
         .
PLIST    DC     CL8'CVTFV'
         DC     CL8'filename'
         DC     CL8'filetype'
         DC     CL8'(T)'   optional
```

CALLS TO OTHER ROUTINES:

ERASE, ALTER, RDBUF, WRBUF, STATE, FINIS

CALLED BY:

User

SOURCE LANGUAGE:

The CVTFV module is an AED-compatible assembly language module. All non-CMS procedures called by CVTFV are AED program library routines. Therefore, to load CVTFV, AEDLIB TXTLIB is required.

OPERATION:

1. If filename and/or filetype not supplied, types 'INCORRECT FORMAT' and returns error code 3.

2. Checks for optional parameter T. If item in parentheses not T, types 'INCORRECT FORMAT' and returns error code 2.

3. Checks for existence of file. If not found, types 'FILE NOT FOUND' and returns error code 1.

4. If file already has variable length records, types 'FILE IS ALREADY VARIABLE' and returns error code 4.

5. Erases 'filename CVTUT1' if it exists.

6. For each record, performs the following operations:

   a. Calls RDCMS procedure to read a record.

   b. If truncation (T) was not specified, goes to step e.

   c. If record length is 80 bytes, deletes bytes 73-80 regardless of contents.

   d. Reduces record length to delete any trailing blanks.

   e. Calls WRCMS procedure to write variable record in work file 'filename CVTUT1'.

7. When all records are processed, closes all files, erases the original file, and alters the name of the work file to the original filename.

## DISK

FUNCTION: To dump a disk file to cards, or to load one or more files from cards to disk.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
        LA     R1, PLIST        R1 must point to P-List as usual
        SVC    X'CA'
        DC     AL4(ERROR)
```

ENTRY REQUIREMENTS:

R1 must point to DISK parameter list, either:

```
        DS     0F
PLIST   DC     CL8'DISK'
        DC     CL8'LOAD'
           or
        DS     0F
PLIST   DC     CL8'DISK'
        DC     CL8'DUMP'
        DC     CL8'        '    Filename
        DC     CL8'        '    Filetype
       [DC     CL2'        ']   Filemode
```

EXIT CONDITIONS:

Normal Return

R15 = 0

Error Returns

R15 = 1 to 8          (See "ERROR RETURNS" later in this section)

CALLS TO OTHER ROUTINES:

ERASE, FINIS, FSTLKP, RDBUF, STATE, UPDISK, WRBUF

CALLED BY:

User

MACROS USED:

FSTB, FVS

ERROR RETURNS (R15 value at exit):

1. Invalid Parameter List
2. FATAL PUNCH ERROR
3. FATAL DISK ERROR
4. FATAL READER ERROR
5. ILLEGAL CARD IN DISK LOAD DECK
6. END CARD MISSING FROM DISK LOAD DECK
7. FILE NOT FOUND
8. READER EMPTY OR NOT READY

OPERATION: The operation of DISK depends on whether the calling program specifies DUMP or LOAD.

DUMP: DISK copies the file designation from the parameter list into bytes 58 - 76 of and 80-byte buffer. (The first four bytes of the buffer contain an identifier consisting of an internal representation of a 12-2-9 punch and the characters 'CMS'.) Then DISK temporarily changes the characteristics of the file in the 40-byte FST entry to make it appear as a file of 800-byte fixed-length records. (The correct FST entry is restored when the file has been dumped, of course.) DISK moves the initial value for sequencing (0001) into bytes 77-80 of the buffer. DISK next calls the RDBUF function program to read the first 50 bytes of the temporary copy into bytes 6-55 of the buffer and then the CARDPH function program to punch the contents of the buffer. Having punched the first card, DISK increments the sequence number (bytes 77-80 of the output buffer) and overlays bytes 6-55 of the buffer with the next 50 bytes of the file by calling RDBUF. It then punches the contents of the buffer. DISK repeats this process for each subsequent 50 bytes of data in the temporary disk file. When the end-of file is encountered, DISK generates an end card (one with N in column 5) and punches it, calls the CLOSIO command program to close punch operations, restores the FST entry to its correct value, and returns to the caller.

LOAD: DISK calls the ERASE command program to erase the temporary file ( (DISK) (TFILE) P3) created during a load operation.. Next, it calls the CARDRD function program to read the first card. (If this card was produced by the dump portion of DISK, it will contain an identifier in columns 1-4.) DISK then checks the identifier in the card. If invalid, it issues a message to the effect that there is an illegal card in the disk load deck, calls the CLOSIO command program to close card reader operations, and returns (via SVCINT) to the calling program (error code 4). If the identifier is valid, DISK determines whether the card is an end card (that is, one with N in the fifth byte). If it is not, DISK moves the file data portion of the card (50 bytes in columns 6-55) into the next available location in an 800-byte output buffer. DISK then calls the CARDRD function program to read the next card, which it processes similarly. When the entire 800-byte output buffer has been filled with data from the input cards, DISK calls the WRBUF function program to write the contents of the buffer into a file designated as (DISK) (TFILE) P3. DISK repeats the process of filling the output buffer and writing its contents into the disk file until the end card is read.

When the end card is read, DISK calls the FINIS command program to close the disk file ( (DISK) (TFILE) P3) created from the file in the card deck. It then calls the ERASE command program to erase the file (if any) that has the same designation as the card file just converted to a disk file. Next, DISK calls the FSTLKP function program to locate the file status table for the disk file. (This file is again (DISK) (TFILE) P3.) Subsequently, DISK moves the designation for the card file from the end card into the corresponding locations in the file status table. This completes the conversion of the first card file in the card reader to a disk file, and DISK calls the TYPLIN function program to type a message at the terminal to the effect that the file has been loaded. DISK processes the next file in the card reader in a similar manner.

When an end-of-file on the card reader is encountered, DISK calls the CLOSIO command program to close card reader operations and returns to the calling program.

Notes: UPDISK is called at the appropriate time when DISK LOAD is being executed, to update the directory for the file being loaded.

DISK is a feasible way to transfer variable-length files, such as MODULE's or SCRIPT files, between one user and another.

DISK DUMP can dump files from any disk. DISK LOAD loads files only onto the P-Disk. The mode number of the file is retained (for example, a T5 file that was dumped would become a P5 file when loaded), except that an SY file becomes P1 when loaded.

The date/time are that of the new file loaded.

## DUMPD

FUNCTION: To dump the contents of one direct access record, specified by a CCHHRR address. (The dump is in hexadecimal.)

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
        LA   1, PLIST
        SVC  X'CA'
        .
        .
        .
PLIST   DC   CL8'DUMPD'
        DC   CL8'        '        device address
        DC   CL8'        '        cylinder
        DC   CL8'        '        head
        DC   CL8'        '        record number
```

OPERATION: DUMPD scans the arguments entered by the user to determine whether the user has requested a dump of a whole cylinder, a track, or a specific record. It converts the arguments found to binary and tests to determine whether the unit being dumped is a 2311 or a 2314. DUMPD then converts the unit address to binary and reads a record into core. It determines whether the right unit has been accessed and if so dumps the record to the printer.

## DUMPF

FUNCTION: To type online, the contents of a specified file in hexadecimal.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
        LA   1, PLIST
        SVC  X'CA'
        .
        .
        .
PLIST   DC   CL8'DUMPF'
        DC   CL8'filename'
        DC   CL8'filetype'
        DC   CL8'        '        starting line (optional)
        DC   CL8'        '        ending line (optional)
        DC   CL8'        '        line-limit (optional)
```

OPERATION: DUMPF scans the P-List to determine the file requested. It then calls STATE to see if the file exists. If the file exists, DUMPF gets the location of the FST. It checks the arguments to determine whether a starting line is supplied, whether an ending line is supplied, and whether the user has requested a line limit.

DUMPF then sets a read pointer to the specific line requested by calling POINT and then calls RDBUF to read the line. DUMPF then converts the data to hexadecimal and prints the hex information online. It repeats this procedure until the requested number of lines have been done.

## DUMPREST

FUNCTION: To dump the contents of a disk to tape or to restore a disk from the contents of the tape.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
         LA    1, PLIST
         SVC   X'CA'
         .
         .
         .
PLIST    DC    CL8'DUMPREST'
```

OPERATION: DUMPREST prompts the user via terminal message to indicate whether he wishes to dump or restore, to specify the addresses of the disk and tape as well as the device type of the disk, and to indicate whether tape rewind is desired. DUMPREST will operate only on CMS-formatted disks (that is, 4 blocks per track for a 2311, 15 blocks per 2 tracks for a 2314. A block is 829 bytes.) If cancel is entered in response to any of these requests, DUMPREST will reinitialize itself.

DUMP: DUMPREST reads a track from the disk and writes it onto tape. It does this for each track on the disk; a block of disk data becomes one tape record. When the end of the disk is reached or if an error occurs during a disk-read operation, DUMPREST writes an end-of-file on the tape, types a message at the terminal indicating the number of cylinders that were successfully dumped, types a message at the terminal indicating the number of recoverable tape-write errors there were, and returns to the calling program.

If an error is encountered during a tape-write operation, DUMPREST retries the operation. If the operation is not successful after 10 retries, DUMPREST types a message at the terminal indicating the number of cylinders that were successfully dumped, types a message at the terminal indicating the number of recoverable tape-write operations, and returns to the calling program.

If the end of the tape is reached before the entire disk has been dumped, DUMPREST proceeds in the same manner as for an unrecoverable tape error.

RESTORE: DUMPREST restores cylinders by reading records from tape and writing them onto disk. When an end-of-file on the tape is encountered, or if a disk error occurs, DUMPREST types a message at the terminal indicating the number of cylinders that were successfully restored, types a message at the terminal indicating the number of recoverable tape errors, and returns to the calling program.

When restoring, DUMPREST retries an unsuccessful tape-read operation a maximum of 10 times. If the retries are not successful, it types both the number of cylinders that were successfully restored and the number of recoverable tape errors at the terminal, and returns to the calling program.

## ECHO

FUNCTION: Test terminal input/output

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
        LA    1, PLIST
        SVC   X'CA'
        .
        .
        .
PLIST   DC    CL8'ECHO'
                     { D }      Translate lower case to upper case
        DC    CL8'   { S }      and interpret delete characters.
                                Do not translate but do interpret delete
                     { X }      characters - do not change line.
        DC    CL2'nn'   '       Repeat input line, nn times.
```

OPERATION: ECHO first checks to see if a parameter has been issued for the input line to be repeated. If none has been stipulated, the default is 1. Next ECHO calls the 'TYPLIN' function to print onto the console 'START CONSOLE TEST'. It then checks the supplied code (U, S, X) to determine which has been supplied. If this parameter is an 'X', ECHO saves the present linend character by setting up a PLIST with a call to the LINEND command. The WAITRD function reads the first input line provided and edits it according to the code that was supplied. When return is made from WAITRD, echo calls the TYPLIN function to type the input line onto the console, where the user can verify that the output is identical to the input. This is repeated until the user enters RETURN, then ECHO restores the original linend character if the X parameter was in effect, calls the TYPLIN function to type 'END CONSOLE TEST', and then returns to the user via a branch to register 14.

## FORMAT

FUNCTION: To set the P-Disk, T-Disk, or other read-write disk to CMS record format, clearing any information currently on the disk; to write a record-label on a read-write disk; or to recalculate the number of cylinders and other disk statistics.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
        LA    R1, PLIST            R1 must point to P-List as usual
        SVC   X'CA'
        DC    AL4 (ERROR)
```

ENTRY REQUIREMENTS:

```
        R1 must point to FORMAT parameter list:
        DS    0F
PLIST   DC    CL8'FORMAT'
        DC    CL8'm'                 m = Disk-Mode (P, T, etc.)
       [DC    CL8'          ']       Additional parameters as needed
              .
              .
              .
       [DC    CL8'          ']       (See examples of valid P-Lists)
```

EXIT CONDITIONS:

<u>Normal Return</u>
R15 = 0

<u>Error Returns</u>
R15 nonzero (See "ERROR RETURNS")

CALLS TO OTHER ROUTINES:

ADTLKP, FREE, FRET, RDTK, RELUFD, STAT, UPDISK, WRTK

CALLED BY (where known):

INIT, or User from terminal

MACROS USED:

ADT, FVS

EXAMPLES OF VALID FORMAT PARAMETER LISTS:

For permanent disk (P-Disk)...

To format a disk very first time
FORMAT P ALL

To format disk subsequently
FORMAT P

To format a disk limiting the number of cylinders (examples)
FORMAT P ALL 54
or
FORMAT P 150
etc.

To write new label on disk
FORMAT P L

To check number of cylinders and disk counts
FORMAT PC

To check number of cylinders and disk counts and to revise (expand or reduce) bitmask (PQMSK) if necessary
FORMAT P R

To revise disk counts for smaller number of cylinders (examples)
FORMAT P R 53
or
FORMAT P R 200

To revise disk counts to leave room for CMS nucleus as written by IPLDISK, on 54-cylinder 2314, or 203-cylinder 2311
FORMAT P R SYS

To suppress normal typed messages (example)
FORMAT P (NOTYPE)

For temporary disk (T-Disk) . . .

To format T-Disk first time or subsequently
FORMAT T or FORMAT T ALL (equivalent)

To check number of cylinders and disk counts
FORMAT T C

To suppress normal typed messages (example)
FORMAT T C (NOTYPE)

ERROR RETURNS (R15 value at exit, with message as shown):

1. NEITHER PERMANENT (P) NOR TEMPORARY (T) DISK SPECIFIED.

2. CONDITION CODE 1, 2, OR 3 ON SIO IN FORMATTING DISK.

3. UNEXPECTED UNIT-CHECK FORMATTING DISK (SENSE-BYTE NOT 81).

4. CE & DE NOT FOUND TOGETHER (IN CSW) WHILE FORMATTING DISK.

5. CONDITION CODE 1, 2, OR 3 ON SIO IN CHECKING NO. CYL. ETC.

6. CE & DE NOT FOUND TOGETHER (IN CSW) WHILE CHECKING NO. CYL.

7. UNEXPECTED UNIT-CHECK CHECKING NO. CYL (SENSE-BYTE NOT 81).

8. NO T-DISK AVAILABLE.

10. DISK READ-ONLY OR NOT LOGGED-IN FOR "FORMAT P C/R" CALL.

11. FORMAT NOT EXECUTED IF "YES" NOT INPUTTED FROM TERMINAL.

12. DISK (OTHER THAN T-DISK) NOT ATTACHED.

13. "FORMAT P R" FAILURE - DATA-LOSS WOULD RESULT.
(NOTE - COUNTS ARE UNCHANGED, TO PRESERVE USER'S DATA).

14. "FORMAT P R SYS" FAILURE - DISK IS SMALLER THAN REQUIRED FOR USE OF "SYS" OPTION.
(NOTE - COUNTS ARE UNCHANGED, TO PRESERVE USER'S DATA).

OPERATION: The CMS FORMAT program has many options, as seen above in the EXAMPLES OF VALID FORMAT PARAMETER LISTS. These include the following:

- FORMATing the P-Disk (or A-Disk, B-Disk, or C-Disk)
  or
  the T-Disk (handled somewhat differently)

- FORMATing a 2311 or 2314

- FORMATing "all" records, or skipping over the first three records

- Limiting the number of cylinders formatted, if desired

- Checking the number of cylinders and verifying other disk counts

- Checking number of cylinders and disk counts as above, but revising the counts as desired (if feasible)

- Writing a new label on Disk

- Typing normal messages, or omitting such typeouts

- The option to cancel the FORMAT call and not format the disk at all, in case FORMAT was called accidentally or inadvisedly.

## Initialization Phase

FORMAT checks the parameter list for which options were specified; it also checks for various parameter list errors.

The disk mode letter is checked; it must be alphabetic, and S (for the S-Disk) is not permitted. ADTLKP is called to find the corresponding active disk table. If an error from ADTLKP occurs, FORMAT types an error message and returns an error code 1. The device-address in NUCON (pointed to by the ADTDTA pointer in the active disk table) is obtained and checked. A value of zero is not valid, and an error message and code are returned.

The disk is then sensed to make sure it is attached and ready, and to see if it is a 2311 (sense byte of x'C8'), or a 2314 (sense byte x'40'). Anything else results in the return of a suitable error message and code.

At this point FORMAT continues, depending upon which of the several options was specified, as described in the following sections.

Real Format

If the disk is really to be formatted (none of the special options C, R, or L was specified), the procedure used is as follows. The general description will be that of the procedure followed for the P-Disk; formatting of the A-Disk, B-Disk, or C-Disk is identical in operation. Where there are differences for the T-Disk, they are noted and also summarized in a later section.

1. To guard against accidental or incorrect call of FORMAT, which wipes out all files on disk, a message is typed on the user's terminal before any tables are cleared or anything is written on disk. This message (with disk-mode and device-address filled in to their correct values) is as follows:

   ```
   **"FORMAT P" WILL ERASE ALL YOUR P-DISK (0191) FILES**
   **DO YOU WISH TO CONTINUE? ENTER "YES" or "NO":
   ```

   The user must type in YES or "YES for formatting to be undertaken. Any other input at all from the terminal (such as "NO") will result in an error-code 11 being returned, nothing at all in the NUCON or Active Disk Table being affected, and the following message:

   ```
   "FORMAT" WILL NOT BE EXECUTED
   ```

   For the T-Disk, if the (NOTYPE) option was given, this entire step is omitted; formatting proceeds with no message to the terminal or further input from the user.

2. If FORMAT P ALL was specified, the user is now prompted to enter a label which is to be written on record 3 of the disk. (See "CMS Disk Label" later in this section.) A message is typed on the terminal as follows:

   ```
   ENTER 6-BYTE LABEL (IF WANTED), OR NULL LINE (IF NOT).
   ```

   WAITRD is then called to obtain the label typed in by the user.

   If a null line (plain carriage return) was entered, some time information from location x'98' in lower core is used in place of the label. Otherwise, the typed-in label (blank-filled if less than six bytes, truncated if more) is written on the first ten bytes of record 3 when the formatting is done. (For example, if the user typed in MYDISK, the label would be: CMS=MYDISK)

For the T-Disk, FORMAT T is treated as FORMAT T ALL; no message on the terminal is given, nor user reply, and a label of CMS=T-DISK is always used.

3. If ALL was not specified, for the P-Disk or equivalent, the first three records of cylinder 0, head 0 are read instead of being written. (If the disk has never been formatted before, this will normally cause an error, and FORMAT P ALL should then be used instead.)

4. At this point, RELUFD is called to release and clear all appropriate old core-resident tables for this disk, and the R/O and R/W flag-bits in the ADTFLG1 flag-byte in the Active Disk Table are cleared.

5. Just before formatting starts, a message is typed, of the following form:

   ```
   FORMATTING P-DISK (2314) . . .
   ```

   This message confirms to the user that the format program is formatting the desired disk, and indicates the disk type. If the (NOTYPE) option was specified, the message is omitted.

   Formatting of the disk then commences. A 2311 is formatted by writing four 829-byte records per head, ten heads per cylinder. A 2314 is formatted with fifteen 829-byte records per two heads, for ten pairs of heads per cylinder. The data written (except for the label) consists of binary zeroes. A read-after-write check is included in the CCW chain for the P-Disk or equivalent, where the data written on disk is immediately read (in non-transmit mode) to check that the formatting was successful. For purposes of speed, the read-after-write check is not performed on the T-Disk, as the T-Disk may be formatted once for each terminal session, while the P-Disk is usually formatted only once in a great while.

   The CCW chain writes no less than 8 bytes for any single CCW command and always writes from a double-word boundary. This is the correct procedure to preclude data-chaining errors, particularly when running on a 360 model slower than a 65.

   If errors do occur, repeated efforts to recover are made; if a permanent error occurs, a message is typed indicating the trouble, and formatting of the disk is truncated at the end of the last cylinder successfully written.

6. Formatting of the disk concludes when the end of disk is reached (determined by a unit check coupled with a sense byte of x'81'), or if a specified limit by the user is reached (for example, 50 cylinders for FORMAT P 50), or if a permanent error occurs, whichever happens first.

   If the number of cylinders formatted is zero, then FORMAT exits with an error message, and no further action is taken.

7. If at least one cylinder was successfully formatted, then FORMAT concludes as follows:

   a. Stores the number of cylinders ADTCYL in the active disk table.

   b. Types a message indicating how many cylinders were formatted, unless the (NOTYPE\ parameter was given, in which case it is omitted.

   c. Stores the unit-type-byte of x'01' or x'08' in the appropriate slot in the NUCON table.

   d. Obtains a 816-byte block from free storage, if necessary, for the first FST hyperblock, clears it, and places its address in the active disk table.

   e. Obtains a 200-byte block from free storage, if necessary, for the QQMSK table, clears it, and places its address in the ADT table.

   f. Obtains free storage for the QMSK bit-mask table, the size depending on the number of cylinders, sets the first word to its default value of x'F0000000', clears the remainder of the table, and places its address in the ADT table.

   g. Initializes all other counts in the ADT table as needed (ADTNUM, etc.), and flags the disk as logged in and read-write.

   h. Calls UPDISK to write the finished file directory on disk.

8. Finally, FORMAT returns to the caller with the appropriate error-code (= 0 if all was successful) in R15.

FORMAT P C Call

If a FORMAT P C (or FORMAT T C, e tc.) call is issued, FORMAT takes the following action:

1. ADTLKP is called (in the initialization process, as usual) to ensure that the disk mode is valid, and the active disk table is checked to make sure that the disk is logged in and in read-write form (error message and return if not).

2. Successive seeks are executed to determine how many cylinders are actually available on the 2311 or 2314 disk.

3. The number of records on disk ADTNUM is computed, depending on the number of cylinders, and compared with the old ADTNUM. Whichever is less is taken as the correct value of ADTNUM. The actual bits in the QMSK bit-mask are then counted to compute the value of ADTUSED (number of records in use), ADTLEFT (number left), and ADTLAST. ADT1ST is cleared, and the number of cylinders ADTCYL (from step 2) is storea.

4. UPDISK is then called to ensure that the recomputed counts are stored on disk.

5. Finally, STAT P (or STAT T, etc.) is called to display the disk counts to the user.

FORMAT P C can be called by the user if desired to ascertain the actual number of cylinders on a disk, and to verify that the other disk counts are correct. If it is desired to revise the number of cylinders and disk counts, FORMAT P R should be called.

FORMAT P R Call

FORMAT P R (or FORMAT T R, etc.) has several uses, particularly when its options FORMAT P R nn (nn being a cylinder count) or FORMAT P R SYS are used.

FORMAT P R (with no options) is used to ascertain the number of cylinders and recompute the disk counts as in FORMAT P C, but also has the capability of revising the disk counts upward if ADTNUM is greater than it was previously.

FORMAT P R nn (where nn is a decimal number of cylinders desired) works like FORMAT P R with no options, except that the number of cylinders is limited to the 'nn' given by the user.

FORMAT P R SYS is a special option used for a 54-cylinder 2314 or 203-cylinder 2311 to recompute the counts to leave room for the CMS nucleus as written on disk by the IPLDISK program, precluding the possible loss of data.

The action taken for FORMAT P R (with or without options) is as follows:

1. ADTLKP is called and the disk checked to make sure it is logged in and in read-write form, as in FORMAT P C.

2. Successive seeks to the disk are performed as in FORMAT P C to determine the actual number of cylinders on the 2311 or 2314 disk.

3. The number of records on disk ADTNUM is computed from the actual number of cylinders, if no options were given. If "nn" was specified, the "nn" count or the actual number of cylinders is used, whichever is less. The revised disk counts are then computed as in FORMAT P C. If the new "ADTLAST" (plus a safety factor for the ADTRES reserve count) is less than the old ADTLAST, a loss of data would result; in this case, a warning message is given to the user, the old disk counts are left intact, and error 13 is returned to the caller. If the new ADTNUM is the same as the old, FORMAT P R finishes up the same as FORMAT P C, with a call to UPDISK and STAT.

4. If the total number of records on disk ADTNUM is not the same as previously (and no data-loss will occur), FORMAT P R obtains a new QMSK bit-mask corresponding to the new disk counts, moves the old QMSK bit-mask thereto, truncating or zero-filling as appropriate, and gives back the old bit-mask to free storage. Then all new counts are stored in the active disk table (including the revised ADTCYL cylinder count), UPDISK is called, and STAT, as in FORMAT P C.

5. FORMAT P R SYS is similar to FORMAT P R nn, but has the following features:

   a. Uses an ADTNUM of 7976 (the largest multiple of 8 records within the block-number of 7980 used by IPLDISK).

   b. If truncation of the disk counts at 7976 would cause loss of data, gives an error-message and error-return similar to the logic for FORMAT P R nn. The old disk counts are retained as is.

   c. A check is also made to ensure that the disk is big enough for the new ADTNUM of 7976 records. If not, an error message is given, error 14 is returned, and the old disk counts are retained.

   d. If the disk is large enough, and no data-loss would result, then FORMAT P R SYS recomputes them on the basis of ADTNUM=7976, with the correct other disk counts, obtains a new QMSK if necessary as in FORMAT P R, stores all corrected counts, calls UPDISK, and finally STAT, as above.

These three options (FORMAT P R, FORMAT P R nn, and FORMAT P R SYS) make it possible to revise disks whenever feasible, to larger or smaller sizes, without the necessity of dumping files out on tape, formatting the disk, and loading them back in again. The only requirement, other than those discussed above, is that when a disk is enlarged via FORMAT P R it was previously formatted at some time to its full size.

## CMS Disk Label

Record 3 (Cylinder 0, Head 0, Record 3) of a CMS Disk now includes a ten-byte label, consisting of the following:

1. Four bytes: CMS=

2. Six bytes: desired label
   (blank-filled if less than 6 bytes;
   truncated if more than 6 bytes)

3. Remaining 819 bytes of record = 00 (binary zeroes)

## Option to Write a Label on Disk

As mentioned earlier, the option to format all of a CMS Disk causes the disk to be formatted including a label on record 3. (See "CMS Disk Label" for details.)

A label can also be entered on a disk that has been formatted previously (either to change an existing label or to place a new label on a CMS Disk that does not have a 10-byte label as now used), without affecting any other information on disk.

This is done by issuing the command FORMAT P L (or FORMAT T L — the T-Disk is quite acceptable for this option). The logic performed by FORMAT P L (or equivalent) is as follows:

1. ADTLKP is called to make sure a disk exists for the mode-letter given, and does not = the S-Disk. (This is accomplished as part of the general initializing process.) The disk must, of course, be attached, ready, and be a read-write disk, for the command to succeed.

2. RDTK is called to read the old label from disk into an 829-byte I/O buffer. (If RDTK should fail, a descriptive error message is given, and the error-code from RDTK is returned to the caller of FORMAT P L.)

3. A message is typed on the user terminal as follows:

   ENTER 6-BYTE LABEL:

4. WAITRD is called to obtain the label typed in by the user.

5. CMS= (four bytes) and the first six bytes of the entered label (blank-filled, if less than 6 were inputted) are moved to the first ten bytes of the 829-byte I/O buffer

6. WRTK is then called to write the new label back on the disk. (If WRTK should fail, a descriptive error message is given, and the error-code from WRTK is returned to the caller of FORMAT P L. If the failure is because the disk is read-only, the error-code = 6).

Summary of Differences in Formatting a T-Disk

As mentioned above, there are several differences in the way a T-Disk is formatted from the procedure used for a P-Disk or other read-write disk. These are summarized as follows:

1. FORMAT T is equivalent to FORMAT T ALL. (All records on the disk are formatted.)

2. `A label of CMS=T-DISK is automatically written on record 3. (See note below.)

3. The read-after-write check in the CCW chain to format the disk is omitted, in the interests of making the formatting of a temporary disk as fast as possible.

4. The requirement for the user to type in YES or "YES before formatting begins is waived if the (NOTYPE) parameter was given.

5. The "NO T-DISK AVAILABLE" error message (if appropriate) is omitted (as well as the normal formatting messages) if the (NOTYPE) parameter was given.

Note: If desired to change the label on a T-Disk after it has been formatted, the command FORMAT T L can be issued, and the replacement label entered on disk.

## GENDIRT

FUNCTION: To complete auxiliary system status tables.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
                        module        other         directory
         GENDIRT
                        name          phases        name
```

OPERATION: GENDIRT will be used primarily by the system programmer whose responsibility it will be to maintain language processor modules on the system disk.

All TEXT decks for the language compiler, the CMS interface and other routines, and the auxiliary directory should be placed onto the read-write system disk. A series of LOAD, REUSE, GENMOD, and LOADMOD commands can now be used to create a modular overlay structure that will be in effect during compilation. After the last module is created, the CMS interface routine, other needed routines, and finally the auxiliary directory are again loaded. The GENDIRT command is issued with the interface name as the first argument, and the auxiliary directory as the last argument.

GENDIRT will operate on the auxiliary directory in the same manner as SYSGEN, when the latter routine completes the nucleus directory, SSTAT. That is, each FST entry will be STATE'd and the address of file's First Chain Link will be placed into the FST slot. Then a GENMOD will be issued to create an interface module.

Subsequently, whenever the language processor is invoked, part of the interface module will be the completed auxiliary directory that is used as an extension to the system disk directory, SSTAT.

To satisfy a reference to a processor module, SSTAT will be searched to locate its FST entry and then its first chain link address. If the entry is not found, the SSTATEXT pointer is used to access the address of any extension directories (if zero, no directories exist). Scanning of the auxiliary directory commences until the FST entry for the desired module is found and the loading of the routine may be executed.

Auxiliary directories are used so that the core-resident nucleus routine, SSTAT, would not be cluttered with specific 40-byte FST entries that are referenced by occasional calls to language processors.

For an example of the usage of GENDIRT, see the CP-67/CMS Installation Guide, GH20-0857.

## MAPPRT

FUNCTION: To create, and optionally print, a file containing a map of entry points in the CMS nucleus.

| ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
      LA     1, PLIST
      SVC    X'CA'
PLIST DC     CL8'MAPPRT'
                  A
      DC     CL8' N '
                  C
                 ON
      DC     CL8' OFF'
                 NO
```

OPERATION: MAPPRT examines the parameter list and sets switches according to the type of file to be created and the location of the printout, if a printout was requested. If the parameter A was specified, the file CMS-NUC ALPHABET P1 is erased; the names of the entry points in the nucleus are then sorted into alphabetical order and written into the file CMS-NUC ALPHABET P1. If the parameter N was specified, the file CMS-NUC NUMERIC P1 is erased; the core locations of the entry points are then sorted into numeric order and written into the file CMS-NUC NUMERIC P1.

If neither A nor N were specified, MAPPRT assumes C, proceeds as for A, above, and then proceeds as for N, above. The file CMS-NUC ALPHANUM P1 is erased, and the contents of the ALPHABET file and the NUMERIC file are combined into the file CMS-NUC ALPHANUM P1.

MAPPRT then determines if the previously created file is to be printed or typed. If NO was specified, MAPPRT returns to the caller. If ON was specified, MAPPRT calls the PRINTF command to type out the file on the console and returns to the caller. If OFF was specified, it calls the OFFLINE PRINT command to print the file and returns to the caller.

Note: MAPPRT should only be called when IPL'ing 190 to obtain its CMS nucleus and after the LOGIN command. MAPPRT will expect the nucleus loader map to be at core positions 1E800 and continue downward to 1D000.

## MODMAP

FUNCTION: To type at the console typewriter the load map associated with the module specified by the MODMAP command.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
        LA     1, PLIST
        SVC    X'CA'
        DC     AL4 (error return)
         .
         .
         .
PLIST   DC     CL8'modmap'
        DC     CL8'        ' module filename
```

OPERATION: MODMAP first calls the STATE function to determine whether the file exists. If the file does not exist, MODMAP returns the error message 'FILE NOT FOUND' and branches back to the user. With confirmation that the file is there, MODMAP checks the number of logical records in the file to determine if a load map does exist with the module specified. (The load map is the last record of the module - normally the third record). If the module has less than three records, it was a transient or was created with the NOMAP option. If there is not a map, MODMAP types the message 'LOAD MAP UNAVAILABLE', and branches back through Register 14 to the user. If the third record is present, MODMAP sets up the module name specified in a PLIST and SVC's to LOADMOD the module into core. MODMAP next loads the address of NUCON to establish certain CMS parameters and addresses, reads and unpacks the third record, which is the load map. MODMAP then sets up the map information in a buffer and places the address of the buffer in a PLIST with the command call in the PLIST to TYPLIN. MODMAP then, via the TYPLIN function, prints the map onto the console typewriter and branches back to the user via Register 14. MODMAP is a transient routine.

## OSTAPE

FUNCTION: To enable users to read a tape consisting of 80 byte, unblocked records and create a CMS file from it. The PDS option is designed to read a tape produced by the OS utility IEBPTPCH and create a set of CMS files from it.

ATTRIBUTES: Disk-resident utility.

CALLING SEQUENCE:

```
        LA     R1, PLIST
        SVC    202
        DC     AL4 (ERROR)
         .
         .
         .
```

PLIST DC CL8'OSTAPE'

DC CL8$\begin{array}{l}\underline{\text{'SYSIN'}}\\ \text{'filetype'}\end{array}$

DC CL8$\begin{array}{l}\underline{\text{'OSTAPE'}}\\ \text{'filename'}\end{array}$

DC CL8'('

DC CL8$\begin{array}{l}\underline{\text{'NPDS'}}\\ \text{'PDS'}\end{array}$

DC CL8$\begin{array}{l}\underline{\text{'NCOL1'}}\\ \text{'COL1'}\end{array}$

DC CL8$\begin{array}{l}\underline{\text{'TAP2'}}\\ \text{'TAPx'}\end{array}$

DC CL8$\begin{array}{l}\underline{\text{'NEND'}}\\ \text{'END'}\end{array}$

DC CL8$\begin{array}{l}\underline{\text{'N MAXTEN'}}\\ \text{'MAXTEN'}\end{array}$

Note: The underlined alternate indicates default option.

OPERATION: OSTAPE sets flag bits either to the default setting or to the requested option setting. If a user filename, filetype, or tape unit is requested, these are saved in locations NAM2, NAM1, and TAPID. Tape records are read; the end-of-file flag is cleared after each read. If OS labels are on the tape, they are typed on the terminal and the next record is read.

If columns 2 - 8 contain 'MEMBER', the option bits are checked for the partitioned data set request. If there is a PDS request and the file is open, FINIS and LOGDISK are called to close it, the user is notified, and the program continues. Otherwise, STATE is called for the file - if it exists, ERASE is called. The open file bit is set on, and the records are brought in. If there is no PDS request, the field is ignored, the file opened, and WRBUF is called to write the record on disk. Succeeding records go directly to WRBUF until an END or MEND card is encountered.

The file is then closed by calls to CMS FINIS and CMS LOGDISK, the user is informed and the MAXTEN counter is updated and checked. If MAXTEN is requested and the limit is reached, the user is informed and the program is terminated. If the MAXTEN is not requested, the tape scans for the next file.

An encounter of two tape marks in a row will also terminate the program.

## SORT

FUNCTION: To sort records from one disk file to a second disk file in ascending order according to specified sort fields.

ATTRIBUTES: Disk-resident.

CALLING SEQUENCE:

```
        LA     1, PLIST
        SVC    202
         .
         .
         .
PLIST   DC     CL8'SORT'
        DC     CL8'filename1'
        DC     CL8'filetype1'
        DC     CL8'filename2'
        DC     CL8'filetype2'
```

OPERATIONS: SORT saves the filenames and filetypes and, after the field definitions are entered, calls STRINIT to set up storage and calculate the amount it has. After issuing the GETMAIN, it analyzes the SORT field parameter and then checks to see if the requested output file already exists. If it does, the user has the option of either erasing the old file, appending the new output, quitting, or entering a new filename and filetype.

If core is exceeded during SORT, the messages:

```
'*CORE OVERFLOW DURING SORT, LAST ITEM PROCESSED: XXX'
'*SIMULATING END-OF-INPUT'
```

are sent to the user and the command is terminated.

## STAT

FUNCTION: To type on the terminal pertinent disk statistics for a given disk, for all read-write disks, or for all disks.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
        LA     R1, PLIST     R1 must point to P-List as usual
        SVC    X'CA'
        DC     AL4 (ERROR)
```

ENTRY REQUIREMENTS:

R1 must point to STAT parameter-list:

```
      DS    0F
PLIST DC    CL8'STAT'
     [DC    CL8'     ']  Disk-mode, '*', '?', or omitted
     [DC    CL8'?    ']  ? = Optional Parameter
```

EXIT CONDITIONS:

Normal Return
R15 = 0
Error Returns (R15 values, with messages as shown):
R15 = 1 : INCORRECT "STAT" PARAMETER-LIST
R15 = 2 : **Z-DISK (CUU) NOT CURRENTLY LOGGED IN**
R15 = 3 : **NO READ-WRITE DISK(S) CURRENTLY LOGGED IN**

CALLS to OTHER ROUTINES:

ADTLKP, ADTNXT

CALLED BY: User

MACROS USED:

ADT, FVS

OPERATION: If a specific disk-mode letter is given to STAT (for example, STAT P, STAT T, STAT S, etc.), the disk statistics for that disk are typed on the terminal, if the disk is currently logged in. If there is no disk corresponding to the disk-mode given, error 1 is returned. If the disk is not currently logged in, error 2 (with the message filled in to include the disk-address and mode-letter) is returned.

If the disk mode-letter is omitted entirely (that is, the command is just "STAT"), then statistics are given on all currently logged-in read-write disks. (If none is currently logged in, error 3 is returned.)

If the disk mode-letter is an asterisk ("STAT *"), then the disk statistics for all currently logged-in disks, both read-write and read-only, are given.

The message typed on the terminal now includes the number of files currently represented in the in-core file directory. For a read-write disk, this also equals the number of files on the disk. For a read-only disk, however, there may be many more files on disk; the count is just that of the files currently available through the directory currently in core. (For example, STAT S would give the number of files in the SSTAT table, equivalent to the P2 files of the S-Disk.)

If a '?' is entered as the only parameter, a brief status of all currently logged-in disks is typed. This brief status gives disk labels, disk address, disk mode, and R/O if the disk is read-only.

If an additional parameter of '?' is given, added information is typed, specifically whether the disk is 2311 or 2314, and whether it is read-only or read-write.

If a user has been logging in several disks and wants to be sure which ones he has logged in at the moment, a call to 'STAT ?' will provide the clue needed as to which disks are indeed logged in. (This would also show the order of search.)

The STAT command is transient-disk-resident. The name of the text deck is 'STATDSK'; thus a procedure for generating a new module of 'STAT' would be as follows (or equivalent):

```
        LOAD      STATDSK    (TRANS TYPE
        GENMOD    STAT
```

The logic of STAT is simple. ADTLKP or ADTNXT is called to find the appropriate Active Disk Table block; if the flag bits indicate the disk is logged in the pertinent disk, statistics are simply converted to printable form and typed. Leading zeros are eliminated by shifting the typeout left for any leading zeros found, and adding a blank at the end, with the subsequent call to TYPLIN deleting the trailing blanks.

## TAPE

FUNCTION: To dump disk files to tape, to restore files that were dumped to tape back onto disk, to rewind a tape, to write an end-of-file mark on a tape, or to skip to the next end-of-file mark on a tape.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
        LA     1, PLIST
        SVC    X'CA'
```

REWIND:

```
PLIST DC    CL8'TAPE'
      DC    CL8'REWIND'
      DC    CL8'TAPn'       CMS device name (optional)
```

WRITEOF:

```
PLIST DC    CL8'TAPE'
      DC    CL8'WRITEOF'
      DC    CL8'n'          Write 'n' EOF marks (optional)
      DC    CL8'TAPn'       CMS device name (optional)
```

SKIP:

```
PLIST DC    CL8'TAPE'
      DC    CL8'SKIP'
      DC    CL8'n'          number of EOF marks (optional)
      DC    CL8'TAPn'       CMS device name (optional)
```

DUMP:

```
PLIST DC    CL8'TAPE'
      DC    CL8'DUMP'
      DC    CL8'filename' or CL8'*'
      DC    CL8'filetype' of CL8'*'
      DC    CL8'filemode'   default of 'P1' if omitted
      DC    CL8'TAPn'       CMS device name (optional)
```

LOAD:

```
PLIST DC    CL8'TAPE'
      DC    CL8'LOAD'
      DC    CL8'n'          stop after 'n' EOF marks (optional)
      DC    CL8'TAPn'       CMS device name (optional)
```

SCAN:

```
PLIST DC    CL8'TAPE'
      DC    CL8'SCAN'
      DC    CL8'n'          stop after 'n' EOF marks (optional)
      DC    CL8'TAPn'       CMS device name (optional)
```

SLOAD:

```
PLIST DC    CL8'TAPE'
      DC    CL8'SLOAD'
      DC    CL8'filename' or CL8'*'
      DC    CL8'filetype' or CL8'*'
      DC    CL8'n'          stop after 'n' EOF marks (optional)
      DC    CL8'TAPn'       CMS device name (optional)
```

Note: The default value for 'TAPn' is TAP2' and the default value for 'n' is'1'.

OPERATION: The operation of the TAPE command program depends on whether the calling program specifies REWIND, WRITEOF, SKIP, DUMP, LOAD, SCAN, or SLOAD.

REWIND: TAPE calls the TAPEIO function program to rewind the tape. It then returns (via SVCINT) to the calling program, which is usually INIT.

WRITEOF: TAPE calls the TAPEIO function program to write an end-of-file marker. It then returns to the calling program.

SKIP: TAPE repeatedly calls the TAPEIO function program to read successive records from the tape until an end-of-file marker is encountered. It then returns to the calling program.

DUMP: TAPE calls the FSTLKP function program to locate the file status table (FST) block for the file. TAPE then temporarily alters the FST block characteristics to 800-byte fixed-length records. TAPE then calls the RDBUF function program to read the first 800-byte block in the file into a buffer and the TAPEIO function program to write the data block from the buffer onto tape. TAPE repeats this procedure of calling RDBUF and TAPEIO for each data block in the file. When an end-of-file is reached, TAPE calls TAPEIO and writes a trailer record on the tape. The trailer record identifies the file and contains an N in the fifth byte, the last 20 bytes of the file, status table for the file in bytes 6 - 25, and the file designation in bytes 70 - 87. Finally, TAPE restores the FST block and calls the FINIS command program to close the file. It then returns to the calling program.

Note: Each data block is written, as a single tape record. The tape record is 805 bytes long. The first four bytes contain a code indicating that the record was produced by the TAPE command program. The next byte is zero, except for the trailer record. The remaining 800 bytes contain the data block.

LOAD: TAPE calls the ERASE command program to erase the file (if any) designated as (DISK) (TFILE) P3. Next it calls the TAPEIO function program to read the first record on the tape. (If the record was produced by the dump portion of TAPE, it will be 805 bytes in length and contain a code in the first four bytes and a data block in the last 800 bytes). TAPE then checks the code in the record. If invalid, TAPE issues a message to the effect that the tape is not in tape load format,and returns to the calling program. If the code is valid, TAPE determines if it is a trailer record (that is,one with N in the fifth byte). If not a trailer record, TAPE calls the WRBUF function program to write the data block contained in the record into a file designated as (DISK) (TFILE) P3. Then TAPE calls the TAPEIO function program to read the next record from the tape, which it processes in the same manner.

If the tape record read is a trailer record, TAPE calls the FINIS command program to close the disk file ((DISK) (TFILE) P3) created from the first tape file. It then calls the ERASE command program.to erase the file (if any) that has the same designation as the tape file just converted to a disk file. Next, TAPE calls the FSTLKP function program to locate the file status table for the disk file just created. (This is again (DISK) (TFILE) P3.) Subsequently, TAPE overlays the last 20 bytes of the file status table, (except for the first chain link address) with the corresponding data from the trailer record and moves the filename, filetype, and date last updated for the tape file from the trailer record into the corresponding locations in the FST block. The directory is then updated with a call to UPDISK. This completes the conversion of the first file on the tape to a disk file, and TAPE calls the TYPLIN function program to type a message at the terminal to the effect that the file has been loaded. TAPE processes the next file on the tape in a similar manner. When the 'n'th end-of-file on the tape is encountered, TAPE returns to the calling program. If 'n' was not specified, TAPE returns to the calling program when the first end-of-file is reached.

SCAN: TAPE sets SCANSWT to disable disk operation and to enable end-of-file printout and then branches into the code for TAPE LOAD. The effect of this command is to list the contents of a tape (including end-of-file marks) at the terminal until the 'n'th end-of-file mark is encountered. The default value of 'n' is one.

SLOAD: TAPE searches for the file whose filename and filetype were specified in the parameter list. When the matching file is found, SLDSWT is set, and TAPE branches into the TAPE LOAD code to copy the file from tape to disk. SLDSWT enables control to return to the SLOAD coding after the file is copied. If neither the filename nor filetype was '*' TAPE returns control to the calling program, otherwise the above search and load procedure is continued until the 'n'th end-of-file mark is encountered. Control is returned to the user when the 'n'th end-of-file mark is encountered even if no files were copied from tape to disk. The default value of 'n' is one.

## TAPEIO

FUNCTION: To (1) read or write tape records, (2) rewind the tape, and (3) write an end-of-file marker on tape.

ATTRIBUTES: Disk resident, transient

Note: For a detailed explanation on TAPEIO, see the write-up on the TAPEIO function.

## TAPRINT

FUNCTION: To print the contents of a tape containing assembler or FORTRAN LISTING files.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
       LA     1, PLIST
       SVC    X'CA'
       .
       .
       .
PLIST  DC     CL8'TAPRINT'
      [DC     CL8'        ']          symbolic tape name
```

OPERATION: TAPRINT receives the symbolic name of the tape to be printed from the parameter list and inserts it into the calling sequence to the TAPEIO function program. It then calls the TAPEIO function program to read the first record from the tape. This record contains 10 blocked print line images. Next, TAPRINT repeatedly calls the PRINTR function program to print each of the 10 print line images on the printer. TAPRINT repeats this process of reading a record from tape and printing the 10 print line images contained therein until it encounters an end-of-file on the tape. At this time, it calls the TYPLIN function program to type a message at the terminal signaling the end-of-file. It then prints the next file on the tape in a similar manner. When two consecutive end-of-files are encountered, meaning that the end of the tape has been reached, TAPRINT calls the TYPLIN function program to indicate this. It then calls the TAPEIO function program to rewind the tape, calls the CLOSIO command program to close printer operations, and returns to the calling program.

Note: If the calling program does not provide a symbolic tape name, TAP2 is assumed.

## TPCOPY

FUNCTION: To copy tape files.

ATTRIBUTES: Disk resident.

CALLING SEQUENCE:

```
        LA    1, PLIST
        SVC   202
        .
        .
        .
PLIST   DC    CL8'TPCOPY'
              (CL8'TAPi'
        DC    {
              (CL8'*'             -TAP1
              (CL8'TAP0'
        DC    {
              (CL8'*'             -TAP2
              (CL8'n'
        DC    {
              (CL8'*'             -1
              (CL8'yes'
        DC    {
              (CL8'*'             -no
```

OPERATION: TPCOPY calls SVCFREE to get free storage and then analyzes the parameter list looking for defaults and errors such as the same unit for both input and output, and a file number less than 1 and greater than 9.

It uses TAPEIO to read tape input records, and calculates the actual length and number of records. This information is given to the user if the file summary option has been selected.

Error messages on read and write tape errors will be passed to the user by TPCOPY.

## WRTAPE

FUNCTION: To write a disk file onto tape.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
        LA    1, PLIST
        SVC   X'CA'
        .
        .
        .
PLIST   DC    CL8'WRTAPE'
        DC    CL8'        '      filename
        DC    CL8'        '      filetype
        DC    CL2'                filemode — defaults to P
        DC    CL8'        '      blocking factor — defaults to 10
        DC    CL3'EOF'           defaults to no-eof
```

EXIT CONDITIONS:

Normal return: R15 = 0

Error returns:

1. TAPE ERROR
2. PARAMETER ERROR
3. BLOCK SIZE TOO LARGE (Byte Count)
4. FILE NOT FOUND
5. FATAL ERROR FROM STATE

CALLS TO OTHER ROUTINES:

STAFF, RDBUF, FINIS, TAPEIO

CALLED BY:

User.

OPERATION: WRTAPE checks the parameter list to determine if the mode has been specified. If not, a P mode is assumed. Checking is then done to determine if a blocking factor was specified. If not, a default blocking factor of 10 is used. If the End-Of-File parameter has been entered, an appropriate flag is turned on.

WRTAPE then calls STATE to locate the file. Once located, an output PLIST is set up from the STATEFST. If the file is a listing file all ASA carriage control characters are converted to machine code.

WRTAPE repeatedly calls RDBUF to read the file and block the records according to the desired blocking factor and calls the TAPEIO function program to write the blocked records to tape, until an end-of-file is encountered. At this time, WRTAPE checks the EOF flag to determine if an END-OF-FILE was requested. If EOF was requested, WRTAPE calls 'TAPEIO' to write an END-OF-FILE.

Note: WRTAPE writes to tape device TAP2.

CONTROL COMMANDS

The commands CPFUNCTN, IPL, KE, KO, KT, KX, LOGIN, LOGOUT, RELEASE, RT, SYN, and VSET control the user's virtual machine environment. These are described in detail on the following pages.

## BLIP

FUNCTION: To enable the user to specify the terminal two-second time count character.

Note: For information on BLIP, see the VSET writeup.

## CHARDEF

FUNCTION: To enable the user to change the default characters for character delete, line delete, EDIT backspace and logical tab, and to specify the hexadecimal representation of characters.

Note: For information on CHARDEF, see the VSET writeup.

## CPFUNCTN

FUNCTION: Transmits console function commands to CP-67 without leaving the CMS virtual machine environment. Permits incorporation of CP console functions in EXEC files and programs.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
        LA      1,PLIST
        SVC     X'CA'
        .
        .
        .
PLIST   DC      CL8'CPFUNCTN'
        DC      CL8'NOMSG'       optional
        DC      C'   'command string'
```

where 'string' is a CP-67 console function

OPERATION: CPFUNCTN calls CONWAIT to drain any stacked terminal output. It then moves 'command string' into a buffer, places the address of the buffer in register 1 and the byte count for the string in register 2. A DIAGNOSE instruction X'83120008' is then executed, transmitting to CP-67 the desired console function. CP-67 returns to CMS the following error codes:

| | |
|---|---|
| 0 | Command accepted |
| 4 | INVALID CP REQUEST |
| 8 | BAD ARGUMENT |

For codes 4 and 8 the indicated messages are typed by CMS. Any other nonzero error code is dependent upon the particular function. At this time, only LINK returns additional codes.

The NOMSG option inhibits typing of the messages for codes 4 and 8. It is intended primarily for use by calls from other programs. Other CP messages will be typed by CP directly.

## IPL

FUNCTION: To initial program load the CMS nucleus into core.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
        LA      1,PLIST
        SVC     X'CA'
        .
        .
        .
        .
PLIST   DC      CL8'IPL'
       [DC      CL8'xxx']       IPL-Device (optional)
```

OPERATION: If the IPL-Device was specified, and the user is running in a virtual machine, the parameters are passed to CP (via a diagnose instruction), and GP performs the IPL.

If the IPL-Device was specified but CMS is running on a real machine, the device number given is used for the disk to be used for the IPL sequence, as described later.

If the IPL-Device is omitted from the parameter list, a halfword "IPLDEV" in NUCON is examined. If this = 0, that is an indication that CMS is being run on a virtual machine via IPLCMS — that is, IPL by name; for this case, CP is invoked by the diagnose instruction to execute IPL CMS.

If CP is not called upon to do the IPL sequence, then CMS uses the given IPL-Device (if present) or the contents of IPLDEV for the disk from which the IPL is to be performed, and then IPL's from that disk as follows:

The IPL sequence from cylinder 0, track 0, records 1 and 2 of the disk is read in. The IPL sequence then reads in the IPLDISK function program from the system disk. IPLDISK then reads in the CMS nucleus.

## KE

FUNCTION: To increase or decrease the length of lines being typed to the terminal.

ATTRIBUTES: Nucleus resident
Imbedded in CONSI

CALLING SEQUENCE:

This routine is not called. CONSI tests each input line from an ATTN interrupt for the presence of KE, KT, RT, KX, and KO.

OPERATIONS: Hit the ATTN key twice to open the console, and enter KE. When CONSI receives control from IOINT, it picks up the KE from the input line, and checks to see if a number has been entered for a specific line length to be typed. If a number was specified, the column limit is set, the KE flag is set, and the ATTN buffer is released. If no number was specified, a default column limit of 72 is assumed. CONSI then returns to IOINT.

## KO

FUNCTION: To kill overrides.

ATTRIBUTES: Nucleus resident.
Imbedded in 'CONSI'

CALLING SEQUENCE:

This routine, along with KT and RT, is not formally called. CONSI tests each input line from an ATTN interrupt for the presence of KT, KO, RT, and KX.

OPERATION: Hit ATTN key twice to open console and enter KO. When CONSI receives control from IOINT, it picks up the KO from the input line and sets the KLOVER flag, releases the ATTN buffer by a call to FRET, and returns to IOINT.

## KT

FUNCTION: To kill typing at the user terminal.

ATTRIBUTES: Nucleus resident.
Imbedded in CONSI.

CALLING SEQUENCE:

This routine, along with KO and RT, is not called. CONSI tests each input line from an ATTN interrupt for the presence of KE, KT, RT, KX, and KO.

OPERATION: Hit the ATTN key twice to open console, and enter KT. When CONSI receives control from IOINT, it picks up the KT from the input line, sets the KT flag, releases the ATTN buffer by a call to FRET, and returns to IOINT.

## KX

FUNCTION: To kill execution during a running program.

ATTRIBUTES: Nucleus resident.
Entry point - Killex in LOG

CALLING SEQUENCE:

```
        Extrn KILLEX
             or
        DC  V(KILLEX)
```

OPERATION: Comes here (via CONSI) if user hits ATTN twice and types KX. (Also, could come here on purpose from DEBUG) Does the following:

1. Calls DESBUF to clear out any stacked-up console I/O.

2. Clears KT and KE flags to ensure typing of console messages.

3. Sets kill-overrides flag to finish up in case overrides set.

4. Calls FINIS*** to close out any open files.

5. Calls CONWAIT to wait for console I/O.

6. Calls CLOSIO to finish reader, printer, and punch.

7. Calls IPL to load a fresh copy of CMS.

## LINEND

FUNCTION: To enable the user to define his own logical linend character in place of the default character of #.

Note: For information on LINEND see the VSET writeup.

## LOGIN

FUNCTION: To bring into core the User File Directory for a given disk (e.g., 191, 192), setting up the necessary information in the Active Disk Table for the given disk mode (e.g., P, T).

ATTRIBUTES: Transient, reentrant

CALLING SEQUENCE:

```
        LA     R1, PLIST           R1 must point to P-list as usual
        SVC    X'CA'
        DC     AL4(ERROR)
```

ENTRY REQUIREMENTS:

R1 must point to LOGIN parameter list:

```
        DS     0F
PLIST   DC     CL8'LOGIN'
        DC     CL8'      '         Additional parameters as needed
        .                                   (See examples of valid plists)
        .
        .
        DC     CL8'      '
        DC     X'FFFFFFFF'         Signifies end of P-List
```

EXIT CONDITIONS:

Normal Return

R15 = C

Error Returns

R15 nonzero        (See "ERROR RETURNS")

CALLS TO OTHER ROUTINES:

ADTLKP, ADTNXT, FREE, FRET, READFST, READMFD, RELUFD

CALLED BY (where known):

INIT, or User (from terminal or EXEC file)

MACROS USED:

ADT, FVS

EXAMPLES OF VALID LOGIN PARAMETER LISTS:

Note: In the following examples, "ccu" stands for a hexadecimal disk-address (e.g., 192, etc.) and "m" or "n" stands for the given Disk-Mode (for example, P).

1. Valid Parameter Lists to LOGIN an entire Disk:

```
LOGIN
LOGIN    (NOTYPE
LOGIN    (NOPROF NOTYPE
LOGIN    (NOPROF  NO TYPE
LOGIN    ccu
LOGIN    ccu  (NOPROF NOTYPE
LOGIN    ccu  m
LOGIN    ccu  m  (NOTYPE NOPROF
```

Example: LOGIN 196 (NOPROF

2. Valid Parameter Lists to LOGIN NO-UFD from a DISK:

```
LOGIN    (NO_UFD
LOGIN    (NO-UFD
LOGIN    ccu (NO_UFD
LOGIN    ccu (NO-UFD
LOGIN    ccu m (NO_UFD
LOGIN    ccu m (NO-UFD
```

Example: LOGIN 192 T (NO-UFD

3. Valid Parameter Lists to LOGIN selected files from a disk "m" known to be read-only:

```
LOGIN    ccu  m  filename   (options
LOGIN    ccu  m  filename   filetype   (options
LOGIN    ccu  m  filename   filetype   filemode   (options
```

Example: LOGIN 194 C * TEXT P1

4. Valid Parameter Lists to LOGIN from a disk "m" as a read-only extension of another disk "x" (accessing either all or selected files):

```
LOGIN   ccu  m, x
LOGIN   ccu  m, x  (options
LOGIN   ccu  m, x  filename  (options
LOGIN   ccu  m, x  filename  filetype  (options
LOGIN   ccu  m, x  filename  filetype  filemode  (options
```

Example: LOGIN 193 A,P CON*UPDG* (NOTYPE

ERROR RETURNS (R15 value at exit, with message as shown):

Note: In the following messages, "m" or "n " stands for a disk-mode (e.g., P or T or A etc.) and ccu stands for a disk-address (e.g., 191 or 192 etc.).

| Error No. | Message | Comments |
|---|---|---|
| 1. | ** m (ccu) DEVICE ERROR ** | File Directory is Unreadable |
| 2. | ** m (ccu) NOT ATTACHED ** | |
| 3. | ** m (ccu) DEVICE ERROR ** | ccu not recognizable device-type |
| 4. | ** m (ccu) R/O — CANNOT LOGIN NO-UFD** | |
| 5. | ** m (ccu) DEVICE ERROR ** | ccu is in old 2311 format no longer supported. |
| 6. | LOG006: PARAMETER ERROR | Error in LOGIN P-List |
| 7. | ** m (ccu) ACCESSED AS n-DISK (R/W)** | |
| 8. | ** m (ccu) NOT ACCESSED — 0 FILES ** | |

OTHER RESPONSES (given with normal return if appropriate but omitted if the NOTYPE option was given):

```
ccu REPLACES m (ccu)
m  (ccu)  R/O
ccu ALSO = n-DISK
ccu  m    RELEASED
ccu  m,x RELEASED
```

OPERATION: LOGIN is the command which is used to bring into core the User File Directory (UFD) for the user's P-Disk or any other disk (except the S-DISK, which is logged in earlier by INITSYS & READFST).

In the CMS initialization process, if the user's first command is anything other than LOGIN or FORMAT P ALL, the LOGIN command is invoked automatically to log in a user's files from his P-Disk. Then, if a PROFILE EXEC exists in the user's directory, this is executed, followed by the first command typed in. If the user wishes to bypass the automatic call of his PROFILE EXEC, his first command must be LOGIN (NOPROF. This logs in his files as usual, but bypasses the call to PROFILE EXEC.

If a LOGIN is issued at any later time, for any disk, no such automatic call to PROFILE EXEC is made — it is effective only on the first command, as described above.

If desired, the PROFILE EXEC on a user's P-Disk can contain EXEC commands to login other disks.

As shown under Valid Parameter Lists above, LOGIN can be called in several ways to bring in the file directory for a given disk. However, these break down to two major cases:

Case 1 LOGIN without the NO-UFD option brings in the directory of existing files for the given disk. If the disk is read-write, the directory of all existing files is brought into core (regardless of any remaining operands in the LOGIN parameter list). If the disk is read-only, the directory of only those files specified as operands in the LOGIN P-List is brought into core; if no specific filenames, filetypes, or filemodes were specified, then the directory of all files (except P0 files) is brought in.

Case 2 "LOGIN (NO-UFD" (or equivalent) brings in the necessary file directory information from the disk-resident file directory, but omits the FST entries of pre-existing files. All necessary tables and disk counters are cleared, giving the user a clean directory for the given disk as if he had called FORMAT or had erased all files. LOGIN (NO-UFD or equivalent is valid only for a read-write disk. Error 4 is returned if it is attempted on a read-only disk.

LOGIN checks the parameter-list for the existence of a ccu disk-address and a possible mode-letter.

If the ccu is provided, the value of the hexadecimal number is computed; leading zeroes are permissible, but the computed value must be nonzero and less than X'6FF'. If provided and legitimate, its value is used in place of the default disk-address (i.e., 191) in the NUCON table.

If a disk mode is given, ADTLKP is called to find the matching Active Disk Table for the given letter. (If the diskmode is omitted, the P-Disk is used as a default.) If a read-only extension is also given (e.g., LOGIN 193 A, P or such) ADTLKP is again called, to ensure that an active disk table exists for the disk given by the extension-mode-letter.

A check is made to ensure that the disk to be logged in is not already logged in as another read-write disk. If so, error 7 is returned, with an error message to the user.

If a disk to be logged in will replace another disk which is currently logged in, then a message indicating that this will occur is typed unless the NOTYPE option has been specified. If, e.g., a disk addressed as 196 is about to replace a currently logged in 191 P-Disk, the message would be:

```
196 REPLACES P (191)
```

After the parameter list has been checked for errors and special options, LOGIN then proceeds as follows, for Case 1 or 2 as described earlier.

Case 1 — LOGIN existing files:

1. RELUFD is called to clear all pertinent information in the old active disk table.

2. If the disk to be logged in will be a read-only extension of another (or of itself), the read-only flag-bit in ADTFLG1 is then set to force the disk to be read-only.

3. READFST is then called to bring in the entire or partial directory of the disk.

4. If this disk was read-only (either from setting the flag-bit from above or from obtaining an error 4 from READFST), a check is made to see if any files at all were accessed; if not, an error 8 is returned, RELUFD is called to clear the Active Disk Table (ADT) entry, and the disk is not logged in. If read-only and at least one file is accessible, then the read-only response is given (unless the NOTYPE option was specified by the caller).

5. If the disk is to be a read-only extension of another, the extension-mode-letter is stored in the ADTMX slot in the ADT block for the disk just logged in. Also, another bit (ADTROX) is set in the ADTFLG1 flagbyte of the ADT for the other disk, to indicate that it has at least one read-only extension.

6. A check is made to see if the disk just logged in is also logged in as any other disk(s). If yes, and the newly logged in disk is read-write, the other disk(s) are released via RELUFD and a message is typed (unless NOTYPE was specified) to indicate the release of the ccu as the other disk(s). If yes, and the newly logged in disk is read-only, a message is typed (unless NOTYPE was specified) indicating that ccu is also logged in as the other mode letter.

LOGIN for case 1 is finished. The disk is logged in, an extension-mode-letter stored if appropriate, informative messages (if any) have been typed, and the disk is ready to use.

Case 2 — LOGIN (NO-UFD:

1. RELUFD is called to clear all pertinent information in the old active disk table.

2. Then READMFD is called to bring in all pertinent information on the disk except the FST hyperblocks containing the FST entries (which would have been brought in if READFST had been called).

3. If an error is returned by READMFD it is returned to the caller of LOGIN, with the error message as shown above. Note that if the disk is read-only, this is treated as an error condition.

4. Upon successful return from READMFD, LOGIN obtains an 816-byte block from free storage for the first FST hyperblock, clears it, and initializes the ADTRES reserve-count and all other necessary pointers and counters in the ADT.

5. The QMSK brought in by READMFD is now cleared, and the appropriate disk counts recomputed and stored to reflect a clean disk.

6. The QQMSK brought in by READMFD is also cleared.

CAUTION: "LOGIN (NO-UFD" (or equivalent) should only be used when all old files on a disk (if any) are to be discarded. It is equivalent, in effect, to FORMAT'ing the disk, or erasing all files thereon, but is much faster and more efficient. Note, however, that if a user issues "LOGIN (NO-UFD" by mistake, the file directory on the given disk has purposely not been updated by LOGIN (no call to UPDISK is made); and therefore the user can recover his files by immediately issuing a LOGIN command for the disk without the NO-UFD option.

Notes:

1. If any disk is logged in as a read-only disk, for whatever reason, only files having a mode-number of 1-6 are accessed. For a read-write disk, all files are accessible, from mode numbers 0-6. Therefore, P0 files on any disk can be considered "Private" to the user who has read-write access to the disk, and no one having read-only access to the disk can reference them.

2. If the first user command is 'LOGIN', then INIT & LOGIN (which work together on the first command issued at the terminal) accept that first command as is, and do not issue any implied automatic login of the user's normal P-Disk (191). Therefore, if the user wishes to login his P-Disk and then immediately login another disk in addition, he should issue a specific login command for his P-Disk first (e.g. LOGIN 191), and then the other LOGIN command (perhaps LOGIN 193 A, P or whatever). This could of course be conveniently done utilizing the CMS linend character, e.g.:

```
LOGIN 191#LOGIN 193 A, P
```

3. If the user wishes to login a disk in a read-only status which is normally read-write, this can be accomplished by making the disk a read-only extension of itself, e.g.:

   ```
   LOGIN 191 P,P
   LOGIN 193 A,A
         etc.
   ```

4. If the user does not wish to login any user disks at all with his first command, this can be accomplished by issuing the command:

   ```
   LOGIN NODISK
   ```

   This is effectively handled as a no-operation by LOGIN when called by INIT to handle the first user command.

## LOGOUT

FUNCTION: To log a user out of the system.

ATTRIBUTES: Nucleus resident

CALLING SEQUENCE:

```
        LA     1,PLIST
        SVC    X'CA'
        .
        .
        .
PLIST   DC     CL8'LOGOUT'
        DC     CL8'      '            Note: An additional command can be added
                                      to the LOGOUT command here, if desired.
                                      For example, LISTF or STAT would be
                                      permissible.

        DC     X'FF'                  (Must follow LOGOUT or 'added-on'
                                      command.)
```

OPERATION: The CMS LOGOUT command calls LOGDISK CHANGE to close any files that may be open and ensure that all file directories are updated. CLOSIO is called to ensure that reader, printer, and punch operations are finished. At this point, any added-on command (such as LISTF or STAT) is called, ignoring any possible error-return. Then the PRNFINAL entry in the CMS timer program is called to compute and print the cumulative CPU time used during the terminal session. Finally, LOGOUT loads a PSW, causing the system to enter the WAIT state with no interrupts enabled. When running under CP/67, the Control Program is reached by this loading of the PSW, and CP/67 indicates its readiness for a new command by typing "CP ENTERED, REQUEST, PLEASE".

The CMS LOGOUT command can conveniently be stacked as the last command in a string of CMS commands to be executed. Furthermore, if an appended command CP LOGOUT is added, forming the command LOGOUT CP LOGOUT, the user will then log out of CP after he has done all his work and logged out of CMS.

## RELEASE

FUNCTION: To release all core-resident tables pertaining to a given disk when it is no longer needed; and to detach the disk, as an option.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
         LA      R1, PLIST
         SVC     X'CA'
         DC      AL4(ERROR)
```

ENTRY REQUIREMENTS:

R1 must point to the Parameter List as usual:

```
         DS      0F

PLIST    DC      CL8'RELEASE'
         DC      CL8'       '          Disk-address (for example, 192)
         DC      CL8'       '          Disk mode (for example, T)
        [DC      CL8'(DETACH)']        Optional if disk to be detached
```

EXIT CONDITIONS:

Normal Return

R15 = 0

Error Returns

R15 = 1 : Invalid RELEASE Parameter List (Disk address not hex number up to 6FF, disk mode not letter from A to Z, etc.)

R15 = 2 : No Active-Disk-Table found for given mode (ADTLKP did not find disk corresponding to disk mode letter given)

R15 = 3 : Disk-Number does not match Device-Table

CALLS TO OTHER ROUTINES:

RELUFD, CPFUNCTN

CALLED BY:

User

MACROS USED:

ADT, FVS

OPERATION: The parameter list is checked for errors. The disk-address must be a hex number (digits from 0 to 9 and letters from A to F, with a value no more than X'6FF'). The disk mode must be alphabetic. It is not legal to RELEASE the S-Disk. Error 1 is returned if any errors are detected. ADTLKP is called to find the Active Disk Table (ADT) block. If an error occurs from ADTLKP, error 2 is returned. If ADTLKP found the ADT block, the given disk-address is checked against the disk number in the NUCON table that is pointed to by the ADT block. If it does not match, error 3 is returned. If all checks so far are correct, RELUFD is called for this disk, and the ADTFRO and ADTFRW read-only and read-write flag bits in the ADTFLG1 flag byte in the ADT block are also cleared, to signal that the disk referenced by the ADT block is not logged in. (It is not an error condition if nothing was logged in when RELEASE was called.)

If the parameter list specified at least "(DET", signifying the "(DETACH)" option is desired, CPFUNCTN is called upon (with a suitable parameter list) to detach the disk, and the error-code from CPFUNCTN is passed back to the caller.

Installation Note: RELEASE is a transient disk resident command, GENMOD'ed with a copy of CPFUNCTN (CMSCONF).

Other Notes: RELEASE is normally called when a disk that has been used for a while is no longer needed, so that its tables will no longer take room in core and so that its file directory cannot be confused with others. When it is desired to log in a read-write disk which is already logged in as another, however, RELEASE must be called first, since it is not practical to have a read-write disk logged in as two separate disks.

## RT

FUNCTION: To resume typing at the terminal.

ATTRIBUTES: Nucleus resident.
Imbedded in CONSI

CALLING SEQUENCE:

This routine, along with KT and KO, is not formally called. CONSI tests each input line from an Attn interrupt for the presence of KT, KO, RT, and KX.

OPERATION: Hit ATTN key twice to open keyboard and enter RT. When CONSI receives control from IOINT, it picks up the RT from the input line and clears the KT flag, releases the ATTN buffer by a call to FRET, and returns to IOINT.

## SYN

FUNCTION: The SYN command allows the user to specify his own command names to be used with or in place of the standard system command names.

ATTRIBUTES: Disk resident, transient

CALLING SEQUENCE:

```
SYN [FILENAME FILETYPE FILEMODE (OPTION1 . . . OPTION-N)]
```

If filemode is omitted, mode of '*' is assumed (P, T, S-Disk)

If filetype is omitted, filetype of 'SYN' is assumed.

If filename is omitted, no user synonyms are set up. (Only the options are processed.)

Options (if any) must be preceded by left-paren, and are as follows:

P prints the standard system abbreviations and user synonyms currently defined.

PUSER prints only the user synonyms currently defined.

STD specifies standard system abbreviations are to be used. This is the default value.

NOSTD specifies standard system abbreviations are not to be used.

MIN use minimum number of characters specified to identify commands. The default value.

EXACT use exact number of characters specified to identify commands.

CLEAR clears any previously defined synonym table set up by SYN.

OPERATION: The SYN command permits user-defined names to be used either alone or in conjunction with the standard CMS system abbreviations — that is, it permits the user to modify the command names acceptable to his own environment.

User-defined synonyms are located in a file identified as "filename filetype filemode" in the format shown in Note 2. If filetype is omitted, a filetype of SYN is assumed; if filemode is omitted, a mode of * is assumed, meaning the P, T, or S-Disk. If no file is specified, no user-defined synonyms are set up, and the system abbreviations are used in the manner defined by the specified options.

All options (if any) are specified between a pair of parentheses. (The right paren, however, may be omitted.) The default options are STD — use standard abbreviations, and MIN — allow a minimum number of characters to represent a command. NOSTD will flag the standard system abbreviations as unusable; MIN accepts abbreviations as long as the minimum number of characters specified in the abbreviation table are present; EXACT accepts only the entry as specified.

SYN can also be used to print out the list of synonyms and abbreviations currently acceptable.

Notes:

1. SYN with no additional parameter is the same as SYN (P); that is, it types a listing of system and user abbreviations currently in effect.

2. The user synonym file "filename filetype filemode" consists of 80-byte fixed-length records in freeform format with columns 73 to 80 ignored. The format for each record is:

| | | |
|---|---|---|
| system-command | user-synonym | count |

where count is the number of characters necessary for the synonym to be accepted. If omitted, the entire synonym must be entered. SYN builds a table from the contents of this file to use for command synonyms.

EXAMPLES:

```
SYN
SYN (CLEAR P)
SYN MYOWN (PUSER)
SYN MY ABBS (NOSTD PUSER)
SYN OUR ABBS SY (NOSTD
```

ERROR CODES (With Messages)

1 INCORRECT 'SYN' PARAMETER LIST

2 NO ABBREVIATIONS AT ALL ("ABBREV" NOT IN NUCLEUS)

3 GIVEN USER SYNONYM FILE NOT FOUND

4 USER SYNONYM FILE BAD (MUST BE 80-BYTE FIXED RECORDS)

5 FAULTY DATA IN USER SYNONYM FILE

*** DISK ERROR READING USER SYNONYM FILE

***Note — Error-code from RDBUF returned to caller.

OTHER RESPONSES:

a. SYSTEM ABBREVIATIONS FLAGGED "NOT IN USE"

A request has been made to print the system abbreviations while a previous NOSTD is in effect.

b. NO USER SYNONYM TABLE CURRENTLY IN USE

A request has been made to print the user-defined synonym table while no such table has been defined by SYN command.

## VSET

FUNCTION: VSET allows the user to control different aspects of his environment at his console. VSET BLIP controls the character designated to notify the user of every two CPU seconds of execution time; VSET CHARDEF controls the definitions for logical symbols, such as line delete, character delete, backspace, and tab characters, and the hexadecimal representation of defined characters; VSET IMPEX controls the order of search for commands: VSET LDRTELS controls the number of pages of core used for loader tables: VSET LINEND controls the definition for the logical line-end character: VSET RDYMSG controls the length of the error and ready messages typed by CMS: VSET REDTYPE controls the color of the CMS error messages: and VSET RELPAG controls the releasing of pages of core upon command completion.

ATTRIBUTES: Disk resident, transient Module=CMSCARE

ENTRY REQUIREMENTS:

R1 must point to parameter list. The calling sequences are shown with the different VSET functions.

EXIT CONDITIONS:

R15 = return-code

All other registers restored.

CALLED BY:

User.

## VSET BLIP

FUNCTION: To enable the user to specify the terminal two-second time count character.

CALLING SEQUENCE:

```
        DS    0F
PLIST   DC    CL8'VSET'
        DC    CL8'BLIP'
        DC    CL8'char'  BLIP character(s) or (OFF)
        DC    CL8'nn'    number of characters in char
```

CALLS TO OTHER ROUTINES:

None

OPERATION: VSET BLIP first determines if the machine is running in Batch mode. If it is, an exit back to the caller is taken. If not, it checks for a BLIP OFF request. If no blip is desired, it sets the timer to a larger positive number (effectively turning off the blip) and exits to the caller. If a blip is desired and it is not the nonprinting default, the desired character is stored in TIMCHAR and the count is moved to the timer CCW string.

## VSET CHARDEF

FUNCTION: To enable the user to change the default characters for logical symbols and to specify the hexadecimal representation of characters.

CALLING SEQUENCE:

For logical symbols:

```
      DS   0F
PLIST DC   CL8'VSET'
      DC   CL8'CHARDEF'
      DC   CL8'type'        B, C, L, or T
      DC   CL8'character'
```

For character representation:

```
      DS   0F
PLIST DC   CL8'VSET'
      DC   CL8'CHARDEF'
      DC   CL8'type'        IN, OU, or IO
      DC   CL8'character'
      DC   CL8'hexcode'
```

CALLS TO OTHER RCUTINES:

FREE, FRET

OPERATION: If the request is for delete character, DELSYM indexes the delete table by a 4 in R4 to insert the new character. If the request is for delete line, the index is 8 for the same operation. For EDIT backspace, it places the address of SVCSECT in R4 and inserts the character at displacement 290. For EDIT logical tab, it uses the same operation with a displacement of 291. In all cases, the return to the caller is through R14. The revised table is kept in free storage, and its address is placed in a filled-in word in the CONGEN section of the NUCON table.

Prior to exit from CHARDEF a check is made to see if the free storage table (as revised) is identical to the standard table. If yes, the free storage table is returned via FRET and its address in CONGEN is cleared.

"CHARDEF IN T XX" causes a table to be set up for use by WAITRD in preference to the usual upper-case translation table, with the desired 'XX' equivalent used as an argument byte corresponding to 'T'.

"CHARDEF OU U XX" causes a table to be set up for use by TYPLIN or TYPE which is used for output translation.

"CHARDEF IO V XX" causes both of the above to be set up.

"CHARDEF IN, OU, or IO" with no additional parameters causes the tables to be returned to free storage via FRET and their use discontinued.

## VSET IMPEX

CALLING SEQUENCE:

```
      DS    0F
PLIST DC    CL8'VSET'
      DC    CL8'IMPEX'
      DC    CL8'      '        ON or OFF
```

CALLS TO OTHER ROUTINES:

None

OPERATION: VSET IMPEX OFF causes a bit to be set in the NUCON table which inhibits the implied exec procedure, when examined by INIT for commands entered from the terminal.

VSET IMPEX ON resets this bit, causing the implied exec procedure to be in effect.

## VSET IDRTBLS

CALLING SEQUENCE:

```
      DS    0F
PLIST DC    CL8'VSET'
      DC    CL8'LDRTBLS'
      DC    CL8'nn'            optional number of pages desired
```

CALLS TO OTHER ROUTINES:

None

OPERATION: VSET LDRTBLS checks to see if any free storage is in use by CMS; if so, the loader tables cannot be revised. If not, a check is made to see if the loader tables can be revised to the number of pages desired. If yes, the revised number of pages of loader tables is stored in the NUCON table.

If no operand is specified, the number of pages of loader tables is obtained from NUCON, and typed.

VSET LINEND

FUNCTION: To enable the user to define his own logical linend character in place of the default character of

CALLING SEQUENCE:

```
      DS    0F
PLIST DC    CL8'VSET'
      DC    CL8'LINEND'
      DC    CL8'char'        optional character for line-end
```

CALLS TO OTHER ROUTINES:

FREE, FRET

OPERATION: BREAK stores the new linend character in BRKCHR, and returns to the caller through register 14.

VSET RDYMSG

CALLING SEQUENCE:

```
      DS    0F
PLIST DC    CL8'VSET'
      DC    CL8'RDYMSG'
      DC    CL8'      '      ON or OFF
```

CALLS TO OTHER ROUTINES:

None

OPERATION: VSET RDYMSG OFF causes a bit to be set in the NUCON table, which, when examined by INIT, causes the abbreviated ready and error messages to be typed.

VSET RDYMSG ON clears this bit, causing the full-length ready and error messages to be typed.

### VSET REDTYPE

CALLING SEQUENCE:

```
      DS    0F
PLIST DC    CL8'VSET'
      DC    CL8'REDTYPE'
      DC    CL8'      '       ON or OFF
```

CALLS TO OTHER ROUTINES:

None

OPERATION: VSET REDTYPE ON sets a bit to the CONGEN part of the NUCON table which causes typeouts to be typed in red if CONWRITE or TYPE is called with an 'R' color code, as for error messages.

VSET REDTYPE OFF clears this bit, so that all messages will be in black.

### VSET RELPAG

CALLING SEQUENCE:

```
      DS    0F
PLIST DC    CL8'VSET'
      DC    CL8'RELPAG'
      DC    CL8'      '       ON or OFF
```

CALLS TO OTHER ROUTINES:

None

OPERATION: VSET RELPAG OFF causes a bit to be reset in the NUCON table, which Operation: when examined by INIT, causes the release-pages feature to be bypassed.

VSET RELPAG ON sets this bit, enabling the release-pages feature.

LIBRARIES

CMS provides two types of libraries — macro and text (subroutine). Macro libraries are searched for missing macros during assemblies. Text libraries are searched for missing subroutines or undefined filenames during the LOAD, USE, or REUSE commands.

To generate, add to, delete, or replace in macro or text libraries, the MACLIB and TXTLIB commands are used. These are described in detail in the following section.

## MACLIB

FUNCTION: To generate a macro library, to add macros to an existing library, and to list the dictionary of an existing macro library.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
      LA   1,PLIST
      SVC  X'CA'
      .
      .
      .
      DS   0D
PLIST DC   CL8'MACLIB'
              ( COMP  )
              ( GEN   )
              ( ADD   )
      DC   CL8'{ LIST  }'
              ( PRINT )
              ( REP   )
              ( DEL   )
      DC   CL8'        '  macro library name
      DC   CL8'        '  filename 1
           .
           .
           .
      DC   CL8'        '  filename N
```

OPERATION: The operation of the MACLIB command program depends on whether the calling program specifies GEN, ADD, LIST, COMP, PRINT, REP, or DEL.

GEN: MACLIB calls the ERASE command program to erase the file (if any) that is identically designated as the macro library to be created. MACLIB then calls the WRBUF function program to write a dummy, 80-byte record as the first record in the macro library. This dummy record will later be replaced by a macro directory descriptor record. Next, MACLIB initializes the index, which corresponds to the item number, to one. Then it calls the STATE function program to locate the file status table for the first macro file — filetype must be ASP360 or COPY. MACLIB next calls the RDBUF function program to read the first record in the first macro file, calls the WRBUF function program to write it into the macro library being created, and increments the index. After writing the first (or any) record, the action taken by MACLIB depends on the nature of the record.

If the record is a macro header record (that is, it contains the characters "MACRO" starting in column 10), MACLIB saves the current index value for subsequent use in calculating the size (that is, the number of items) of the macro. Then, it stores the index value in the appropriate entry in the macro dictionary (refer to "TABLE/ RECORD FORMATS" later in this section), reads the next record, which is the prototype record, obtains the macro name from that record, moves the name to the appropriate entry in the macro dictionary, writes the prototype record into the macro library, increments the index, and reads and processes the next record.

If the record read is either a comment or an element óf the body of the macro, MACLIB merely reads and processes the next record.

If the record is a macro trailer record (that is, it contains the characters "MEND" starting in column 10), MACLIB calculates the size of the macro, places the size in the appropriate entry in the macro dictionary, increments a pointer to point to the next entry in the macro dictionary, and returns to read and processes the next entry.

MACLIB repeats this process for all records in the first macro file. When an end-of-file is encountered, it calls the FINIS command program to close that file, and processes the next macro file similarly.

When all macro files are processed, MACLIB writes the macro dictionary out at the end of the macro library, overlays the dummy record at the start of the macro library with a dictionary header record (refers to "TABLE/RECORD FORMATS"), closes the newly created macro library, and returns (via SVCINT) to the calling program, which is usually INIT.

Note: Throughout its processing, MACLIB checks to ensure that the records in each macro definition are in correct sequence. If they are not, it signals the error by means of a terminal message (error-code 4), and returns to the calling program.

ADD: MACLIB calls the STATE function program to determine if the macro library to which the macros are to be added exists. If it does not, it signals the error and returns to the calling program. If the macro library exists, MACLIB calls the RDBUF function program to read the dictionary header record into main storage so that it can get the starting location of the macro dictionary. It then sets the read pointer in the file status table to point to the start of the macro dictionary and repeatedly calls the RDBUF function program to read the macro dictionary into main storage. MACLIB next calls the FINIS command program to close the macro library. Having closed the library, MACLIB calls the POINT function program to set the write pointer to the start of the old macro dictionary. Next, MACLIB sets a pointer to the next available location in the macro dictionary and then proceeds to add the macros in the same manner as it does if GEN is specified.

DEL: The specified macro name is deleted from the macro library dictionary.

REP: The macro filename (with a filetype of ASP360 or COPY) is used as the name of the macro to be replaced. If the replacing macro has more items than already exist in the library, the macro is DELeted from the library and then ADDed to the end of the MACLIB. Otherwise, if the replacing macro is not larger than the existing macro, it will occupy exactly the same position within the library as the replaced macro.

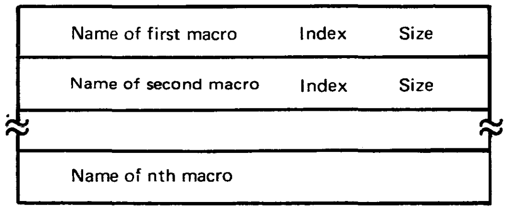
LIST: MACLIB reads the macro dictionary into main storage as it does for ADD. It then calls the TYPLIN function program to print a heading for the list. Next, MACLIB obtains the first entry in the dictionary, moves the name, index, and size to a buffer, and calls TYPLIN to print the contents of the buffer at the terminal. MACLIB repeats this for each entry in the dictionary. When all entries are processed, MACLIB returns to the calling program.

PRINT: The same function as LIST is performed with the following results: a file identified as "libname" MAP P1 will be written onto the user-disk area and automatically printed onto the OFFLINE PRINTer.

TABLE/RECORD FORMATS: The formats of the macro dictionary and the dictionary header record are described below.

MACRO DICTIONARY: In the macro dictionary (see Figure 30) each entry is 12 bytes in length and contains three fields. The name field (8 bytes) contains the name of the macro. The index field (2 bytes) indicates where, within the macro library, the first record (item) in the macro is located. The index field (2 bytes) is expressed as an item number. The size field (8 bytes) contains the size of the macro. It is expressed in terms of the number of items in the macro.

DICTIONARY HEADER RECORD: The dictionary header record (see Figure 31) defines the location and size of the macro dictionary. It is an 80-byte record and contains three meaningful fields. The first field (bytes 1-6) contains the characters 'MACLIB'. The second field (bytes 7 and 8) is a pointer to the start of the macro dictionary. It is expressed as an item number. The third field (bytes 11 and 12) contains the size of the macro dictionary (in bytes).



| | | |
|---|---|---|
| Name of first macro | Index | Size |
| Name of second macro | Index | Size |
| | | |
| Name of nth macro | | |

Figure 30. Macro Dictionary Format

| Bytes | Contents |
|---|---|
| 1-6 | MACLIB |
| 7-8 | Pointer to start of macro dictionary |
| 9-10 | Size of macro dictionary |
| 11-80 | Not used |

Figure 31. Dictionary Header Record Format

## TXTLIB

FUNCTION: To create a text library, to add text files to an existing text library, to create a disk file that lists the control section and entry point names in a text library, or to type at the terminal the control section and entry point names in a text library.

ATTRIBUTES: Disk resident

CALLING SEQUENCE:

```
        LA    1,PLIST
        SVC   X'CA'
        .
        .
        .
        DS    0D
PLIST   DC    CL8'TXTLIB'
                  (GENERATE)
                  |ADD      |
        DC    CL8'< PRINT    >  '
                  |LIST     |
                  (DELETE   )
        DC    CL8'       '         library name
        DC    CL8'       '         filenamel/csectnamel
        .
        .
        .
        DC    CL8'       '         filenameN/csectnameN
```

OPERATION: The operation of TXTLIB depends on whether the calling program specifies GENERATE, ADD, PRINT, or LIST.

GENERATE: TXTLIB calls the SVCFREE function program to obtain a block of free storage for use as a work area. It then calls the ERASE command program to erase the existing text library (if any) with the same name as the one to be created. Next, TXTLIB initializes the index and saves it for subsequent use to calculate the size of the first control section. Then TXTLIB calls the STATE function program to determine if the first input text file specified exists. If it does not, TXTLIB signals an error by means of a terminal message and processes the next input text file.

If the text file exists, TXTLIB calls the RDBUF function program to read the first record in the file, increments the index, and calls the WRBUF function program to write the record into the text library. Subsequent processing of this record (or of any record read from an input file) depends upon its nature.

If the record is not an ESD, LDT, or END record, TXTLIB merely reads and processes the next record in the input file.

If the record is a ESD record, TXTLIB obtains the first ESD data item in the record. If this data item is for a section definition (SD) or label definition (LD), TXTLIB puts the associated name into the next available entry in the text library dictionary. Next, it places the saved index value, which indicates the relative location within the library in terms of items (that is, 80-byte records) of the start of the control section, into the dictionary entry. It then obtains and similarly processes the next ESD data item in the record. If the obtained data item is neither for a section definition nor for a label definition, TXTLIB skips it and obtains the next data item. When all the data items in the ESD record are processed, TXTLIB reads and processes the next record in the input file. (During the processing of the ESD record, the name and index fields of one or more entries in the dictionary may be filled in. The size field of these entries, which indicates the size of the corresponding control section in terms of number of items (that is, 80-byte records), will be filled in when the next LDT record is encountered. Also, the index fields in the dictionary entries for the section definition and all label definitions of a control section will contain the same value.)

If the record read is an LDT record, TXTLIB computes the size of the control section in terms of number of items, stores the size in the successive entries in the dictionary that were partially filled when the preceding ESD record(s) was processed, saves the current index value for use in computing the size of the next control section, and reads and processes the next record in the input text file.

If the record read is an END record, TXTLIB generates an LDT record from the information on the END record, writes the LDT record into the text library, and processes the LDT record as previously described.

When an end-of-file on the input text file is encountered, TXTLIB calls the FINIS command program to close that file, obtains the next file, and adds its contents to the text library in a similar fashion.

When the last input file has been processed, TXTLIB successively calls the WRBUF function program to write the dictionary (80 bytes at a time) at the end of the text library, constructs a dictionary header record, and writes the header record at the beginning of the text library. (TXTLIB has left room at the beginning of the library for the header record.) Finally, TXTLIB calls the FINIS command program to close the text library, calls the SVCFRET function program to release the free storage used as a work area, and returns (via SVCINT) to the calling program, which is usually INIT.

OVERFLOW: The maximum number of entries allowed in the dictionary is 1000. Each time an ESD card is encountered the total is checked. If the number exceeds 1000, the pointer to the end of the file is set back to the end of the last complete CSECT, the dictionary is written out, and the program completes in the normal way often issuing a message to indicate which CSECT caused an overflow.

ADD: TXTLIB calls the STATE function program to determine whether the text library to be added to exists. If it does not, TXTLIB types a message at the terminal to that effect and returns to the calling program. If the library exists, TXTLIB calls the RDBUF function program to read the header record into main storage. From the header record, TXTLIB obtains the location and size of the dictionary. It again calls RDBUF to read the entire dictionary into main storage. Then, TXTLIB sets the write pointer to the location of the dictionary in the text library. This is done so that the dictionary will be written over when the new text files are added to the library. Next, TXTLIB calls the FINIS command program to close the library. It then adds the new text files to the end of the library by following a procedure identical to that for GENERATE.

PRINT: TXTLIB calls the STATE function program to determine whether the text library whose control section and entry point names are to be placed into a disk file exists. If it does not, TXTLIB types a message at the terminal to that effect and returns to the calling program. If the library exists, TXTLIB reads the header record into main storage and then reads the dictionary into main storage. Next, it calls the FINIS command program to close the library. TXTLIB then calls the ERASE command program to erase the previously created disk file (that is, the MAP file), if one exists. Next, TXTLIB calls the WRBUF function program to write a heading into the new MAP file being created. Subsequently, TXTLIB repeatedly calls the WRBUF function program to write a record into the new MAP file for each entry in the dictionary. If the dictionary entry represents the start of a control section, the corresponding record consists of the name of the control section, the location of the control section within the text library in terms of an index value, and the size of the control section in terms of number of items (that is, 80-byte records). If the dictionary entry is for an entry point (that is, a label definition), the corresponding record consists only of the entry point name. When all dictionary entries are processed, TXTLIB writes a record containing a count of the number of entries in the dictionary into the MAP file. It then calls the FINIS command program to close the MAP file, releases the free storage previously obtained, and returns to the calling program.

LIST: The processing performed by TXTLIB if LIST is specified is essentially the same as that for PRINT. However, in this case, TXTLIB calls the TYPLIN function program, rather than WRBUF, to type the records produced for the entries in the dictionary at the terminal.

DELETE: TXTLIB takes a filename of filetype TXTLIB and a list of CSECT names in the TXTLIB file to be deleted. TXTLIB scans the dictionary and copies everything not found in the list of CSECT names to be deleted into a new dummy file (.DUMMY TXTLIB). A new dictionary is created for this new TXTLIB file. When the operation is complete, the original TXTLIB file is erased and the file .DUMMY TXTLIB has its name altered to that of the original file.

Note: If a CSECT name occurs twice within the TXTLIB file, only the first occurrence is deleted. A CSECT name may be entered into the argument list two or more times to delete two or more CSECT's with the same CSECT name.

TABLE/RECORD FORMATS: The formats of the text library dictionary and the dictionary header record are described below.

TEXT LIBRARY DICTIONARY: This dictionary has room for 1000 entries. Each entry is associated with either a control section name (section definition ESD item) or an entry point name (label definition ESD item). An entry is 12 bytes in length and contains three fields. The name field (8 bytes) contains either the control section or entry point name. The index field (2 bytes) contains the location of the corresponding control section from the start of the text library. This field is expressed as an item number. The size field (2 bytes) contains the size of the control section in terms of number of items (i.e., 80-byte records). The text library dictionary is illustrated in Figure 32.

DICTIONARY HEADER RECORD: The dictionary header record defines the location and size of the text library dictionary. It is an 80-byte record and contains four meaningful fields. The first field (bytes 65-68) is a pointer to the start of the text library dictionary. It is expressed as an item number. The second field (bytes 69-72) contains the size of the macro dictionary (in bytes). The third field (bytes 73-76) contains the number of items in use and the fourth (bytes 77-80) the number of items not in use. The dictionary header record is illustrated in Figure 33.

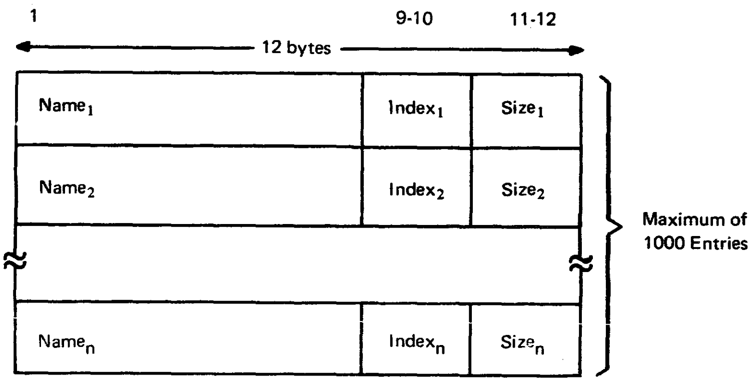


Figure 32. Text Library Dictionary Format

| Bytes | Contents |
|---|---|
| 1-64 | Not used |
| 65-68 | Pointer to start of dictionary |
| 69-72 | Size of dictionary - 12 in bytes |
| 73-76 | #of items |
| 77-80 | #free |

Figure 33. Text Library Dictionary Header Record Format

## FORTRAN Subroutines

In addition to the standard program library, the following subroutines found in the FORTRAN library SYSLIB TXTLIB are provided to aid the terminal user in more effectively utilizing CP/CMS:

| Entry Point | Filename of Source Deck |
|---|---|
| CPNMON/CPNMOF | IXCFREM |
| NLSTON/NLSTOF | IXCFREM |
| DEFINE | IXCDEF |
| DSDSET | IXCDSD |
| ERASE | IXCRENM |
| GETPAR | IXCGETP |
| LOGDSK | IXCRENM |
| RENAME | IXCRENM |
| REREAD | IXCRERD |
| TRAP | IXCBPTRP |
| BLIP/TRAP | IXCBPTRP |

### CPNMON/CPNMOF (alias NLSTON/NLSTOF)

Purpose: This routine provides the ability to enter namelist data from the terminal in a free format mode.

Usage:

Before issuing any free format reads you must issue:

```
Call CPNMON, or
Call NLSTON
```

To return to standard namelist format, issue:

```
Call CPNMOF, or
Call NLISTOF
```

Regular namelist data is entered in the following manner:

```
b̸&list1b̸a=1, b=2, c=3, 4, 5b̸&end
```

Free format data for the same variables would be entered:

```
1,2,3,4,5
```

Data must be entered in sequential order, unlike regular namelist mode in which the order is of no consequence, as the variables appear in the specific namelist referenced. In addition, the namelist must be exhausted before attempting to read another namelist, since if a second read to a second namelist is attempted before the first namelist has been exhausted, the data intended for the second namelist will be placed in the first read locations.

The call to CPNMON or NLSTON results in a dynamic overlay of FORTRAN IHCNAMEL calls to IHCFIOCS, with a transfer to IHCFREM, the free format routine. At execution of the namelist read, control is passed to the IHCFREM at this point. IHCFREM then goes to IHCFIOCS and picks up the record. It then constructs a standard namelist record, which it passes back to IHCNAMEL at the point where processing normally continues.

External References:

IHCNAMEL
IHCFIOCS

Entry Points:

CPNMON
NLSTON
CPNMOF
NLSTOF

## DEFINE

Purpose: 1) To permit the use of sequential access disk files as direct access files.
2) To tie a CMS filename-type to a FORTRAN DSRN.

Usage:

CALL DEFINE (DSRN, NAME, type, recno, recsiz)
where DSRN = FORTRAN file number
NAME = filename
TYPE = filetype
RECNO= integer or integer variable containing location of the record number to be read or written
RECSIZ= maximum record size
then issue a normal sequential read or write, not a direct access read.

Constraints:

1) Must be fixed-length records
2) Must be a disk file
3) If a data set reference number is assigned to a new filename and filetype, the record number for the old filename filetype is lost.

External References:

1) Rereadv
2) FIOCS
3) Deftblv
4) IBCOM

## DSDSET

Purpose: To enable users to alter the data set default specifications for all the defined units in the csect 'IHCVATBL' with the exception of DSRN's 5,6 & 7.

Usage:

Call DSDSET (DSRN, BLKSIZE, TYPE, LKECL)
where

DSRN = the data set reference number of the data set that is to be modified

BLKSIZE= is the new blocksize to be used as default on the data set

TYPE = code number from 1 - 5 to change the default RECFM
1 - fixed
2 - fixed/blocked
3 - variable
4 - variable/blocked
5 - undefined

LRECL = is the logical record length to be used as a default on the data set. It is optional in the parameter list, but must exist in correct relationship to the BLKSIZE if the RECFM is defined as fixed/blocked.

External References:

IHCUATBL
IHCFCOMH

## ERASE

Purpose: To erase a file from within a FORTRAN program.

Usage: Call ERASE (FNAME, FTYPE, <FMODE>)
FNAME is filename
FTYPE is filetype
FMODE is filemode

Note: This subroutine uses the CMS function ERASE to erase the file named in the call.

### GETPAR

Purpose: To enable the user to obtain parameters entered from the terminal at program load time.

Usage:

> CALL GETPAR ( name , item-no , 'RITE' , 7LAST )
> where name is the variable (real *8) that is to be set to the values of the parameter number 'ITEM-NO'.
> ITEM-NO is the location (integer *4) in the initial parameter list of the parameter desired; it may be any non-negative value, with 0 indicating the parameter pointed to by GR1 on initial entry (that is the program name); if item number exceeds the number of parameters in the string, no parameter is passed and control will pass to statement 'lost' if &LAST was specified. 'RITE is an optional literal which causes the current parameter to be right-justified in its double word field, with leading blanks supplied; this is useful in reading numeric parameters; and, finally, &LAST is an optional statement label to which control is passed if item number exceeds the number of variables.

Notes: The program load parameter is accessed by indirect addressing:

```
L R3, = V(CMSFORTR) gets original FORTRAN area
L R3, 4(0, R3) gets original CMS save area
L R3, 24(0, R3) gets Reg 1
```

### LOGDSK

Purpose: To close all open files and write the user file directory on P-disk.

Usage: CALL LOGDSK

Notes: This subroutine uses the CMS function LOGDSK to close all open files and write the user file directory on disk.

### RENAME

Purpose: To change file identifiers within a FORTRAN program.

Usage: Call RENAME (OLDFN, OLDFT, NEWFN, NEWFT)

OLDFN is old file name
OLDFT is old file type
NEWFN is new file name
NEWFT is new file type

Notes: This subroutine uses the CMS function ALTER to make the desired changes.

## REREAD

Purpose: To enable the user to reread a record which has already been read into core.

Usage: Call REREAD ( DSRN, BLKSIZE )

DSRN may be any number between 0 and 4 or 8 and 99.
Default value is 99.

BLKSIZE may be any integer greater than zero.
Default value is 140 bytes.

The reread unit is a block of core which is obtained when the call "REREAD" is issued. The user then can use the reread unit any time he chooses by first writing data into the block, rewinding the unit and then reading it under what ever format control he chooses. Failure to rewind the unit will result in an end-of-file condition because the pointer will be positioned at the end of the buffer.

Possible uses include testing a code on input and branching to different reread routines based on the codes.

```
Ex. Call reread (25, 80)

          Read (1, 10) list
          If co180 eq. x go to 40
          If co180 eq. y go to 50

40 write (25, 10) list

          rewind 25
          read (25, 20) list

50 write (25, 10) list

          rewind 25
          read (25, 30) list
```

External References: VF10CS
F10CS#
IBCOM#
DEFTBLV

Illegal usage: DSRN 5, 6, 7

negative DSRN
Floating point blocksize
unformatted records

## TRAP

Purpose: To enable the user to specify a program interrupt location.

Usage: Call TRAP (statement number).

Note: The TRAP flag is set so that on a program interrupt you will go to your routine and not into DEBUG.

## BLIP/TRAP

Purpose: Provides the ability to use the CMS functions BLIP and TRAP from within a FORTRAN program. For a discussion of TRAP, see TRAP.

Usage:

1. Call BLIP (character string, count)
2. Call BLIP (0)
3. Call BLIP (character)

The first way enables a user to specify a printing blip character or character string - up to 8 characters - with a count of how many there are in the string.

The second way gives just a character with a count of one.

The third way resets the blip character to the non-printing CMS character.

Note: Since the blip character types on the terminal every 2 seconds of CPU time, this can be used to roughly measure time spent in various parts of a complex program by calling different blip characters for each part to be measured.

## SECTION 5: SERVICE PROGRAM DESCRIPTIONS

This section describes the programs that provide the monitor and command programs with the services provided. These include the function, loader, and processing programs. The following text contains detailed descriptions of the various CMS function programs.

### STORAGE MANAGEMENT FUNCTION PROGRAMS

The following text describes the operation of the routines which manage free storage; they include FREE, FRET, SVCFREE, and SVCFRET.

### FREE

FUNCTION: To allocate free storage

CALLING SEQUENCE:

```
LA     0, NDBLWDS        number of double words wanted in register 0
L      15, =A(FREE)
BALR   14, 15
```

OPERATION: Refer to "Main Storage Management" in Section 2.

COMMENTS: FREE returns the address of the allocated storage in register 1. Registers 0 and 2 thru 15 are preserved.

### FRET

FUNCTION: To release storage that is no longer needed by a program to free storage.

CALLING SEQUENCE:

```
LA     0, NDBLWDS        number of double words to be released in
                         register 0.
LA     1, BLOCK          starting address of storage to be released
                         in register 1.
L      15, = A(FRET)

BALR   14, 15
```

OPERATION: Refer to "Main Storage Management" in Section 2.

COMMENTS: All 16 registers are preserved.

## SVCFREE

FUNCTION: To allocate free storage

CALLING SEQUENCE:

```
        LA     1, PLIST
        SVC    X'CA'
        .
        .
        .
PLIST   DC     CL8'SVCFREE'
        DC     F'  '                number of double words wanted
        DC     A(*)                 address of allocated storage returned to
                                    caller here
```

OPERATION: SVCFREE calls the FREE function program that will allocate the requested storage and return its starting address to SVCFREE. SVCFREE then loads the starting address into the last entry in the parameter list and returns (via SVCINT) to the caller.

## SVCFRET

FUNCTION: To return storage that is no longer needed by a program to free storage.

CALLING SEQUENCE:

```
        LA     1, PLIST
        SVC    X'CA'
        .
        .
        .
PLIST   DC     CL8'SVCFRET'
        DC     F'  '                number of double words to be returned
        DC     A(      )            address of first double word
```

OPERATION: SVCFRET calls the FRET function program to return the storage to free storage. It then returns (via SVCINT) to the calling program.

## FILE MANAGEMENT MACROS

Several macros are used for convenience by the various file management and related function programs and routines. Appendix B shows which programs and routines use each particular macro.

These I/O macros are as follows:

| Macro | Description or Use |
|---|---|
| ADT | Shows one Active Disk Table entry (See Figure 34.) |
| AFT | Shows one Active File Table entry (See Figure 35.) |
| DIOSCT | Macro corresponding to DIOSECT (used by RDTK-WRTK-DSKERR) (See Figure 36.) |
| FSTB | Shows one FST Entry (See Figure 37.) |
| FVS | Macro corresponding to most of FVS C-Sect (heavily used to reference I/O tables and routines) (See Figure 38.) |

CMS "ADT" MACRO

(showing form of an Active Disk Table Block)

```
         MACRO
         ADT
*
* ACTIVE DISK TABLE BLOCK
*
ADTSECT  DSECT
*
*        NEEDED FOR READ-ONLY DISKS AND READ-WRITE DISKS
*
ADTID    DS    CL6               DISK-IDENTIFIER (LABEL)
ADTFLG3  DS    1X                -RESERVED FOR FUTURE USE-
ADTFTYP  DS    1X                FILETYPE FLAG-BYTE
ADTPTR   DS    1A                POINTER TO NEXT ADT BLOCK IN CHAIN
ADTDTA   DS    1A                DEVICE TABLE ADDRESS IN NUCON
ADTFDA   DS    1A                FILE DIRECTORY (PSTAT) ADDRESS
ADTMFDN  DS    1F                NUMBER DBL-WORDS IN MFD
ADTMFDA  DS    1A                MASTER FILE DIRECTORY ADDRESS
ADTHBCT  DS    1F                FST HYPERBLOCK COUNT
ADTFSTC  DS    1F                NUMBER OF FST 40-BYTE ENTRIES (FILES)
ADTCHBA  DS    1A                POINTER TO CURRENT FST HYPERBLOCK
ADTCFST  DS    1F                DISPLACEMENT OF CURRENT FST ENTRY
ADT1ST   DS    1F                DISP. OF 1ST WORD IN BIT-MASK WITH 'HOLE'
ADTNUM   DS    1F                NUMBER OF RECORDS (NUMTRKS)
ADTUSED  DS    1F                NUMBER OF RECORDS IN USE (QTUSEDP)
ADTLEFT  DS    1F                NUMBER OF RECORDS LEFT (QTLEFTP)
ADTLAST  DS    1F                DISP. OF LAST NONZERO BYTE IN BIT-MASK
ADTCYL   DS    1F                NUMBER OF CYLINDERS ON DISK (NUMCYLP)
ADTM     DS    1C                MODE LETTER (P,T,S,A,B,C, ETC.)
ADTMX    DS    1C                EXTENSION-OF-MODE LETTER (P,T,S, ETC.)
ADTFLG1  DS    1X                FIRST FLAG-BYTE
ADTFLG2  DS    1X                SECOND FLAG BYTE
*
ADT2ND   DS    0D
*
*        NEEDED JUST FOR READ-WRITE DISKS
*
ADTMSK   DS    1A                800-BYTE (PQMSK) BIT-MASK ADDRESS
ADTQQM   DS    1A                200-BYTE (PQQMSK) BIT-MASK ADDRESS
ADTPQM1  DS    1F                PQMSIZ = NO. BYTES IN PQMSK > 215
ADTPQM2  DS    1F                PQMNUM = NO. 800 BYTE-REC FOR PQMSK
ADTPQM3  DS    1F                RQNUM = NO. DBL-WORDS IN PQMSK
ADTLHBA  DS    1A                POINTER TO LAST FST HYPER-BLOCK
ADTLFST  DS    1F                DISP. OF LAST FST IN LAST HYPER-BLOCK
ADTNACW  DS    1H                NUMBER OF ACTIVE WRITE FILES - HALFWORD
ADTRES   DS    1H                RESERVE-COUNT (RESRVCNT) - HALFWORD
*
ADTLBM   EQU   ADT2ND-ADTSECT    LENGTH OF MINIMUM ADT BLOCK (BYTES)
ADTLDM   EQU   ADTLBM/8          LENGTH OF MINIMUM ADT BLOCK IN DBL-WORDS
*
ADTLB    EQU   *-ADTSECT         LENGTH OF FULL ADT BLOCK (BYTES)
ADTLD    EQU   (ADTLB+7)/8       LENGTH OF FULL ADT BLOCK IN DBL-WORDS
```

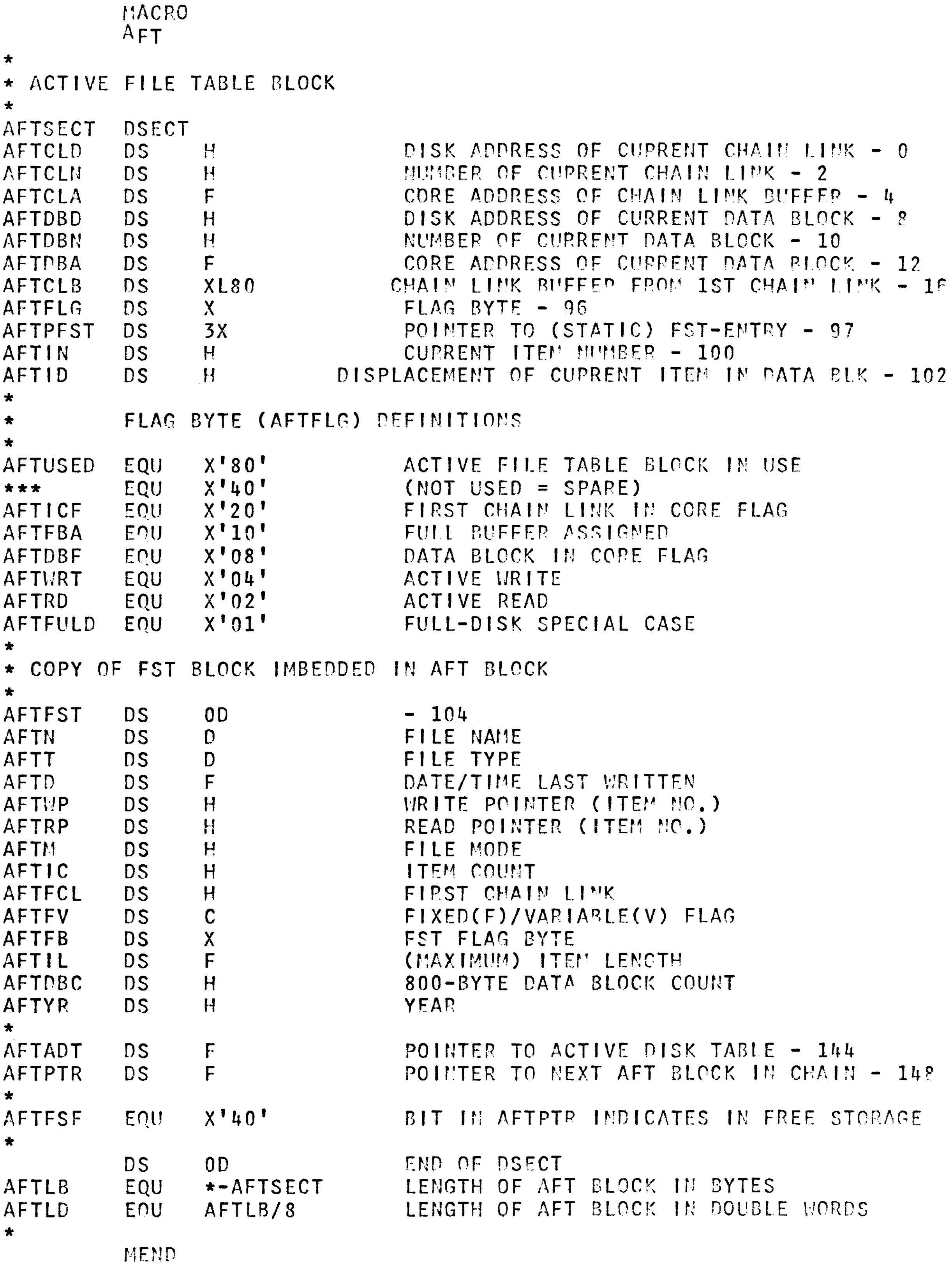
Figure 34. CMS ADT macro (sheet 1 of 2)

## CMS "ADT" MACRO

## (continued)

```
  *         FIRST FLAG-BYTE (ADTFLG1) DEFINITIONS
  *
  ADTFSF    EQU   X'80'          ADT BLOCK IN FREE STORAGE
  ADTFRO    EQU   X'40'          READ-ONLY DISK (ATTACHED & READY)
  ADTFRW    EQU   X'20'          READ-WRITE DISK (ATTACHED & READY)
  ADTFFSTF  EQU   X'10'          1ST FST HYPERBLOCK IS IN FREE STORAGE
  ADTFFSTV  EQU   X'08'          FST HYPERBLOCKS ARE OF VARYING LENGTH
  ADTFQQF   EQU   X'04'          200-BYTE QQMSK IS IN FREE STORAGE
| ADTROX    EQU   X' 02'         THIS DISK HAS READ-ONLY EXTENSION (S)
  ADTFMIN   EQU   X'01'          ADT BLOCK IS MINIMUM SIZE
  *
  *         SECOND FLAG-BYTE (ADTFLG2) DEFINITIONS
  *
  ADTFMFD   EQU   X'80'          MFD IS IN CORE
  ADTFALNM  EQU   X'40'          ALL FILENAMES ARE IN CORE
  ADTFALTY  EQU   X'20'          ALL FILETYPES ARE IN CORE
| ADTFMDRO  EQU   X' 10'         MODES 1-6 ARE IN CORE
| ADTFALMD  EQU   ADTFMDRO+X' 08'  ALL MODES (0-6) ARE IN CORE
  ADTFALUF  EQU   ADTFMFD+ADTFALNM+ADTFALTY+ADTFALMD  ALL UFD IS IN CORE
  ADTWMSG   EQU   X'04' READ-ONLY WARNING MESSAGE HAS BEEN GIVEN BY WRBUF
  *
  *         OTHER PARAMETERS
  *
  ADTRL     EQU   800            LOGICAL RECORD LENGTH
  ADTML     EQU   5              MAXIMUM BIT MASK LENGTH - IN RECORDS
  *
  *         NUCON DEVICE TABLE OFFSETS
  *
  DTAD      EQU   0              DEVICE NUMBER
  DTADT     EQU   3              DEVICE TYPE BYTE
  DTAS      EQU   4              SYMBOLIC DEVICE NAME
            MEND
```

Figure 34. CMS ADT macro (sheet 2 of 2)



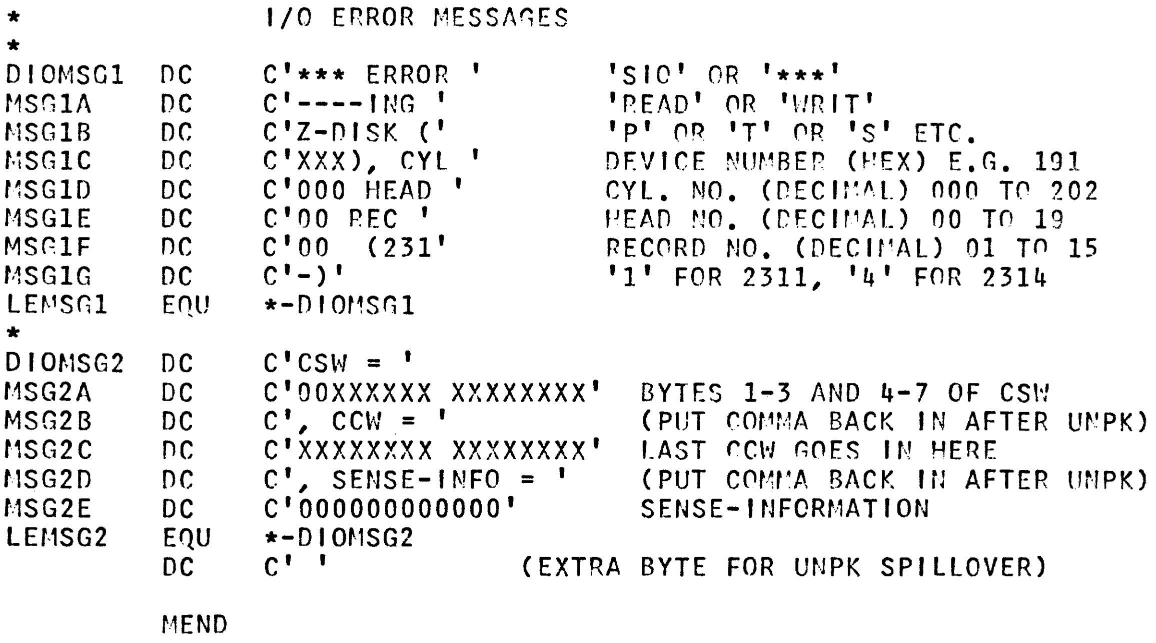
```
         MACRO
         AFT
*
* ACTIVE FILE TABLE BLOCK
*
AFTSECT  DSECT
AFTCLD   DS    H                DISK ADDRESS OF CURRENT CHAIN LINK - 0
AFTCLN   DS    H                NUMBER OF CURRENT CHAIN LINK - 2
AFTCLA   DS    F                CORE ADDRESS OF CHAIN LINK BUFFER - 4
AFTDBD   DS    H                DISK ADDRESS OF CURRENT DATA BLOCK - 8
AFTDBN   DS    H                NUMBER OF CURRENT DATA BLOCK - 10
AFTDBA   DS    F                CORE ADDRESS OF CURRENT DATA BLOCK - 12
AFTCLB   DS    XL80            CHAIN LINK BUFFER FROM 1ST CHAIN LINK - 16
AFTFLG   DS    X                FLAG BYTE - 96
AFTPFST  DS    3X               POINTER TO (STATIC) FST-ENTRY - 97
AFTIN    DS    H                CURRENT ITEM NUMBER - 100
AFTID    DS    H           DISPLACEMENT OF CURRENT ITEM IN DATA BLK - 102
*
*        FLAG BYTE (AFTFLG) DEFINITIONS
*
AFTUSED  EQU   X'80'            ACTIVE FILE TABLE BLOCK IN USE
***      EQU   X'40'            (NOT USED = SPARE)
AFTICF   EQU   X'20'            FIRST CHAIN LINK IN CORE FLAG
AFTFBA   EQU   X'10'            FULL BUFFER ASSIGNED
AFTDBF   EQU   X'08'            DATA BLOCK IN CORE FLAG
AFTWRT   EQU   X'04'            ACTIVE WRITE
AFTRD    EQU   X'02'            ACTIVE READ
AFTFULD  EQU   X'01'            FULL-DISK SPECIAL CASE
*
* COPY OF FST BLOCK IMBEDDED IN AFT BLOCK
*
AFTFST   DS    0D               - 104
AFTN     DS    D                FILE NAME
AFTT     DS    D                FILE TYPE
AFTD     DS    F                DATE/TIME LAST WRITTEN
AFTWP    DS    H                WRITE POINTER (ITEM NO.)
AFTRP    DS    H                READ POINTER (ITEM NO.)
AFTM     DS    H                FILE MODE
AFTIC    DS    H                ITEM COUNT
AFTFCL   DS    H                FIRST CHAIN LINK
AFTFV    DS    C                FIXED(F)/VARIABLE(V) FLAG
AFTFB    DS    X                FST FLAG BYTE
AFTIL    DS    F                (MAXIMUM) ITEM LENGTH
AFTDBC   DS    H                800-BYTE DATA BLOCK COUNT
AFTYR    DS    H                YEAR
*
AFTADT   DS    F                POINTER TO ACTIVE DISK TABLE - 144
AFTPTR   DS    F                POINTER TO NEXT AFT BLOCK IN CHAIN - 148
*
AFTFSF   EQU   X'40'            BIT IN AFTPTR INDICATES IN FREE STORAGE
*
         DS    0D               END OF DSECT
AFTLB    EQU   *-AFTSECT        LENGTH OF AFT BLOCK IN BYTES
AFTLD    EQU   AFTLB/8          LENGTH OF AFT BLOCK IN DOUBLE WORDS
*
         MEND
```

Figure 35. CMS AFT macro

```
         MACRO
         DIOSCT

IOOLD    DC    1D'0'          IO-OLD-PSW (FROM INTERRUPT ROUTINE)
CSW      DC    1D'0'          CSW (FROM INTERRUPT ROUTINE)
*
*              WAIT CALLING SEQUENCE
*
         DS    0F
PWAIT    DC    CL8'WAIT'
         DC    C'DSK-'        FILLED IN TO CORRECT SYMBOLIC DISK NO.
         DC    F'0'
         DC    F'0'
*
QQDSK1   DC    F'0'           1ST TWO BYTES ALWAYS = 0
QQDSK2   EQU   QQDSK1+2       HALFWORD COPY OF 16TH TRACK DISK-ADDRESS
*
*              CCW CHAIN
*
CCWI     CCW   X'07',SEEKADR,CC,6       = SEEK
CCW2     CCW   X'31',SEEKADR+2,CC,5     = SEARCH
CCW3     CCW   X'08',*-8,0,1            = TIC BACK TO SEARCH
RWCCW    CCW   X'00',*-*,CC+SILI,*-*    = READ OR WRITE DATA
CCWNOP   CCW   X'03',0,SILI,1           = NO-OP FOR CE & DE TOGETHER
*
SEEKADR  DC    XL7'00'        SEEK/SEARCH INFO (1ST 3 BYTES ARE 0)
*
IOCOMM   DC    X'00'          SET TO READ (06) OR WRITE (05)
*
SENCCW   CCW   X'04',SENSB,SILI,6    = SENSE COMMAND (USED IF ERROR)
*
CC       EQU   X'40'          COMMAND-CHAIN
SILI     EQU   X'20'          ...
*
*              I/O INFO
*
LASTCYL  DC    F'0'           BECOMES 'LAST CYLINDER-NUMBER USED'
LASTHED  DC    F'0'           BECOMES 'LAST HEAD-NUMBER USED'
*
DEVTYP   DC    X'00'          UNIT-TYPE = 01 (2311), 08 (2314)
FLAG     DC    X'00'          FLAG BYTE
*
SENSB    DC    XL6'00'        SENSE-INFORMATION
*
*        MISCELLANEOUS STORAGE...
*
DOUBLE   DC    1D'0'          (FOR 'CVD' USE)
*
*        KEEP THE FOLLOWING THREE IN ORDER...
XRSAVE   DS    15F            REGISTERS 0-14 SAVED HERE FOR RDTK-WRTK
         DC    AL3(0)         FIRST 3 BYTES OF R15 ERROR-CODE
ERRCODE  DC    AL1(*-*)       ERROR-CODE (IN R15 AT EXIT)
*
*        KEEP THE FOLLOWING TWO IN ORDER..
FREER0   DC    F'0'           NO. DBL-WORDS OF FREE STORAGE (IF ANY)
DIOFREE  DC    F'0'           ADD. OF FREE STORAGE FOR BUFFER OR CCW'S
*
R1SAVE   DC    F'0'            ACTIVE-DISK-TABLE POINTER SAVED HERE
```

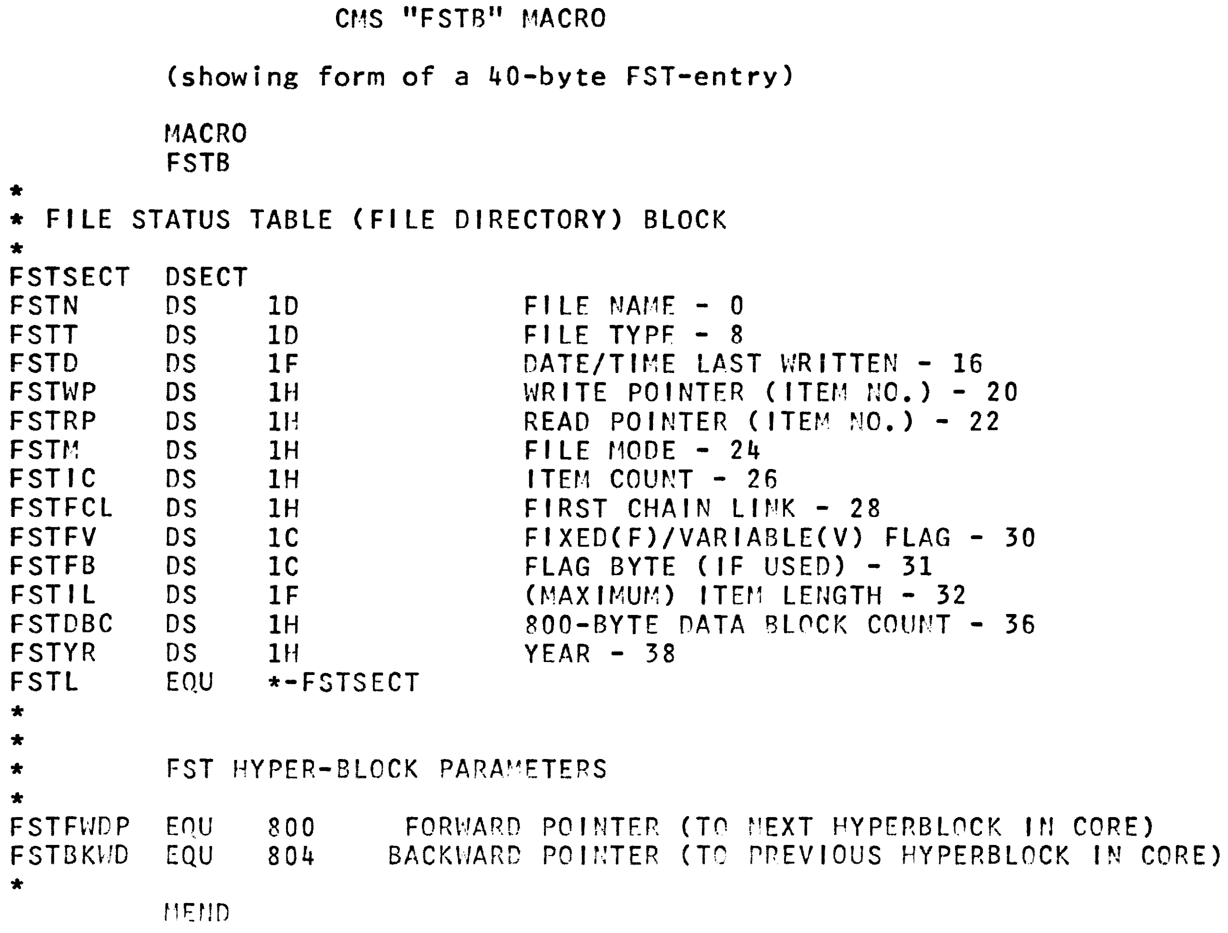
Figure 36. CMS DIOSCT macro (sheet 1 of 2)



```
*               I/O ERROR MESSAGES
*
DIOMSG1  DC     C'*** ERROR '          'SIO' OR '***'
MSG1A    DC     C'----ING '            'READ' OR 'WRIT'
MSG1B    DC     C'Z-DISK ('            'P' OR 'T' OR 'S' ETC.
MSG1C    DC     C'XXX), CYL '          DEVICE NUMBER (HEX) E.G. 191
MSG1D    DC     C'000 HEAD '           CYL. NO. (DECIMAL) 000 TO 202
MSG1E    DC     C'00 REC '             HEAD NO. (DECIMAL) 00 TO 19
MSG1F    DC     C'00  (231'            RECORD NO. (DECIMAL) 01 TO 15
MSG1G    DC     C'-)'                  '1' FOR 2311, '4' FOR 2314
LEMSG1   EQU    *-DIOMSG1
*
DIOMSG2  DC     C'CSW = '
MSG2A    DC     C'00XXXXXX XXXXXXXX'  BYTES 1-3 AND 4-7 OF CSW
MSG2B    DC     C', CCW = '            (PUT COMMA BACK IN AFTER UNPK)
MSG2C    DC     C'XXXXXXXX XXXXXXXX'  LAST CCW GOES IN HERE
MSG2D    DC     C', SENSE-INFO = '     (PUT COMMA BACK IN AFTER UNPK)
MSG2E    DC     C'000000000000'        SENSE-INFORMATION
LEMSG2   EQU    *-DIOMSG2
         DC     C' '          (EXTRA BYTE FOR UNPK SPILLOVER)

         MEND
```

Figure 36. CMS DIOSCT macro (sheet 2 of 2)



```
                    CMS "FSTB" MACRO

         (showing form of a 40-byte FST-entry)

         MACRO
         FSTB
*
* FILE STATUS TABLE (FILE DIRECTORY) BLOCK
*
FSTSECT  DSECT
FSTN     DS     1D               FILE NAME - 0
FSTT     DS     1D               FILE TYPE - 8
FSTD     DS     1F               DATE/TIME LAST WRITTEN - 16
FSTWP    DS     1H               WRITE POINTER (ITEM NO.) - 20
FSTRP    DS     1H               READ POINTER (ITEM NO.) - 22
FSTM     DS     1H               FILE MODE - 24
FSTIC    DS     1H               ITEM COUNT - 26
FSTFCL   DS     1H               FIRST CHAIN LINK - 28
FSTFV    DS     1C               FIXED(F)/VARIABLE(V) FLAG - 30
FSTFB    DS     1C               FLAG BYTE (IF USED) - 31
FSTIL    DS     1F               (MAXIMUM) ITEM LENGTH - 32
FSTDBC   DS     1H               800-BYTE DATA BLOCK COUNT - 36
FSTYR    DS     1H               YEAR - 38
FSTL     EQU    *-FSTSECT
*
*
*        FST HYPER-BLOCK PARAMETERS
*
FSTFWDP  EQU    800     FORWARD POINTER (TO NEXT HYPERBLOCK IN CORE)
FSTBKWD  EQU    804    BACKWARD POINTER (TO PREVIOUS HYPERBLOCK IN CORE)
*
         MEND
```

Figure 37. CMS FSTB macro

```
                       CMS "FVS" MACRO

         MACRO
         FVS
FVSECT   DSECT
DISK$SEG DS    15F    (1) FOR FSTLKP, FSTLKW, ACTLKP, TRKLKP, QQTRK
REGSAV3  DS    15F    (2) REGISTERS SAVED BY RDBUF, WRBUF, FINIS, STATE
RWFSTRG  DS    18F    (3) REMAINING STORAGE FOR RDBUF, WRBUF, FINIS
*
ADTFVS   DC    2F'0'          ADTLKP
*
*        SAVE-AREA FOR LOWEST-LEVEL ROUTINES:
*        E.G. READMFD, RELUFD, UPDISK, ETC.
REGSAV0  DS    15F     -- (1) SAVED R0-R15
         DC    AL3(00) -- (2) FIRST 3 BYTES OF RETURN-CODE
ERRCOD0  DC    AL1(*-*) -- (3) ERROR-CODE GOES HERE
*
TRKLSAVE EQU   REGSAV0        FOR TRKLKP/X ONLY WHEN CALLED BY QQTRK/X
*
*        SAVE-AREA FOR NEXT-TO-LOWEST LEVEL ROUTINES:
*        E.G. READFST, LOGDISK, UPUFD, ERASE, ETC.
REGSAV1  DS    15F     -- (1)
         DC    AL3(00) -- (2)
ERRCOD1  DC    AL1(*-*) -- (3)
*
AACTLKP  DC    V(ACTLKP)
AACTNXT  DC    V(ACTNXT)
AACTFREE DC    V(ACTFREE)
AACTFRET DC    V(ACTFRET)
AADTLKP  DC    V(ADTLKP)
AADTNXT  DC    V(ADTNXT)
AFSTLKP  DC    V(FSTLKP)
AFSTLKW  DC    V(FSTLKW)
ARDTK    DC    V(RDTK)
AWRTK    DC    V(WRTK)
ATRKLKP  DC    V(TRKLKP)
ATRKLKPX DC    V(TRKLKPX)
AQQTRK   DC    V(QQTRK)
AQQTRKX  DC    V(QQTRKX)
AREADFST DC    V(READFST)
AREADMFD DC    V(READMFD)
ARELUFD  DC    V(RELUFD)
AUPDISK  DC    V(UPDISK)
AKILLEX  DC    V(KILLEX)
ATFINIS  DC    V(TFINIS)
ARDBUF   DC    V(RDBUF)
AWRBUF   DC    V(WRBUF)
AFINIS   DC    V(FINIS)
ASTATE   DC    V(STATE)
ASTATEW  DC    V(STATEW)
APOINT   DC    V(POINT)
F65535   DC    F'65535'       = X'0000FFFF'
*
F4       DC    F'4'
H4       EQU   F4+2
```

Figure 38. CMS FVS macro (sheet 1 of 2)

CMS "FVS" MACRO (continued)

```
AFREE    DC    V(FREE) -- (1)
F100     DC    F'100' -- (2)
*
AFRET    DC    V(FRET)         (INTO R15)
JSR0     DC    F'0'            R0 AND ...
JSR1     DC    F'0'            R1 SAVED HERE FOR FRET CALLS.
*
*        PARAMETER-LIST TO READ/WRITE MFD...
RWMFD    DC    A(*-*) -- CORE-ADDRESS
F800     DC    F'800' -- 800 BYTES
         DC    A(H4)
FVSDSKA  DC    A(*-*) -- ADD. OF ACTIVE-DISK-TABLE
*
DSKLST   DS    0F              ALL-PURPOSE RDTK/WRTK P-LIST...
DSKLOC   DC    A(*-*)          CORE LOC. OF ITEM
RWCNT    DC    A(*-*)          BYTE-COUNT (USUALLY 800)
DSKADR   DC    A(*-*)          DISK ADDRESS OF ITEM
ADTADD   DC    A(*-*)          ADDRESS OF CORRECT ACTIVE-DISK-TABLE
*
FINISLST DC    CL8'FINIS'      P-LIST TO CLOSE ALL FILES
         DC    CL8'*'
         DC    CL8'*'
         DC    CL2'*'
*
         DS    0H              HALFWORD CONSTANTS ...
FFF      DC    X'FFFF'         MEANS NO SIGNIFICANT DATA PAST 215TH BYTE
FFE      DC    X'FFFE'        1968-ERA MFD STILL SUPPORTED ON INPUT ONLY
FFD      DC    X'FFFD'         NEWEST SIGNAL FOR FULL 2314 HANDLING
*
*        'SIGNAL' = SCRATCH HALFWORD USED BY READMFD OR ERASE...
SIGNAL   DC    H'0'            = 0000, X'FFFF', X'FFFE', OR X'FFFD'
SWTCH    EQU   SIGNAL+1        00, FF, FE, OR FD
*
UFDBUSY  DC    X'00'           NONZERO MEANS 'UFD IS BUSY BEING UPDATED'
KXFLAG   DC    X'00'           NONZERO MEANS 'KX' DESIRED ASAP.
EXTFLAG  DC    X'00'           NONZERO MEANS EXTERNAL INTERRUPT WANTED.
FLGSAVE  DC    X'00'           FOR USE AS NEEDED (AS BY 'FINIS')
*
*        FLAG BITS FOR 'UFDBUSY' FLAG...
WRBIT    EQU   X'80'           WRBUF
UPBIT    EQU   X'40'           UPDISK - READMFD
FNBIT    EQU   X'20'           FINIS
ERBIT    EQU   X'10'           ERASE - ALTER - READFST
DIOBIT   EQU   X'08'           RDTK/WRTK
*
FVSFLAG  DC    X'00'           (FOR GENERAL USE - AS NEEDED)
*
*        MISCELLANEOUS STORAGE USED BY ERASE (OR ALTER) ....
ERSFLAG  DC    X'00'           FLAG FOR USE BY ERASE OR ALTER
*
FVSERAS0 DC    F'0'            (1) - R0 TO/FROM FSTLKW (FOR ERASE)
FVSERAS1 DC    F'0'            (2) - R1 TO ACTLKP OR FSTLKW (FOR ERASE)
FVSERAS2 DC    F'0'            (3) ADDRESS OF FREE STORAGE USED BY ERASE

         DS    0D
STATEFST DC    10F'0'          40-BYTE COPY OF FST-ENTRY
STATER0  DC    F'0'            R0 AND R1 FROM FSTLKP ON MOST RECENT
STATER1  DC    F'0'            SUCCESSFUL STATE ARE SAVED HERE.
*
         MEND
```

Figure 38. CMS FVS macro (sheet 2 of 2)

## FILE MANAGEMENT FUNCTION PROGRAMS

The file management function programs are used to create and read CMS files, to locate specified files, and to enable specified items in a file to be directly accessed. The file management function programs, which are generally called via SVC X'CA', include RDBUF, WRBUF, FINIS, STATE, STATEW, and POINT.

### <u>RDBUF</u>

FUNCTION: To read one or more successive items from a specified disk file.

CALLING SEQUENCE:

```
      LA     R1, PLIST           R1 must point to P-List as usual
             then either
      SVC    X'CA'               Call RDBUF via SVC
      DC     AL4 (RDERROR)       Error-return (for example, if end-of-file)
             or
      L      R15, ARDBUF         Where ARDBUF = V(RDBUF)
      BALR   R14, R15            Call RDBUF via BALR (within nucleus)
      BNZ    RDERROR             Transfer if error(for example, end-of-file)
```

ENTRY REQUIREMENTS:

```
      R1 must point to RDBUF parameter list:
      DS     0F
PLIST DC     CL8'RDBUF'          (note — immaterial if called by BALR)
      DC     CL8'       '        filename
      DC     CL8'       '        filetype
      DC     CL2'       '        filemode
      DC     H'      '           item number of first (or only) item to be read
      DC     A(      )           address of buffer into which item(s) read are
                                 to be placed (that is, address of input buffer)
      DC     F'      '           size of input buffer.
      DC     CL2'    '           F/V Flag (in leftmost byte)
      DC     H'      '           number of items to be read
      DC     A(*-*)              number of bytes read returned here
```

EXIT CONDITIONS:

```
      Normal Return
             R15 = 0             (and condition-code = 0)
      Error Returns
             R15 nonzero         (and condition-code = 2)
```

CALLS TO OTHER ROUTINES:

ACTFREE, ACTLKP, FREE, FRET, FSTLKP, RDTK

CALLED BY (where known):

LOADMOD (in particular — called by BALR), and by all programs (usually by SVC) which read CMS files.

MACROS USED:

AFT, FSTB, FVS

ERROR RETURNS (R15 value at Exit):

1. Given file not found.

2. User Memory Area not within memory limits.

3. Permanent disk error from RDTK.

5. Number of items = 0.

7. Fixed/variable flag in FST entry = "R" (Should be "F" or "V").

8. Given memory area was smaller than actual size of item read (Note: nonfatal; number of bytes corresponding to size of buffer have been read).

9. File open for writing — must be closed before it can be read.

11. Number of items greater than 1, for variable-length file.

12. End of File (Item number specified exceeds number of items in file)

13. Variable file has invalid displacement in active file table (indicates coding error — should not occur).

Note: All errors except error 8 cause the function call to be aborted. Error 8 is legitimate if reading the first portion of a large record into a little buffer.

OPERATION: After performing some error checks, RDBUF calls ACTLKP to determine if the given file is in the active file table. If it is found but is an active write, an error 9 is given. If an active read, then processing proceeds as described under "File Active". If the file is active but neither a read nor a write, then it must have been placed in the active table by a POINT function call; processing continues as described below at the point after the entry is placed in the active file table by ACTFREE.

FILE NOT ACTIVE: If the file is not found by ACTLKP in the active file table, RDBUF checks to see if the file referenced at STATEFST (left by the most recent call to STATE) matches the caller's parameter list. (As many commands STATE a file to find its existence and characteristics and then immediately RDBUF the first record, there is a good chance this will occur — thus saving a needless search of the FST tables). If found at STATEFST, the addresses of the active disk table and the FST entry itself are obtained from the eight bytes immediately following the STATEFST copy, and FSTLKP is not called. If the file is not found in STATEFST, then FSTLKP is called to find the given file. (If not found by FSTLKP, an error 1 occurs). If found by FSTLKP, or found in STATEFST as above, then ACTFREE is called to find or create an entry in the Active File Table and insert the 40-byte FST entry therein.

When the file has been placed in the Active File Table (or was already there from a POINT function as mentioned above), RDBUF marks the file as being active. Next, RDBUF obtains buffer space into which to read the data blocks and into which to read the first chain link. It then calls the RDTK function program to read the first chain link into main storage. RDBUF next moves the first 80 bytes of the first chain link into the chain link directory in the active file table entry. Then RDBUF determines if the item(s) to be read is/are of fixed or variable length. If of variable length, processing proceeds as described under "Variable-Length Item" in this section. If of fixed length, processing proceeds as described below.

Fixed-Length Item: RDBUF calculates the number of bytes to be read. This is equal to the item length multiplied by the number of items to be read. It then calculates (from the item number supplied in the parameter list) the data block from which the item(s) is/are to be read. This calculation also yields the displacement from the start of the data block of the first byte to be read. Next, RDBUF determines whether the affected data block is in main storage. If it is not, RDBUF determines whether the chain link required to access the needed data block is in main storage. If the required chain link is not in main storage, RDBUF calls the RDTK function program to read it into main storage. After the required chain link has been read into main storage, or if it is already in main storage, RDBUF determines whether the affected data block exists. (It will if its corresponding entry in the chain link that is in main storage contains a valid disk address.) If the affected data block does not exist, RDBUF fills the input buffer with zeroes and returns to the calling program. If it does exist, RDBUF reads it into the data block buffer.

If the affected data block is in main storage when RDBUF is called, or if it is not, after it has been read into main storage (if necessary), RDBUF determines whether it contains all of the bytes to be read. (It will if the result of 800 minus the previously calculated displacement is greater than or equal to the number of bytes to be read.) If the data block contains all of the bytes to be read, RDBUF moves them from the data block buffer (where the data block resides) to the input buffer and returns to the calling program. If the data block does not contain all of the bytes to be read, RDBUF moves the pertinent bytes from the data block buffer to the input buffer. It then reads the next data block into main storage, obtains the remaining bytes to be read from it, moves them to the input buffer, and returns to the calling program. (If the 800 bytes in the next data block are not sufficient to satisfy the read, RDBUF moves the entire 800 bytes to the input buffer and reads the next data block to get the remaining bytes. RDBUF repeats this procedure until the number of bytes in the input buffer equals the number of bytes to be read. It then returns to the calling program.)

Variable-Length Records: RDBUF reads successive data blocks (starting with the first) until it locates the one that contains the start of the variable-length item to be read. It then moves the item length to the start of the input buffer. If the first data block contains the entire item, RDBUF returns to the calling program. If the first data block does not contain the entire item, RDBUF reads the next data block into the data block buffer, moves the remainder of the item to the input buffer, and returns to the calling program. If the remainder of the variable-length item is not completely contained with the 800 bytes of the second data block, RDBUF reads the next data block to get the remaining bytes. RDBUF repeats this procedure until the entire variable-length item has been placed in the input buffer. It then returns to the calling program.

FILE ACTIVE: If the file is active, RDBUF determines whether the item to be read is of fixed or variable length. If of fixed length, it proceeds as described for fixed-length items under "File Not Active". If of variable length and the item to be read immediately follows the one just read, RDBUF moves the variable-length item into the input buffer in the previously described manner. If the variable-length item to be read precedes the one just read, RDBUF proceeds as described for variable-length records under "File Not Active". If the variable-length item to be read follows, but not immediately, the one just read, RDBUF reads forward from the current location in the file until it locates the data block containing the start of the desired item. It then moves that item to the input buffer as previously described.

Notes:

1. If feasible, RDBUF reads any physical blocks of 800 bytes or more directly into the caller's buffer, rather than into a free storage buffer and then moving the data. For example, if a caller (say PRINTF) calls for forty 80-byte records, totaling 3200 bytes, RDBUF (when it has the data-block disk addresses available from the appropriate chain link) calls RDTK to read the 3200 bytes directly into the caller's buffer. This procedure saves considerable processing, SIO's to the disk, data moving, etc.

2. RDBUF, in addition to various other error checking, checks the core-address given by the caller. This core address must be no lower than the beginning of free storage (FREAR), with the single exception of the storage area BLK1, which is legal for certain applications. If the core-address is not above FREAR or within BLK1, an error code 2 is given, and no reading occurs. This safeguards the CMS nucleus from being clobbered by an invalid RDBUF parameter list in any program.

## WRBUF

FUNCTION: To write one or more successive items into a specified disk file.

CALLING SEQUENCE:

```
          LA      R1, PLIST          R1 must point to P-List as usual
                  then either
          SVC     X'CA'              Call WRBUF via SVC
          DC      AL4(ERROR)         Error-return (for example, if read-only disk)
                  or
          L       R15, AWRBUF        Where AWRBUF = V(WRBUF)
          BALR    R14,R15            Call WRBUF via BALR (within nucleus)
          BNZ     ERROR              Transfer if error (for example, read-only disk)
```

ENTRY REQUIREMENTS:

```
          R1 must point to WRBUF parameter list:
          DS      0F
PLIST     DC      CL8'WRBUF'         (Note - immaterial if called by BALR)
          DC      CL8'       '       filename
          DC      CL8'       '       filetype
          DC      CL2'   '           filemode
          DC      H'     '           item number of first (or only) item to be written
          DC      A(      )          address of buffer containing item(s) to be written
                                     (that is, address of output buffer)
          DC      F'     '           size of output buffer (number of bytes to be
                                     written)
          DC      CL2'   '           F/V Flag (in leftmost byte)
          DC      H'     '           Number of items to be written
```

EXIT CONDITIONS:

Normal Return
R15 = 0 (and condition-code = 0)

Error Returns
R15 nonzero (and condition-code = 2)

CALLS TO OTHER ROUTINES:

ACTFREE, ACTFRET, ACTLKP, ADTLKP, DISKDIE, FREE, FRET, FSTLKW, KILLEXF, QQTRK, QQTRKX, RDTK, TRKLKP, TRKLKPX, WRTK

CALLED BY (where known):

GENMOD (in particular — called by BALR), and by all programs (usually by SVC) which write CMS disk files

MACROS USED:

ADT, AFT, FSTB, FVS

ERROR RETURNS TO CALLER (R15 value at Exit):

1. Filename or filetype not specified or illegal

2. User memory address = 0

4. First character mode illegal

5. Second character mode illegal

6. Item number + number of items too large - will not fit in a halfword

7. Attempt to skip over unwritten variable-length item

8. Number of bytes not specified

9. File already active for reading

10. Maximum number of CMS files (3500) reached

11. F-V flag not F or V

12. Mode SY (SYSTEM) or other read-only disk

14. Attempt to write on T-Disk which is not yet formatted

15. Length this item not same as previous

16. Characteristic (F-V Flag) not same as previous

17. Variable-length item greater than 65K bytes

18. Number of items greater than 1 for variable-length file

19. Maximum number of data blocks per file (16060) reached

OTHER ERROR RETURNS:

Transfers to DISKDIE (within FINIS) on a permanent I/O Error.
Transfers to KILLEXF (within LOGOUT) if disk is full.

OPERATION: WRBUF first performs a series of tests to ensure that the parameter list is legal. If it is not, WRBUF signals the error and returns to the calling program. If the parameter is legal, WRBUF calls the ACTLKP routine to see if the file exists and is active; if yes, processing proceeds as described under "File Active." If not, WRBUF calls the FSTLKW function program to determine whether the specified file exists. If yes, processing proceeds as described under "File Exists, Not Active." If not, processing proceeds as described under "File Does Not Exist."

FILE DOES NOT EXIST: If the file does not exist, WRBUF calls ADTLKP to determine the active disk table pertaining to the given mode, and checks to ensure that the disk is available and in read-write status (error return if not). The ACTFREE is called to obtain an available slot in the Active File Table for the file about to be created. Then WRBUF initializes the AFT entry with necessary information including the name, type, and mode of the file. WRBUF then calls the QQTRK routine to obtain an available sixteenth of a track of disk space for use as the first chain link and stores the disk address returned by QQTRK in the file status table. Next, WRBUF calculates (from the item number supplied in the parameter list) the data block into which the item(s) is/are to be written. This calculation also yields the location within the data block at which the item(s) will reside. (The calculation is ((N-1) *L)/800. N is the item number, L is the item length, and 800 is the length of a data block. The quotient produced by this calculation is the number of the affected data block and the remainder is the displacement into the data block at which the item(s) will reside.) Next, WRBUF calculates the number of bytes to be written. This is equal to the item length multiplied by the number of items to be written. Both values are obtained from the parameter list. WRBUF then marks the file active, obtains buffer space for the data block, and determines if the item to be written is of fixed or variable length. If of variable length, processing proceeds as described under "Variable-Length Item". If of fixed length, processing proceeds as described below.

Fixed-Length Item: WRBUF determines the chain link that should contain the address of the affected data block. (Ordinarily, at this point, this will be the first chain link and it will exist in main storage.) If this chain link does not exist (that is, its corresponding entry in the first chain link is not a valid disk address), WRBUF calls the TRKLKP function program to obtain a quarter of a track for the new chain link, inserts the disk address returned by TRKLKP into the chain link directory of the active file table entry, and obtains storage for use in constructing the new chain link. If the chain link exists, WRBUF calls the RDTK function program to read it into main storage. WRBUF then determines if the affected data block exists. (It will if the corresponding entry in the chain link that is in main storage contains a valid disk address.) If it does not exist, WRBUF calls the TRKLKP function program to obtain a quarter of a track for the new data block, inserts the disk address returned by TRKLKP into the appropriate entry in the chain link that is in main storage, and clears the data block buffer for use in constructing the data block. If the data block exists, WRBUF calls the RDTK function program to read it into the data block buffer. WRBUF then calculates the number of bytes in the data block buffer that are available for use. (The number of bytes available is equal to 800 minus the previously calculated displacement.) Next, WRBUF determines whether the number of bytes to be written is greater than the number of bytes available in the data block buffer. If the number of bytes to be written is not greater than the number available, WRBUF moves the bytes to be written from the input buffer to the data block buffer and returns to the calling program. (In this case, the data block is not written onto disk because it is not full.) If the number of bytes to be written exceeds the number of bytes available, WRBUF moves sufficient bytes into the data block buffer to fill it, and writes the completed data block onto disk. WRBUF then determines if the chain link that should contain the address of the data block that is to receive the overflow from the previous data block is in main storage. If it is not, WRBUF writes the current chain link (that is, the one in main storage) onto disk and retrieves the chain link containing the address of the data block that is to receive the overflow. This chain link may or may not exist. If the chain link does not exist, WRBUF allocates disk space for the new chain link in the previously described manner and determines if the data block that is to receive the overflow exists as previously described. If the chain link exists, WRBUF reads it into main storage and determines if the data block that is to receive the overflow exists. When the data block that is to receive the overflow is in main storage (that is, in the data block buffer), WRBUF calculates the number of bytes remaining to be written. If this is not greater than the number of bytes available in the data block buffer (on overflow, all 800 bytes of the data block buffer are available), WRBUF moves the remaining bytes from the input buffer to the data block buffer and returns to the caller. If the number of bytes remaining to be written is greater than the number of bytes available in the data block buffer, WRBUF moves sufficient bytes into the data block buffer to fill it, writes the data block onto disk, and moves the overflow into the next data block as described.

Variable-Length Item: WRBUF reads successive data blocks (starting with the first) into the data block buffer until it locates the one that contains the item immediately preceding the one that corresponds to the item number specified in the parameter list. It then locates the end of that item. (This may entail reading additional data blocks, depending on the length of the item.) When it locates the end of the item, WRBUF moves the length of the item to be written from the input buffer to the location in the data block buffer immediately after the end of the previous item. It then moves the item to be written from the input buffer to the data block buffer in the same manner as for fixed-length items. (If overflow occurs, it is handled in the same manner as for fixed-length items.)

FILE EXISTS, NOT ACTIVE: If the file exists but is not active, WRBUF calculates the data block into which the item(s) is to be written. This calculation also yields the location within the data block at which the item(s) will reside. ACTFREE is called to obtain an available slot in the Active File Table and to store the FST entry therein. Next, WRBUF marks the file as active, reads the first chain link into main storage, and moves the first 80 bytes of the first chain link into the chain link directory of the active file table entry. WRBUF then determines if the item(s) to be written are of fixed or variable length. For both of these item types, WRBUF proceeds as described under the corresponding heading in "FILE DOES NOT EXIST" in this section.

FILE ACTIVE: If the file is active, WRBUF calculates the data block into which the item(s) is/are to be written. This calculation also yields the displacement into the data block at which the item(s) will reside. Next, WRBUF determines the nature of the item(s) to be written. If of variable length, WRBUF proceeds as described under "Variable-Length Item". If of fixed length, it proceeds as described below.

Fixed-Length Item: WRBUF determines whether the affected data block is in main storage. If it is, WRBUF proceeds as described under "File Does Not Exist", starting at the point where the number of bytes available in the data block buffer is calculated. If the affected data block is not in main storage, WRBUF proceeds in essentially the the same manner as described under "File Does Not Exist", starting at the point where the data block is written onto disk. (In this case, an overflow condition is not being processed; however, the logic used to obtain the affected chain link and data block is essentially the same. Also, because this is not an overflow condition, when the affected data block is resident in the data block buffer, the number of bytes available in that buffer is equal to 800 minus the calculated displacement.)

Variable-Length Item: If the variable-length item to be written immediately follows the one that was just processed, WRBUF moves the item length from the input buffer into the data block buffer immediately after the end of the previous item. It then moves the item to be written from the input buffer into the data block buffer immediately after the length. This is done in the usual manner. (If overflow occurs, it is handled in the usual manner.) If the item to be written does not immediately follow the one that was just processed, WRBUF proceeds in the same manner as described under the variable-length item portion of "FILE DOES NOT EXIST".

Notes:

1. WRBUF can only write a certain number of logical records or items, regardless of how much disk space may be available, because the "number of items" is kept in a halfword in the 40-byte FST entry for that file, and is limited by the size of a number which will fit in a (16-bit) halfword. To avoid running into this limitation before it is too late to close the file successfully, WRBUF checks that the item-number (when a WRBUF call has been completed) will not exceed a given limit. If it does, an error code 6 is returned, and no more data is written. The file may, however, at this point be successfully closed (via FINIS), and can later be read by RDBUF. At present this limiting number of records happens to be 65533. (65535 would have been the absolute limiting factor.)

2. In calls to QQTRK for obtaining the first chain link for a new file, and to TRKLKP for obtaining either a new Nth chain link or a data block, error codes are checked from these function programs for the full disk condition. If any of these situations occur, WRBUF carefully sets or resets any flags or conditions as needed, and calls upon the KILLEXF code to close all files, compact the directory, update the user file directory, and re-IPL. The file which was being WRBUF'ed (unless null) is then available and complete insofar as the data being written could fit in the space available.

3. Because of the design of the first chain link in the CMS file system, there is a limitation of 16060 800-byte data blocks for any given file. If a file being WRBUF'ed reaches this limit, an error 19 is returned, and no more data is written. The file may be closed, and can then be successfully read (or erased), but it cannot be made any larger. (A file of this size would fill more than half of a full-size 2314 disk).

4. There is also a limit of 3500 files that can be represented for any given disk, as limited by the layout of the MFD block. If a disk already has reached this maximum and an attempt to WRBUF a new file is made, WRBUF returns an error code 10, and the new file is not opened.

## FINIS

FUNCTION: To close one or more input or output disk file(s).

CALLING SEQUENCE:

```
        LA      R1, PLIST           R1 must point to P-List as usual
                then either
        SYC     X'CA'               Call FINIS via SVC
        DC      AL4(ERROR)          Error-return (for example, if file not open)
                or
        L       R15, AFINIS         Where AFINIS = V(FINIS)
        BALR    R14, R15            Call FINIS via BALR (within nucleus)
        BNZ     ERROR               Transfer if error (for example, file not open)
```

ENTRY REQUIREMENTS:

```
        R1 must point to FINIS parameter list:
        DS      0F
PLIST   DC      CL8'FINIS'          (Note - immaterial if called by BALR)
        DC      CL8'     '          filename
        DC      CL8'     '          filetype
        DC      CL2' '   '          filemode
```

EXIT CONDITIONS:

```
        Normal Return
                R15 = 0             (and condition-code = 0)
        File Not Open
                R15 = 6             (and condition-code = 2)
```

CALLS TO OTHER ROUTINES:

ACTFRET, ACTLKP, DISKDIE, ERASE, FREE, FRET, FSTLKW, RDTK, UPDISK, WRTK

CALLED BY (where known):

GENMOD & LOADMOD (called by BALR), UPUFD, LOGDISK, and by all commands which use RDBUF & WRBUF.

MACROS USED:

ADT, AFT, FSTB, FVS

OPERATION: FINIS checks the caller's parameter list for '*' in the filename or filetype, or a nonalphabetic character for the mode; if any of these conditions are met, a flag is set to check for additional entries in the Active File Table.

After this preliminary check, FINIS calls ACTLKP to find an AFT block that matches the caller's parameter list. If none is found, an error 6 is given as shown in the exit conditions.

If a match is found, a check is made to determine whether the file is an active write, an active read, or neither. If neither, it was placed there by POINT, but was not read or written subsequently. Action is taken in these three cases as described in the following paragraphs.

Active Read File

If the file found by ACTLKP is an active read file, FINIS takes the following steps to close the file:

1. Release to free storage the 800-byte buffer used for the data block (via a call to FRET).

2. Also release either the 200- or 800-byte buffer currently in use for the chain link.

3. If the file has a mode number of 3 or 4 (for example, P3, T4, etc.), it is now erased. This is done by calling FREE to obtain free storage for a suitable call to ERASE, then calling ERASE to eliminate the file, and then giving back the free storage via FRET. Care is taken to preserve information to avoid re-enterability problems between FINIS and ERASE.

4. Next, ACTFRET is called to release this slot in the Active File Table.

5. Finally, if either the filename, filetype, or filemode indicated that additional files should be checked, FINIS returns to the portion of code which calls ACTLKP, to check for any more AFT blocks that may match the caller's P-List.

6. Finally, when all appropriate file(s) have been closed, FINIS gives a normal return as indicated under exit conditions.

File Active from a Point Call

For this case (active but neither a read nor a write), ACTFRET is called, etc., as shown above in steps 4, 5, and 6 for the "Active Read File" case.

Active Write File

If the file found by ACTLKP is an active write file, FINIS takes the following steps to close the file:

1. Checks the pointer (AFTPFST) in the AFT block to the FST entry (if any) in the FST hyperblocks. In nonzero, proceed to step 2. If zero (as is the case for a new file never before closed), the special FSTLKW entry to obtain an empty 40-byte FST entry is called, and the AFTPFST pointer is set to the address provided by FSTLKW.

2. Moves the 40-byte entry from the AFTFST slot in the AFT block to its location within the FST hyperblocks, sets the mode-letter therein to P, and clears the flag-byte.

3. Unless the first five letters of the filetype = SYSUT, the time of day and year are computed in the same manner as GETCLK, and the date-time stored in FSTD in the FST entry, and the year in FSTYR. (If the filetype indicates a utility file is being FINIS'ed, this step is unnecessary and is therefore omitted.)

4. Next the current data block pointed to by AFTDBA is written on disk.

5. Then the free storage block that was used for the data block is returned to free storage via FRET.

6. If the first chain link is not in core, the current chain link (unless null) is written on disk, and the first chain link brought into core.

7. The linkage portion of the first chain link (AFTCLB) is moved from the AFT block to the first chain link, and the first chain link written on disk.

8. Then the free storage block used for the chain link (either 200 or 800 bytes in length) is returned to free storage via FRET.

9. The write pointer is computed as the number of items plus one and stored in the FST entry.

10. The number of active write files for this active disk table (ADTNACW) is decremented by one.

11. If the number of active write files (ADTNACW) is now = 0, then UPDISK is called to update the file directory for this active disk table.

12. ACTFRET is then called to release this slot in the Active File Table.

13. Then if either the filename, filetype, or filemode indicated that additional files should be checked, FINIS returns to the portion of code that calls ACTLKP, to check for any more AFT blocks that may match the caller's P-List.

14. Finally, when all appropriate file(s) have been closed, FINIS gives a normal return as indicated under exit conditions.

Notes:

1. A special entry to the FINIS program called "TFINIS" (called only by BALR) is provided for some special logic necessary for use by ERASE and ALTER. This logic uses some, but not all, of the above steps in closing an input or output file. See the description of "TFINIS" under File Management Routines for details.

2. If a permanent disk error occurs in closing an output file, FINIS types a warning message on the user's terminal, and loads a PSW with a disabled wait state, rather

than trying to continue. This procedure is purposely followed to preserve the user's old file directory, as part of CMS's double directory scheme. This code is also enterable from without the FINIS program, as the DISKDIE entry point, and is also invoked from WRBUF, ERASE, and UPDISK in the event of a permanent I/O error, again to preserve the old directory. The warning message (self-explanatory) is as follows:

DISK HARDWARE ERROR; NOTIFY OPERATOR; RE-IPL WHEN CORRECTED

## STATE

FUNCTION: To locate the file status table entry for a given file, and if found to provide the caller with a copy thereof.

CALLING SEQUENCE:

```
      LA      R1, PLIST           R1 must point to P-List as usual
              then either
      SVC     X'CA'               Call STATE via SVC
      DC      AL4(NOT FOUND)      Error-return (if not found)
              or
      L       R15, ASTATE         Where ASTATE = V(STATE)
      BALR    R14,R15             Call STATE via BALR (within nucleus)
      BNZ     NOTFOUND            Transfer if error (not found)
```

ENTRY REQUIREMENTS:

```
      R1 must point to STATE parameter list:
      DS      0F
PLIST DC      CL8'STATE'          (Note - immaterial if called by BALR)
      DC      CL8'        '       filename
      DC      CL8'        '       filetype
      DC      CL2'  '             filemode
      DC      CL2   '             not used
ADCONDC       A(*-*)              Address of copy of FST entry returned here if
                                  file was found
```

EXIT CONDITIONS:

File Found

```
              R15 = 0             (and condition-code = 0)
              also, ADCON in P-List filled in to V(STATEFST)
```

File Not Found

```
              R15 = 1             (and condition-code = 2)
```

CALLS TO OTHER ROUTINES:

ACTLKP, FSTLKP

CALLED BY (where known):

LOADMOD (in particular - called by BALR), and by all programs (usually by SVC) that check a file's existence and characteristics before reading it.

MACROS USED:

ADT, AFT, FSTB, FVS

OPERATION: STATE calls ACTLKP to see if the given file is in the Active File Table. If found, the active 40-byte FST entry is moved from AFTFST in the AFT block to the copy at STATEFST to be provided to the caller, and the STATER0 and STATER1 words following STATEFST are set to the addresses of the ADT block and the FST entry respectively; then V(STATEFST) is stored in the caller's P-List (if necessary) as described below.

If the file was not found by ACTLKP, STATE then calls FSTLKP to find the given file. If not found by FSTLKP, the error-code from FSTLKP (= 1 for file not found) is returned to the caller as shown in the exit conditions.

If the file was found by FSTLKP, then the 40-byte entry is moved to STATEFST, and the R0 and R1 values obtained from FSTLKP are stored at the STATER0 and STATER1 words following STATEFST. (These words are used by RDBUF to avoid an extra search of the FST tables under the circumstances given in the RDBUF description). Where the file was found by FSTLKP, the mode letter is stored in the STATEFST copy using the same algorithm as ACTFREE, being carefully chosen from that of the caller's parameter list, or the ADTM or ADTMX mode given by the Active Disk Table.

The result of the choice of mode-letter facilitates the feature of a read-only extension of a given disk. For example, if an A-Disk is a read-only extension of a P-Disk, if the caller's parameter list specified the A-mode, the mode stored in STATEFST will be A; but if the caller specified P or '*', the mode stored in STATEFST will be P.

After setting up STATEFST and the two words that follow, as described above, STATE stores the address of STATEFST in the caller's P-List (unless it is already there), and returns to the caller as shown in the exit conditions.

## STATEW

FUNCTION: To locate the file status table entry for a given file on a read-write disk, and if found to provide the caller with a copy thereof.

CALLING SEQUENCE:

```
        LA      R1, PLIST             R1 must point to P-List as usual
                then either
        SVC     X'CA'                 Call STATEW via SVC
        DC      AL4(NOTFOUND)         Error-return (if not found)
                or
        L       R15, ASTATEW          Where ASTATEW = V(STATEW)
        BALR    R14, R15              Call STATEW via BALR (within nucleus)
        BNZ     NOTFOUND              Transfer if error (not found)
```

ENTRY REQUIREMENTS:

```
        R1 must point to STATEW parameter list:
        DS      0F
PLIST   DC      CL8'STATEW'           (Note - immaterial if called by BALR)
        DC      CL8'        '         filename
        DC      CL8'        '         filetype
        DC      CL2'  '               filemode
        DC      CL2'  '               not used
ADCONDC         A(*-*)                Address of copy of FST entry returned here
                                      if file was found
```

EXIT CONDITIONS:

```
        File Found
                R15 = 0               (and condition-code = 0)
                also, ADCON in P-List filled in to V(STATEFST)
        File Not Found
                R15 = 1               (and condition-code = 2
```

CALLS TO OTHER ROUTINES:

ACTLKP, FSTLKW

CALLED BY (where known):

OFFLINE

MACROS USED:

ADT, AFT, FSTB, FVS

OPERATION: STATEW is identical to STATE in operation, except that FSTLKW is called to find the given file on a read-write disk, instead of FSTLKP as called by STATE.

STATEW is included in the STATE function program.

## POINT

FUNCTION: To place a file status table entry in the Active File Table (if necessary), and to set the read pointer and/or write pointer for that file to a given item number.

CALLING SEQUENCE:

```
        LA      R1, PLIST         R1 must point to P-List as usual
                then either
        SVC     X'CA'             Call POINT via SVC
        DC      AL4(ERROR)        Error-return
                or
        L       R15, APOINT       Where APOINT = V(POINT)
        BALR    R14, R15          Call POINT via BALR (within nucleus)
        BNZ     ERROR             Transfer if error
```

ENTRY REQUIREMENTS:

```
        R1 must point to POINT parameter list:
        DS      0F
PLIST   DC      CL8'POINT'        (Note - immaterial if called by BALR)
        DC      CL8'      '       filename
        DC      CL8'      '       filetype
        DC      CL2'  '           filemode
        DC      H'    '           write pointer
        DC      H'    '           read pointer
```

EXIT CONDITIONS:

Normal Return:
R15 = 0 (and condition-code = 0)

File Not Found:
R15 = 1 (and condition-code = 2)

Parameter List Error:
R15 = 2 (and condition-code = 2)

CALLS TO OTHER ROUTINES:

ACTFREE, ACTLKP, FSTLKP

CALLED BY (where known):

Disk resident routines

MACROS USED:

AFT, FVS

OPERATION: POINT checks for possible parameter list errors (for example if called from a terminal), and exits with error 2 if parameter list is faulty.

If not, POINT calls ACTLKP to determine if the FST entry for the given file is already in the Active File Table. If yes, the read and/or write pointers are set as described below.

If not found by ACTLKP, then POINT calls FSTLKP to find the file. If it is not found, error 1 is returned to the caller. If found, then ACTFREE is called to place the given file in the active file table.

POINT then checks the read pointer provided by the caller; if it is zero, no action is taken. But if nonzero, then its value is stored in the read pointer (AFTRP) in the active file table.

Next, POINT checks the write pointer provided by the caller; if it is zero, no action is taken. If the write pointer is a halfword of all ones (that is, = 65535), then the write pointer AFTWP is set to the number of items (AFTIC) plus one. If the write pointer is neither 0 nor 65535, then its value is stored in the write pointer (AFTWP) in the active file table.

When through, POINT returns to the caller as shown in the exit conditions above.

## FILE MANAGEMENT ROUTINES

The file management routines are used to locate specified files, to read file management tables from disk into main storage, to write file management tables from main storage onto disk, to delete old copies of file management tables, and to enable specified items in a file to be directly accessed. The file management routines, which are called via BALR R14,R15 from the CMS initialization process, various commands, and the file management function programs, include TFINIS, RELUFD, READFST, READMFD, FSTLKP, FSTLKW, UPDISK, UPUFD, and SYSGEN. The LOGDISK command is also included in this section.

### TFINIS

FUNCTION: To temporarily close a given file or active disk table, for the purpose of updating the file directory.

CALLING SEQUENCE:

```
          L         R15,ATFINIS          where ATFINIS = V(TFINIS)
          BALR      R14,R15
```

ENTRY REQUIREMENTS:

1. EFINIS Entry - to close a particular file without updating the directory or removing from Active File Table

   R0 = Pointer to Active Disk Table
   R1 = Pointer to Active File Table

2. TFINIS Entry - to temporarily close all output files for a given Active Disk Table

   R0 = Pointer to Active Disk Table
   R1 = 0

EXIT CONDITIONS:

Normal Return
R15 = 0 (and condition-code = 0)

File Not Open
R15 = 6 (and condition-code = 2)

CALLS TO OTHER ROUTINES:

ACTLKP, DISKDIE, FRET, FSTLKW, RDTK, WRTK

CALLED BY (where known):

EFINIS Entry called by ERASE
TFINIS Entry called by ALTER and ERASE

MACROS USED:

ADT, AFT, FSTB, FVS

OPERATION: The TFINIS Routine is part of the FINIS function program. It is called, however, only by BALR, as from ALTER or ERASE (not via SVC).

The EFINIS entry is differentiated from the TFINIS entry from R1 being zero (for TFINIS), or nonzero (for EFINIS).

See the FINIS description for information on the FINIS steps, some of which are followed by EFINIS and TFINIS, as described below.

1. The EFINIS logic is as follows:

   Active Read File
      Gives back the free storage buffers as done in steps 1 and 2 of the "Active Read File" in the FINIS description. (Note that ACTFRET is not called - this is done later by ERASE.)

   Active File From Point
      No action taken. (ERASE calls ACTFRET later.)

   Active Write File
      Performs selected steps of those followed by the "Active Write File" logic as given in the FINIS description, namely steps 4 through 10 (omitting steps 1-3 and 11-14).

2. The TFINIS logic, for temporarily closing all output files for a given disk (called by ERASE and ALTER) is as follows:

   Search through Active File Table for entries (if any) whose active disk table matches that provided to TFINIS. For each one found (if any), action is as follows:

   Active Read File
      No action taken.

   Active File From Point
      No action taken.

   Active Write File
      Performs selected steps of those followed by the "Active Write File" logic as given in the FINIS description, namely steps 1, 2, 3, 4, 6, 7, 9, 13, and 14 (omitting steps 5, 8, and 10 through 12).

Note: One additional step is performed if needed; if it was necessary to bring the first chain link into core in step 6, the Nth chain link is brought back into core after step 7.

See the ERASE and ALTER commands for further insight into the reasons for the EFINIS and TFINIS logic as outlined above.

## RELUFD

FUNCTION: For a given disk, to release all tables kept in free storage and to clear appropriate information in the active disk table.

CALLING SEQUENCE:

```
L       R15,ARELUFD        where ARELUFD = V(RELUFD)
BALR    R14,R15
```

ENTRY REQUIREMENTS:

R0 must point to Active Disk Table

EXIT CONDITIONS:

R15 = 0 (and condition-code = 0)

CALLS TO OTHER ROUTINES:

FRET

CALLED BY (where known):

LOGIN, and RELEASE, plus disk resident routines

MACROS USED:

ADT, FVS

OPERATION: For the given Active Disk Table, the following tables are returned to free storage via FRET, if they are currently in core:

1. All FST hyperblock extensions (if any)
2. The first FST hyperblock if it was in free storage
3. Master File Directory
4. QMSK bit-mask
5. QQMSK table if it was in free storage

In clearing any of the above, the appropriate flag-bits are also cleared, and any pointers pointing to the old tables.

For certain tables, RELUFD clears them if they exist but are not in free storage, namely:

1. First FST hyperblock if not in free storage (for example, PSTAT)
2. QQMSK if not in free storage (for example, PQQMSK)

RELUFD also clears all information in the Active Disk Table from ADTMFDN through ADTCYL, and sets the ADTMX extension-mode-letter to a blank. Also, unless the ADT table is minimum size (ADTFMIN flag-bit set in ADTLFG1), RELUFD clears all information in the active disk table from ADTPQM1 to ADTRES, and also clears the ADTFLG2 flag-byte.

RELUFD is called by RELEASE for releasing an active disk, and by LOGIN and FORMAT to clear all information before reading in or creating a new user file directory for the given disk.

RELUFD replaces the old RELPSTA routine.

## READFST

FUNCTION: For a read-write disk, to read all of the User File Directory into core; for a read-only disk, to read in all or part of the User File Directory, at the caller's option.

CALLING SEQUENCE:

```
        L         R15,AREADFST          where AREADFST = V(READFST)
        BALR      R14,R15
```

ENTRY REQUIREMENTS:

R0 must point to Active Disk Table
R1 must point to Parameter-List as usual:

```
                  DS        0F
        PLIST     DC        CL8'        '   Immaterial
                  DC        CL8'        '   FILENAME (or '*')
                  DC        CL8'        '   FILETYPE (or '*')
                  DC        CL2'  '         MODE (e.g. P, '*', or P2)
```

EXIT CONDITIONS:

```
        Normal Return
                  R15 = 0               (and condition-code = 0)

        Error Returns                   (condition-code = 2)
                  R15 = 4               Disk is read-only (nonfatal)
                  R15 = 1,2,3, or 5:    Same error conditions as READMFD
                                        (error from READMFD passed along as is)
```

CALLS TO OTHER ROUTINES:

FREE, FRET, RDTK, READMFD

CALLED BY (where known):

SYSGEN, INIT, LOGIN

MACROS USED:

ADT, FVS

OPERATION: READFST, together with READMFD, brings into core all or part of the user file directory for the given disk. If the disk is read-write, all of the UFD is brought into core; if read-only, the QMSK and QQMSK tables are not brought in by READMFD, and READFST can bring in selected portions of the FST entries, if specified.

READFST does the following:

First READMFD is called to read in its part of the UFD. If an error other than 4 is returned by READMFD, READFST passes back this error code to the caller and exits immediately. If READMFD was successful, or returned an error-4 indicating the disk was read-only, READFST continues. An 816-byte buffer for the first FST hyperblock is obtained from free storage if needed, and an 800-byte work area is obtained.

The FST hyperblocks on disk are now read into the work area, one at a time. All null FST entries are ignored; other entries are moved from the work area to the core-resident FST hyperblocks if the disk is read-write, thus resulting in compacted directory in core of all files. If the disk is read-only, each FST entry in the work area is checked against the parameter list provided to READFST. (If any field in the parameter list is '*' or X'FF', the filename, filetype, or mode number is accepted without checking.) Thus, for a read-only disk READFST can read in all files, or all 'P2' files, (as it does for the S-Disk when called by SYSGEN) or any conditions that satisfy the parameter list.

READFST gets more hyperblocks from free storage when needed and refills the work buffer from disk when needed, until all FST entries have been checked and moved into the FST hyperblocks if acceptable.

All appropriate counts in the Active Disk Table (number of files, pointer to last FST entry, etc.) are initialized as needed.

When through, READFST returns the core resident MFD to free storage if the disk was read-only, as it is not needed any more. For a read-write disk, however, the MFD is purposely left in core for use by the UPDISK routine the next time the UFD is updated.

The work-buffer is returned to free storage, and READFST returns to the caller, returning an error-code 0 or 4 (if the disk was found to be read-only by READMFD).

## READMFD

FUNCTION: To read the Master File Directory (MFD) and other information into core from disk.

CALLING SEQUENCE:

```
        L         R15,AREADMFD           where AREADMFD = V(READMFD)
        BALR      R14,R15
```

ENTRY REQUIREMENTS:

| | |
|---|---|
| R0 | must point to Active Disk Table |

EXIT CONDITIONS

Normal Return

| | |
|---|---|
| R15 = 0 | (and condition-code = 0) |

Error Returns

| | |
|---|---|
| R15 = 1 : | Disk error reading MFD, or first word of MFD = 0 |
| R15 = 2 : | Disk not attached |
| R15 = 3 : | Unrecognizable DASD device (neither 2314, nor 2311 or equivalent) |
| R15 = 4 : | Disk is found to be read-only (Note: nonfatal) |
| R15 = 5 : | Disk is apparently a 2311 with the old device-dependent disk addresses (from 1966-67 era) (Unit-type-byte = 0) |

CALLS TO OTHER ROUTINES:

FREE, FRET, RDTK, WRTK

CALLED BY (where known):

READFST or LOGIN

MACROS USED:

ADT, FVS

OPERATION: READFST and READMFD when used together (READFST calling READMFD) are called by LOGIN or SYSGEN to bring all or part of a user file directory into core. READMFD does not bring in any FST hyperblocks (that being done by READFST), and is therefore called directly by LOGIN (without calling READFST) when logging in a disk "NO-UFD".

READMFD does the following:

A sense command is issued to the disk to make sure it is attached and ready. Error 2 is made if not attached.

The fourth sense byte is checked to make sure the disk is a 2314, or a 2311 (or equivalent). Error 3 is made if neither. The sense byte of 08 (for 2314) or 01 (for 2311) is stored in the fourth byte of the 12-byte device-table entry in the NUCON table, for the particular device given by the active disk table.

Next, the Master File Directory (record no. 4 - that is, cylinder 0, head 0, record 4) is read from disk into a buffer obtained from free storage. (Error 1 if cannot be read successfully). The first halfword of the MFD is checked to make sure data is there (error 1 if not).

The disk addresses at the beginning of the MFD are then checked, scanning for an ending sentinel of FFFF, FFFE, FFFD, to determine how many FST hyperblocks are on disk, and where the QMSK extensions (if any) are stored on disk. The number of FST hyperblocks is stored as ADTHBCT in the Active Disk Table for future use (usually by READFST). The disk counts ADTNUM etc., are stored in the Active Disk Table from the MFD, along with the number of cylinders.

At this point, if the read-only flag-bit ADTFRO in the ADTFLG1 flag is set, READMFD accepts the disk as read-only, leaves the MFD block in a free storage buffer of just enough size to include all the disk-addresses at the beginning, and exits to the caller, normally READFST.

If the disk is not flagged read-only, READMFD attempts to write the MFD exactly as is right back on disk, using WRTK. If unsuccessful because the disk is read-only (an error 6 from WRTK), READMFD finishes up as described in the above paragraph (the read-only bit being set now), and exit is subsequently made with error 4 (nonfatal).

If the MFD was successfully written back on disk, the read-write flag-bit is set, and READMFD continues. Next the right amount of free storage is obtained for the QMSK bit-mask (depending on the size of ADTNUM - the total number of records on disk). Then the QMSK extensions (if any) are read into the free storage area, double-word aligned, and with an integral number of double-words in the read (to prevent a possible chaining check when running on CP), and then moved to the proper place in the QMSK buffer; the first 215 bytes of the QMSK are then moved from the MFD to the QMSK in core. (If the QMSK is less than 215 bytes, only the correct number of bytes is moved). When through, the QMSK is laid out in core as one contiguous table, with an integral number of bytes, padded (if necessary) to an integral number of double words, in free storage.

The location of the QMSK is of course stored where needed, and the other counts (ADTPQM1 through ADTPQM3), computed and stored.

Free storage is then obtained for the QQMSK, if necessary, and the 200-byte QQMSK table moved into position from the MFD.

Lastly, as mentioned above for the read-only cases, just enough free storage is obtained for the information on disk addresses contained at the beginning of the MFD to be kept in core, the crucial data is moved thereto, and the 800-byte buffer is released.

To summarize:

For a read-write disk, READMFD reads in all of the User File Directory except the FST hyperblocks and initializes all appropriate information in the Active Disk Table.

For a read-only disk, READMFD reads in all of the User File Directory except the FST hyperblocks, the QMSK and QQMSK tables, and the counts in the latter half of the active disk table associated with the QMSK and QQMSK tables, which are not needed in core for a read-only disk.

## FSTLKP

FUNCTION: To find a specified 40-byte FST entry within the FST tables for read-only or read-write disk(s).

CALLING SEQUENCE:

```
        L         R15,AFSTLKP              where AFSTLKP = V(FSTLKP)
        BALR      R14,R15
```

ENTRY REQUIREMENTS:

1. To search appropriate disk table(s) from the beginning:

R0 = Immaterial
R1 = Pointer to usual P-List (with sign-bit plus):

```
                  DS        0F
        PLIST     DC        CL8'    '   Immaterial
                  DC        CL8'    '   FILENAME or '*'
                  DC        CL8'    '   FILETYPE or '*'
                  DC        CL2'  '     FILEMODE or '*'
```

2. To search appropriate disk table(s), picking up from where you left off previously, starting with next 40-byte FST entry:

R0 = Pointer to Active Disk Table
R1 = Pointer to usual P-List but with sign-bit negative

EXIT CONDITIONS:

File Found:

R0 = Pointer to Active Disk Table
R1 = Pointer to (address of) 40-byte FST entry found
R15 = 0 (and condition-code = 0)

File Not Found:

R0 = 0
R1 = 0 (with sign-bit negative)
R15 = 1 (and condition-code = 2)

Parameter List Error:

R0 = 0
R1 = 0 (with sign-bit negative)
R15 = 2 (and condition-code = 2)

CALLS TO OTHER ROUTINES:

ADTLKP, ADTNXT

CALLED BY (where known):

DISK, INIT, POINT, RDBUF, STATE, plus disk resident routines

MACROS USED:

ADT, FVS

OPERATION: FSTLKP checks to ensure that R1 is not zero (a calling error), and initializes to test for either a read-only or read-write disk. Then the parameter list is checked to ensure that the filename and filetype are present (calling error if not), and checks to see if the mode-letter is alphabetic, and if so whether a mode-number is given.

If the mode is alphabetic, ADTLKP is called to check for a disk whose mode-letter ADTM matches the parameter list. If the mode is '*' or equivalent (not alphabetic), ADTNXT is called to check for any available disk. An error return from ADTLKP or ADTNXT triggers a 'file not found' return from FSTLKP. On a successful return, FSTLKP checks to make sure the disk found is logged in (as either read-only or read-write). If not, the logic continues as described below, where the given FST entry was not found on the disk.

If the disk found by ADTLKP or ADTNXT is logged in, FSTLKP checks through the various FST hyperblocks in core to find a matching FST entry for the filename (if given in the parameter list) and filetype (if given). Note - If R1 was negative at entry to FSTLKP, the search for the given FST entry starts from where if left off as given by the ADTCHBA (current hyperblock address) and ADTCFST (current FST entry displacement) pointers in the active disk table for the given disk.

If the filename and filetype are both given and match explicitly, the file is deemed 'found' irrespective of any mode-number in the parameter list. If either (or both) was '*' in the parameter list, however, and the mode-number was given, then the mode-number in the parameter list must match the mode-number in the FST entry.

Thus, for example, a call to FSTLKP for "SOME FILE P5" would consider "SOME FILE P1" (on the P-Disk) a match even though the mode-number is wrong. (This logic is purposely provided to avoid misleading the user, since you cannot have two files on the same disk with same filename and filetype, but different mode numbers.) A search for "*FILE P5", however, would not consider "SOME FILE P1" to match, since the mode number differs.

(Note - this logic is now consistent throughout CMS, including some programs such as LISTF that search the FST hyperblocks themselves. That is, if the filename and filetype match explicitly, the mode number need not be correct for a match; but if the filename and/or filetype is '*' and the mode-number is given, then it must equal the FST mode-number to be considered a match.)

If FSTLKP finds the matching file on the given disk, it returns the addresses of the active disk table (ADT) and the FST entry in R0 and R1 as shown in exit conditions, and remembers where it found the file in the ADTCHBA and ADTCFST pointers in the ADT block.

FST entry not found on the disk

If the FST entry was not found on the disk just checked, FSTLKP checks the mode supplied in the P-List. If it was '*' (or equivalent), ADTNXT is called and the next disk (if any) is checked as above for the matching file.

If the mode, on the other hand, was alphabetic, ADTNXT is called to determine if another disk is available for checking. If so, the ADTMX extension-mode-letter is checked to see if it matches the mode given in the parameter list. If it does, this indicates that the new disk is a read-only extension of the one previously checked, and the given file is looked up on this disk. If found, successful return is given pointing to this disk and the FST entry found. If not, this process is repeated until a match is found, or until no more disk(s) with a matching ADTMX letter are found.

## FSTLKW

FUNCTION: To find a specified 40-byte FST entry within the FST tables for read-write disk(s); also, to find an empty 40-byte entry for use by FINIS.

CALLING SEQUENCE:

```
        L       R15,AFSTLKW         where AFSTLKW = V(FSTLKW)
        BALR    R14,R15
```

ENTRY REQUIREMENTS:

1. To search appropriate disk table(s) from the beginning:

R0 = Immaterial

R1 = Pointer to usual P-List (with sign-bit plus)

```
                DS       0F
PLIST           DC       CL8'       '     Immaterial
                DC       CL8'       '     FILENAME or '*'
                DC       CL8'       '     FILETYPE or '*'
                DC       CL2'  '          FILEMODE or '*'
```

2. To search appropriate disk table(s) picking up from where you left off previously, starting with next 40-byte FST entry:

R0 = Pointer to Active Disk Table

R1 = Pointer to usual P-List but with sign-bit negative

3. To find an empty 40-byte entry for a completed new output file (called only by 'FINIS')

R0 = Pointer to Active Disk Table

R1 = 0

EXIT CONDITIONS:

File Found:

R0 = Pointer to Active Disk Table

R1 = Pointer to (address of) 40-byte FST entry found or provided

R15 = 0 (and condition-code = 0)

File Not Found:

R0 = 0

R1 = 0 (with sign-bit negative)

R15 = 1 (and condition-code = 2)

Parameter List Error:

R0 = 0

R1 = 0 (with sign-bit negative)

R15 = 2 (and condition-code = 2)

CALLS TO OTHER ROUTINES:

ADTLKP, ADTNXT, FREE

CALLED BY (where known):

ALTER, ERASE, FINIS, STATEW, TFINIS, WRBUF, plus disk resident routines

MACROS USED:

ADT, FVS

OPERATION: FSTLKW checks to see if R1 = 0, indicating a special entry made by FINIS to find an empty 40-byte FST entry, or the regular entries made to locate a specific file.

If R1 is nonzero, FSTLKW checks the specific read-write disk (if the mode letter was alphabetic) or all read-write disks (if the mode letter was '*' or equivalent) for the given file. This search is almost identical to that performed by FSTLKP, except that only read-write disk(s) are examined, and read-only extension(s) via the ADTMX mode letter are not applicable. (See FSTLKP description for details.)

If R1 = 0, the location of the last file is determined from the ADTLHBA and ADTLFST pointers in the given active disk table. If the 40-byte entry at this location is empty (= 0), its address is returned. If not, a check is made to see if the next 40-byte entry in the same in-core hyperblock is available; if yes, its address is returned and the ADTLFST pointer updated by 40. If not, then a new 808-byte block is obtained from free storage, cleared, chained to the end of the last FST hyperblock, all appropriate pointers and counters updated, and the address of the first 40-byte entry in the new block returned to the caller.

In any event, the empty 40-byte entry is made available to the caller (FINIS), and all counters and pointers updated insofar as necessary.

Note: FSTLKW is included with the FSTLKP routine.

## UPDISK

FUNCTION: To reserve space on disk for rewriting a new copy of the User File Directory (UFD) on disk, and then to update the UFD on disk.

CALLING SEQUENCE:

```
         L       R15,AUPDISK         where AUPDISK = V(UPDISK)
         BALR    R14,R15
```

ENTRY REQUIREMENTS:

1. To update UFD when an output file is closed:

   R0 must point to Active Disk Table
   R1 must be positive and nonzero (any value)

2. To reserve records via TRKLKP for future use:

   R0 must point to Active Disk Table
   R1 = zero

3. To update UFD when records were reserved previously:

   R0 must point to Active Disk Table
   R1 must be negative (any value)

EXIT CONDITIONS:

Successful Update of UFD
R15 = 0 (and condition-code = 0)

Failure updating UFD
Transfer of Control to DISKDIE - see OPERATION for details.

CALLS TO OTHER ROUTINES:

FREE, FRET, TRKLKP, TRKLKPX, WRTK, DISKDIE

CALLED BY (where known):

| ALTER, DISK, ERASE, FINIS, LOGDISK, LOGIN, plus disk resident routines

MACROS USED:

ADT, FVS

OPERATION: UPDISK is the routine that updates the user file directory for a given disk., It is called in one of two ways: when FINIS has closed the last open output file for a given disk, it calls the first entry to update the UFD for that disk; when ERASE finds a file to be erased it first calls the second entry to reserve some tracks, then does its erasing function, then calls the third entry to write the new UFD on disk.

This logic makes possible what is called a "double directory" scheme, wherein the old directory still exists on disk until the MFD itself is finally rewritten on record 4 of the disk, completing the new directory. If the system is interrupted in any way in the middle of the process, the old directory is still intact, and any old files pointed to thereby are still intact.

UPDISK, then, has basically two steps. Entry (2) - called only by ERASE - does only the first step; Entry (3) - also called by ERASE - does only the second step.

The action taken by each step is as follows:

Step 1: An 800-byte buffer for the constructing of a new MFD is obtained from free storage, and the first 600 bytes are cleared. Available disk-addresses are obtained from TRKLKP for each FST hyperblock and stored in the MFD, just obtained, in sequential halfwords. Then, if there are no more than 215 bytes of significant data in the QMSK bit-mask, a sentinel of X'FFFF' is stored following these, and step 1 is complete. If the QMSK is larger than 215 bytes in length and there is significant data past the 215th byte, however, additional disk-addresses are obtained (as many as are needed), and stored following a sentinel of X'FFFD'. All needed disk-addresses for the new MFD have now been obtained and stored, without affecting any old data on disk.

If entry (2) was called by ERASE, UPDISK exists at this point.

Step 2: Now we continue in line, or enter here if entry (3) is called by ERASE.

UPDISK now cycles through the old MFD (if any) left in core by the file management programs, and returns old disk-addresses contained therein by calling TRKLKPX to return them to the QMSK table. The old MFD is then returned to free storage via FRET.

Now the FST hyperblocks in core, and any PQMSK extensions, are written on disk, calling WRTK, using the disk-addresses reserved above in Step 1. Next, all the disk counts (ADTNUM, ADTCYL, etc.), the first 215 bytes (or less) of the QMSK, the QQMSK, and the unit-type byte are moved to the MFD, and the MFD finally written back on record 4 of the disk, completing the new UFD on disk.

Finally, the significant part of the new MFD, still in core, including the disk-addresses of the FST hyperblocks, the FFFF or the FFFD sentinel, and the disk-addresses of the QMSK extension(s) are retained in core (in the high-numbered end of the buffer that was used for the new MFD), and the rest of the 800-byte buffer given back to free storage via FRET.

Notes: If a permanent disk error occurs writing the new UFD at any point, the UPDISK routine purposely transfers to the DISKDIE code (elsewhere), so that the old directory will be intact until the disk error can be corrected.

See also "UPUFD" and "LOGDISK", which still serve a useful purpose, and are retained for compatibility with existing programs. UPUFD and LOGDISK are included as entry-points in the UPDISK routine.

## UPUFD

FUNCTION: To close all CMS file(s), thereby updating the user file directory (UFD) for any disk(s) which had output files open.

CALLING SEQUENCE:

```
L       R15, = V(UPUFD)        (or equivalent)
BALR    R14,R15
```

ENTRY REQUIREMENTS:

No register requirements.

EXIT CONDITIONS:

R15 = 0

CALLS TO OTHER ROUTINES:

FINIS

CALLED BY (where known):

INIT, EXECTOR (disk-resident part of EXEC)

MACROS USED:

FVS

OPERATION: 'FINIS * * * ' is called to close any open files, in the course of which, if any output files are open, the file directories for the appropriate disk(s) are automatically updated by UPDISK.

UPUFD is called by INIT and EXECTOR to ensure that files are closed (and directories updated) in the event a user program left any files open on its completion.

Note: UPUFD is included with the UPDISK routine; it includes code common to LOGDISK, a nearly identical function also included with UPDISK.

## LOGDISK

FUNCTION: To close all CMS file(s), and (as an option) to update the file directory for the P-Disk in particular.

CALLING SEQUENCE:

```
        LA          R1, PLIST
        SVC         X'CA'
```

ENTRY REQUIREMENTS:

```
        R1 points to parameter list:
                    DS          0F
        PLIST       DC          CL8'LOGDISK'     (or CL8'FINUFD')
                    [DC         CL8'CHANGE']     optional parameter
```

EXIT CONDITIONS:

R15 = 0

CALLS TO OTHER ROUTINES:

FINIS, UPDISK

CALLED BY (where known):

LOGOUT

MACROS USED:

ADT, FVS

OPERATION: 'FINIS * * * ' is called to close any open files, in the course of which, if any output files are open, the file directories for the appropriate disk(s) are automatically updated by UPDISK.

If the 'CHANGE' option was specified, and the P-Disk is logged in and known to be read-write, the P-Disk file directory is then specifically updated via a call to UPDISK.

LOGDISK can be called by a user program if it is desired to close all files and update the directories thereby.

LOGDISK CHANGE can be called by a user if he specifically wants his P-Disk file directory updated; for example, if he has made a change to his in-core P-Disk directory via the DEBUG command and wishes the UFD on disk to be updated reflecting that change.

LOGDISK CHANGE is also called by LOGOUT, KILLEX, and KILLEXF.

Note: LOGDISK is included with the UPDISK routine; it includes code common to UPUFD, a nearly identical function also included with UPDISK.

## SYSGEN - INITSYS

SYSGEN is the entry point name, INITSYS is the filename.

FUNCTION: To generate a System Status Table (SSTAT), consisting of all the P2 files from the S-Disk, so that the location of all disk-resident commands is known by the various file management programs.

CALLING SEQUENCE:

```
        L       R15, = V(SYSGEN)
        BALR    R14,R15
```

ENTRY REQUIREMENTS:

S-Disk must be attached and ready.
SYSDEV in NUCON table must contain address of S-Disk
(No parameters provided by registers.)

EXIT CONDITIONS:

SSTAT table available in free storage; pointers to beginning and end stored in SYSREF table; ADT block for S-Disk initialized.

R15 = 0

CALLS TO OTHER ROUTINES:

FREE, READFST

CALLED BY:

INITSUB

MACROS USED:

ADT (references ADTS = Active Disk Table for S-Disk)

OPERATION: During the system initialization process, the program INITSUB calls the SYSGEN routine. SYSGEN initializes the free storage management scheme to have several pages of free storage available starting at one page above 'LAST'. Then SYSGEN points R0 to the ADTS Active Disk Table, which contains appropriate flags for reading the S-Disk as a read-only disk, with a parameter list of '**P2'. READFST then accordingly reads in the file directory of the S-Disk, but accepting only P2 files therefrom, storing the appropriate FST hyperblocks in the free storage area reserved. SYSGEN then reinitializes the free storage pointers as needed, obtains exactly the right amount of space for the System Status Table (using the information on number of files computed and saved by READFST), and moves the various 40-byte entries to the new SSTAT table, substituting 'SY' forP2 in each entry.

Size and length counts at the beginning of this table are initialized, a pointer at the end (STATEXT) cleared, and the addresses of SSTAT and STATEXT stored in the SYSREF part of the NUCON table. After appropriate reinitialization, SYSGEN then returns control to INITSUB.

When CMS is IPL'ed by name ("IPL CMS"), the SYSGEN function need not be performed, saving considerable input-output operations and time over the IPL'ing CMS by "IPL190".

## DISK SPACE MANAGEMENT ROUTINES

The disk space management routines allocate and release free disk storage; they include QQTRK, QQTRKX, TRKLKP, and TRKLKPX.

### QQTRK

FUNCTION: To allocate a 200-byte disk area to a calling program.

CALLING SEQUENCE:

```
L           R15, AQQTRK       where AQQTRK = V(QQTRK)
BALR        R14, R15
```

ENTRY REQUIREMENTS:

R1 must point to Active Disk Table block<br>
R13 must point to FVS area

EXIT CONDITIONS:

Normal Return

R1 contains disk-address of available 200-byte area (see Figure 24 for format)
R15 = 0 (and condition-code = 0)

No 200-byte area available (Error 1)

R1 = 0
R15 = 1 (and condition-code = 2)

Error by Caller (Error 2)

R1 same as at entry
R15 = 2 (and condition-code = 2)

CALLS TO OTHER ROUTINES:

TRKLKP, TRKLKPX

CALLED BY (where known):

WRBUF

MACROS USED:

ADT, FVS

OPERATION: Refer to "Disk Space Management" in Section 2 for general operation.

Notes: At entry, QQTRK checks that the pointer to the Active Disk Table block in R1 is positive and nonzero, that the disk referenced thereby is read-write, and that the user file directory, including the QQMSK table, is indeed in core. If not, an illegal halfword H'0002' is executed as a debugging aid (resulting in a program interrupt to DEBUG). If the program is continued (via "go" from DEBUG), an error code 2 is returned to the caller. In actual practice, the above has not been known to occur, as QQTRK is called only by WRBUF, which has been thoroughly debugged.

When calling TRKLKP, if QQTRK obtains an error 4 indicating "very few" records left, QQTRK returns the record just obtained via TRKLKPX before returning with error-code 1 to the caller, so that sufficient records are held in reserve to update the file directory in handling the full-disk situation.

Note the change in entry requirements for QQTRK - R1 must contain a pointer to the active disk table. (Formerly 24(R1) held the disk mode - P or T.)

## QQTRKX

FUNCTION: To make a 200-byte area that is no longer needed by one program available for allocation to another.

CALLING SEQUENCE:

```
        L       R15, AQQTRKX     where AQQTRKX = V(QQTRKX)
        BALR    R14, R15
```

ENTRY REQUIREMENTS:

R0 (rightmost 16 bits) must hold the disk address of the 200-byte disk area being returned. (See Figure 24 for format.)

R1 must point to Active Disk Table block.

R13 must point to "FVS" Area.

EXIT CONDITIONS:

Normal Return

R15 = 0 (and condition-code = 0)

Error by Caller (Error 2)

R15 = 2 (and condition-code = 2)

QQMSK is Full (Error 3)

R15 = 3 (and condition-code = 2)

CALLS TO OTHER ROUTINES:

TRKLKPX

CALLED BY (where known):

ERASE, WRBUF

MACROS USED:
ADT, FVS

OPERATION: Refer to "DISK SPACE MANAGEMENT" in Section 2 for general operation.

Notes: Like QQTRK, QQTRKX checks for errors by the caller, and an error 2 (with error halt first) is given if such errors occurred. In actual practice, this has not been known to occur, as QQTRKX is called only by ERASE and WRBUF, which have been thoroughly debugged.

If a user had an extremely large number of files and erased them sporadically, it is theoretically possible that the QQMSK table could become full from the other three parts of 800-byte records being returned for each returned 200-byte record. If this should occur, QQTRKX detects this condition, and does not permit the table to overflow. An error 3 is given, which is nonfatal. Processing continues, and a user's files are intact (except those intentionally erased). The QQMSK table would then contain some entries for which all four parts would not subsequently be found, but these would still be available for allocation by QQTRK.

Note the change in entry requirements for QQTRKX - R1 must contain a pointer to the active disk table. (Formerly 24(R1) held the disk mode - P or T).

QQTRKX is an entry-point in the QQTRK routine.

## TRKLKP

FUNCTION: To allocate an 800-byte disk area to a calling program.

CALLING SEQUENCE:

```
        L          R15, ATRKLKP       where ATRKLKP = V(TRKLKP)
        BALR       R14, R15
```

ENTRY REQUIREMENTS:

R1 must point to Active Disk Table block
R13 must point-to a save-area of at least eleven words

EXIT CONDITIONS:

### Normal Return

R1 contains disk-address of available 800-byte area
(See Figure 25 for format)

R15 = 0      (and condition-code = 0)

### Very Few Records Left (Error 4) - Nonfatal

R1 contains disk-address of available 800-byte area
(Same as above)

R15 = 4      (and condition-code = 2)

### Error by Caller (Error 2)

R1 same as at entry
R15 = 2      (and condition-code = 2)

CALLS TO OTHER ROUTINES:

None

CALLED BY (where known):

QQTRK, UPDISK, and WRBUF

MACROS USED:

ADT

OPERATION: Refer to "DISK SPACE MANAGEMENT" in Section 2 for general operation.

Notes: Like QQTRK and QQTRKX, TRKLKP checks for errors by the caller, and an error 2 (with error halt first) is given if such occurred. In actual practice, this has not been known to occur, as TRKLKP is called only by QQTRK, UPDISK, and WRBUF, which have been thoroughly debugged

TRKLKP now remembers (in ADT1ST) the displacement of the first fullword in the QMSK that has a zero-bit in it anywhere, to speed up searches after the first call to TRKLKP for any disk. (TRKLKPX of course maintains this word when records are returned.)

When the number of records remaining on the given disk no longer exceeds a reserve count (ADTRES) that is maintained by the file system, an error 4 (indicating very few records left) is returned. This feature enables WRBUF or QQTRK, on the one hand, to return the record via TRKLKPX and invoke the full-disk logic at KILLEXF, while UPDISK, on the other hand, can use the record for completing the new user file directory. (This is part of CMS's double directory scheme for maximum file integrity.)

Note the change in entry requirements for TRKLKP - R1 must contain a pointer to the active disk table. (Formerly 24(R1) held the disk mode - P or T).

## TRKLKPX

FUNCTION: To make an 800-byte disk area that is no longer needed by one program available for allocation to another.

CALLING SEQUENCE:

```
        L       R15, ATRKLKPX    where ATRKLKPX = V(TRKLKPX)
        BALR    R14, R15
```

ENTRY REQUIREMENTS:

R0 (rightmost 16 bits) must hold the disk address of the 800-byte disk area being returned. (See Figure 25 for format)

R1 must point to Active Disk Table block

R13 must point to a save-area of at least eleven words

EXIT CONDITIONS:

Normal Return

R15 = 0

Error by Caller (Error 2)

R15 = 2    (and condition-code = 2)

Out of Range 800-byte area returned (Error 5)

R15 = 5    (and condition-code = 2)

Already Clear 800-byte area returned (Error 6)

R15 = 6    (and condition-code = 2)

CALLS TO OTHER ROUTINES:

None

CALLED BY (where known):

ERASE, QQTRK, QQTRKX, UPDISK, WRBUF

MACROS USED:

ADT

OPERATION: Refer to "DISK SPACE MANAGEMENT" in Section 2 for general operation.

Notes: Like QQTRK, QQTRKX, and TRKLKP, TRKLKPX checks for errors by the caller, and an error 2 (with error halt first) is given if such an error occurred. In actual practice, this has not been known to occur, as TRKLKPX is called only by ERASE, QQTRK, QQTRKX, UPDISK and WRBUF, which have been thoroughly debugged.

TRKLKPX now maintains (in ADTIST) the displacement of the first fullword in the QMSK that has a zero-bit in it anywhere (this being used by TRKLKP for speeding up the search of the QMSK table).

Note the change in entry requirements for TRKLKPX - R1 must contain a pointer to the active disk table. (Formerly 24 (R1) held the disk mode - P or T).

TRKLKPX is an entry-point in the TRKLKP Routine.

## ACTIVE FILE TABLE MANAGEMENT (AFT) ROUTINES

Four routines are used for active file table management - ACTLKP, ACTNXT, ACTFREE, and ACTFRET. These are described on the following pages.

### ACTLKP

FUNCTION: Find the active file table block whose filename, filetype, and mode match the one supplied by the caller.

CALLING SEQUENCE:

```
        L        R15, AACTLKP     where AACTLKP = V(ACTLKP)
        BALR     R14, R15
```

ENTRY REQUIREMENTS:

R0 = 0: Start search at beginning of Active File Table.

or

R0 = nonzero: Given present Active File Table Block, resume searching at next AFT Block (if any).

R1 must point to Parameter List as usual:

```
         DS       0F
PLIST    DC       CL8'        '    Immaterial
         DC       CL8'        '    FILENAME
         DC       CL8'        '    FILETYPE
         DC       CL2'  '          MODE
```

R13 must point to a save-area of at least ten words (normally would point to FVS = DISK$SEG area).

EXIT CONDITIONS:

Match Found

R1 = Address of Matching Active File Table Block<br>
R15 = 0 (and condition-code = 0)

Match Not Found

R1 same as at entry<br>
R15 = 1 (and condition-code = 2)

CALLS TO OTHER ROUTINES:

None

CALLED BY (where known):

ALTER, ERASE, FINIS,. POINT, RDBUF, STATE, STATEW, TFINIS, and WRBUF.

MACROS USED:

AFT

OPERATION: If R0 = 0 at entry, ACTLKP starts searching the Active File Table starting with the first block (at FVSAFT). If R0 = nonzero at entry, the given value in R0 is taken as the address of the present AFT block, and searching commences with the next block (if any). (This feature facilitates searching for more than one matching file by the calling function.)

ACTLKP examines each block in the active file table to see whether it has a filename, filetype, and mode matching those in the parameter list. If the given filename and/or filetype was specified as '*' in the parameter list, a matching filename or filetype, respectively, is assumed. If the mode was specified as either '*', binary 0, blank (X'40'), or X'FF', the mode is assumed correct. If not, the given mode letter must equal the mode in the Active File Table block for a match. (It is not necessary to check the mode number.)

Exit conditions are returned as specified above, with the address of the matching block (if any) returned in R1. The condition-code is set per R15, for convenience of the caller (who can omit the usual LTR instruction before checking the return-code).

## ACTNXT

FUNCTION: Find the next (or first) AFT block in the Active File Table.

CALLING SEQUENCE:

```
L       R15, AACTNXT     where AACTNXT = V(ACTNXT)
BALR    R14, R15
```

ENTRY REQUIREMENTS:

| | |
|---|---|
| R1 = 0: | Find First Active File Table Block. |
| or | |
| R1 = nonzero: | Given present Active File Table Block, find next AFT Block (if any). |

EXIT CONDITIONS:

| | | |
|---|---|---|
| R1 = 0: | No more blocks in Active File Table. | |
| R1 = nonzero: | R1 holds address of next (or first) AFT Block. | |
| R15 = 0 | (in any event) | (and condition-code = 0) |

CALLS TO OTHER ROUTINES:

None

CALLED BY (where known):

ERASE

MACROS USED:

AFT

OPERATION: If R1 = 0 at entry, the address of the first Active File Table block (FVSAFT) is returned.

If R1 = nonzero at entry, the given value in R1 is taken as the address of the present AFT block, and the pointer to the next AFT block (if any) is loaded into R1 (with the high-order byte stripped off).

In either event, no check is made as to whether the AFT block whose address is returned contains an active file (this being done by the caller).

ACTNXT can be used (for example, by ERASE) for scanning through the Active File Table for particular conditions other than might be found by calling ACTLKP.

Note: ACTNXT is an entry-point in the ACTLKP Routine.

## ACTFREE

FUNCTION: Find an empty block in the Active File Table, or add a new block from free storage to the Active File Table, if necessary, and place a file status table entry (if given) into the AFT block.

CALLING SEQUENCE:

```
L       R15,AACTFREE     where AACTFREE = V(ACTFREE)
BALR    R14, R15
```

ENTRY REQUIREMENTS:

R0 must point to Active Disk Table.
R1 points to FST entry (or = 0).
R11 must point to Parameter List belonging to Caller (P-List provided to RDBUF, WRBUF, or POINT, etc.).
R13 must point to a save-area of at least ten words (normally would point to FVS = DISK$SEG area).

EXIT CONDITIONS:

R1 = Address of Active File Table Block used for newly created.
R15 = 0 (and condition-code = 0).

CALLS TO OTHER ROUTINES:

FREE

CALLED BY (where known):

POINT, RDBUF, WRBUF

MACROS USED:

AFT, ADT, FSTB

OPERATION: ACTFREE is called by RDBUF, WRBUF, or POINT for placing a 40-byte file status table entry in the Active File Table, after a call to FSTLKP or FSTLKW to determine the location of the FST entry. (For a new file being created by WRBUF, R1 = 0 at entry indicating a new FST entry is about to be created.)

ACTFREE scans through the Active File Table looking for an empty slot, determined by the AFTUSED bit of AFTFLG for a given block being 0. If no empty block is found, a new block is obtained from free storage via FREE and chained onto the end of the Active File Table; and a bit (AFTFSF) is set to indicate that the block is in free storage.

After the empty block is found or created, ACTFREE clears the first 104 bytes, stores necessary pointers, and moves the 40-byte FST entry (if provided) into the space provided at AFTFST.

The mode-letter stored at AFTM in the Active File Table is carefully chosen from that of the caller's parameter list (given by R11 at entry), or the ADTM or ADTMX mode given by the Active Disk Table. The STATE function uses the same algorithm for choosing this mode-letter.

The result of the choice of mode-letter facilitates the feature of a read-only extension of a given disk. For example, if an A-Disk is a read-only extension of a P-Disk and the caller's Parameter List specified the A-mode, the mode stored will be A; but if the caller specified P or '*', the mode stored will be P.

Note: ACTFREE is an entry-point in the ACTLKP Routine.

## ACTFRET

FUNCTION: Remove an AFT block from the Active File Table, returning it to free storage if appropriate.

CALLING SEQUENCE:

```
        L          R15, AACTFRET    where AACTFRET = V(ACTFRET)
        BALR       R14, R15
```

ENTRY REQUIREMENTS:

R1 must hold the address of the AFT block being returned.

R13 must point to a save-area of at least ten words (normally would point to FVS = DISK$SEG area).

EXIT CONDITIONS:

Returned block was in AFT Table

R15 = 0 (and condition-code = 0)

Returned block was not in AFT Table

R15 = 1 (and condition-code = 2)

CALLS TO OTHER ROUTINES:

FRET

CALLED BY (where known):

ERASE, FINIS, WRBUF

MACROS USED:

AFT

OPERATION: The Active File Table is searched to find the AFT block matching the address provided by the caller. When it is found, the AFTFLG flag-byte and AFTPFST pointer are cleared (it is not necessary to clear all the other information). If the AFT block was in free storage, it is given back via FRET, and the chain patched accordingly.

If the AFT block address provided in R1 at entry time is not found in the Active File Table, this indicates a programming bug on the part of the caller. As a debugging aid, an illegal halfword H'0001' is executed, resulting in a program interrupt, normally to the DEBUG command, to warn the user that an illegal call has been issued. If the program is continued (via "go" from DEBUG), an error code 1 is returned to the caller. In actual practice, the above has not been known to occur, as ACTFRET is called only by ERASE, FINIS, and WRBUF, which have been thoroughly debugged.

Note: ACTFRET is an entry-point in the ACTLKP Routine.

## ACTIVE DISK TABLE MANAGEMENT (ADT) ROUTINES

Two routines are used for active disk table management - ADTLKP and ADTNXT. These are described on the following pages.

### ADTLKP

FUNCTION: Find the active disk table block whose mode matches the one supplied by the caller.

CALLING SEQUENCE:

```
        L       R15,AADTLKP             where AADTLKP = V(ADTLKP)
        BALR    R14,R15
```

ENTRY REQUIREMENTS:

R1 must point to Parameter List as usual:

```
        DS      0F
PLIST   DC      CL24'       '  Immaterial
        DC      CL2'   '       Mode (for example, P, T, S, A, B, C)
```

EXIT CONDITIONS:

```
        Active Disk Table Block Found
                R1  = Address of matching Active Disk Table Block
                R15 = 0         (and condition-code = 0)
        Active Disk Table Block Not Found
                R1 same as at entry
                R15 = 1         (and condition-code = 2)
```

CALLS TO OTHER ROUTINES:

None

CALLED BY (where known):

ALTER, FSTLKP & FSTLKW, LISTF, LOGIN, RDTK & WRTK, RELEASE, STAT, TAPE, & WRBUF, plus disk resident routines

MACROS USED:

ADT

OPERATION: ADTLKP searches through the Active Disk Table starting at IADT to find an ADT block whose mode-letter, as given by ADTM, matches that of the caller. If a matching ADT block is found, its address is returned in R1 as indicated above. If not, an error-return is given. In either event, the condition-code is set per R15, for convenience of the caller (who can omit the usual LTR instruction before checking the return-code).

ADTLKP is called by many programs and functions to determine the address of the Active Disk Table pertaining to a given disk (for example, P-Disk, T-Disk, S-Disk).

## ADTNXT

FUNCTION: Find the next (or first) ADT block in the Active Disk Table.

CALLING SEQUENCE:

```
L       R15,AADTNXT             Where AADTNXT = V(ADTNXT)
BALR    R14,R15
```

ENTRY REQUIREMENTS:

```
R1 = 0:          Find First Active Disk Table Block
 or
R1 = nonzero:    Given present Active Disk Table Block, find next ADT Block
                 (if any)
```

EXIT CONDITIONS:

```
Active Disk Table Block Found
          R1  = Address of ADT Block
          R15 = 0            (and condition-code = 0)
Active Disk Table Block Not Found (none left)
          R1  = 0
          R15 = 1            (and condition-code = 2)
```

CALLS TO OTHER ROUTINES:

None

CALLED BY (where known):

FSTLKP & FSTLKW, LISTF, LOGIN, STAT

MACROS USED:

ADT

OPERATION: If R1 = 0 at entry, the address of the first Active Disk Table Block at IADT is returned.

If R1 = nonzero at entry, the given value in R1 is taken as the address of the present ADT block, and the pointer to the next ADT block (if any) is loaded into R1. If there is no ADT block left, error code 1 is given in R15 as shown under exit conditions above.

In any event, no check is made whether any disk is currently logged in for the disk table returned (these checks being performed by the caller).

ADTNXT is used by various routines and commands for scanning through the chain of active disks for conditions to be satisfied, for example LISTF and STAT. ADTNXT is also used by FSTLKP to check for a disk which may be a read-only extension of another.

Note: ADTNXT is an entry-point in the ADTLKP routine.

## DISK HANDLING FUNCTION PROGRAMS

The following text describes the routines which handle I/O operations for direct access devices: RDTK and WRTK. Both entry points reside in the module DISKIO.

### RDTK

FUNCTION: To read one or more 800-byte records (quarter-tracks) from disk, or to read one 200-byte record (sixteenth-track) from disk.

CALLING SEQUENCE:

```
        L       R15, ARDTK      where ARDTK = V(RDTK)
        BALR    R14, R15
```

ENTRY REQUIREMENTS:

```
        R1 must point to Parameter List as follows:
        DS      0F
PLIST   DC      A(BUFF)         Core-address of buffer into which data
                                is to be read
        DC      F'  '           Size of above buffer (byte count)
        DC      A(DISKAD)       Core-address of list containing
                                disk addresses of all record(s)
                                to be read
        DC      A(DSKTBL)       Core-address of (pointer to) Active
                                Disk Table block pertaining to disk
                                to be read
```

Notes: (1) The format of each disk address supplied in the list for each 800-byte record is a halfword block number (from 1 up) as shown in Figure 25.

(2) If the buffer-size (byte count) is zero, a 200-byte record (sixteenth-track) is to be read. Figure 24 shows the format of this type of disk address.

(3) For compatibility with possible existing programs that may call RDTK directly, RDTK will accept a disk mode (for example, CL2'P' or CL2'SY', etc.) in place of the pointer to the active disk table; ADTLKP is called in this case to find the equivalent disk.

EXIT CONDITIONS:

Normal Return

R15 = 0 (and condition - code = 0)

Error Returns

R15 = 1: No active-disk-table block found by ADTLKP where mode-letter was supplied (see note 3)

R15 = 2: Permanent I/O error reading disk

R15 = 3: Permanent SIO error attempting to read disk

R15 = 5: Disk-address = 0 (a programming bug by caller) or not within disk limits (Sense-byte = X'81' on error analysis)

R15 = 7: Attempt to read into CMS nucleus below FREAR

CALLS TO OTHER ROUTINES:

ADTLKP, FREE, FRET, WAIT, IOERR

CALLED BY (where known):

ERASE, FINIS, RDBUF, READFST, READMFD, TFINIS, and WRBUF, plus disk resident routines

OPERATION: RDTK determines whether or not the calling program wishes to read a sixteenth track. If it does not, RDTK checks to see if the byte count exceeds 800 bytes, implying more than one 800-byte record to be read. If so, free storage is obtained, and chained CCW's are used insofar as practical to read as many records as possible with one SIO to the disk. If not more than 800, no free storage is necessary as a standard CCW package (in DIOSECT) is used. In any event, whether or not repeated SIO's are necessary, the data is read into the designated buffer, free storage is returned if necessary, and return is made to the caller. If the byte count exceeds 800, it need not necessarily be an exact multiple of 800.

If the calling program wishes to read a sixteenth track, RDTK calls the FREE function program to obtain an 800-byte buffer. It then reads the quarter track containing the desired sixteenth track into the buffer. (This quarter track is pointed to by the first disk address in the list.) Next, RDTK extracts the desired sixteenth track from the 800-byte buffer and moves it to the 200-byte buffer specified in the parameter list. It then calls the FRET function program to release the 800-byte buffer into which the quarter track was read and returns to the calling program. Sixteenth tracks are reserved for first chain link areas and should not be used for other purposes.

Note: If the 200-byte sixteenth-track is the first 200 bytes of an 800-byte buffer, RDTK reads 200 bytes directly into the designated core-area instead of obtaining free storage and moving the 200 bytes later. Otherwise, the reading of 200-byte sixteenth tracks is as described above.

## WRTK

FUNCTION: To write one or more 800-byte records (quarter-tracks) on disk. or to write one 200-byte record (sixteenth-track) on disk.

CALLING SEQUENCE:

```
        L       R15, AWTRK      where WRTK = V(WRTK)
        BALR    R14, R15
```

ENTRY REQUIREMENTS:

R1 must point to Parameter List as follows:

```
        DS      0F
PLIST   DC      A(BUFF)         Core-address of buffer from which
                                data is to be written
        DC      F'  '           Size of above buffer (byte count)
        DC      A(DISKAD)       Core-address of list containing disk
                                addresses of all record(s) to be written
        DC      A(DSKTBL)       Core-address of (pointer to) Active Disk
                                Table block pertaining to disk to be written
```

Notes: (1) The format of each disk address supplied in the list for each 800-byte record is a halfword block number (from 1 up) as shown in Figure 25.

(2) If the buffer size (byte count) is zero, a 200-byte record (sixteenth-track) is to be written. Figure 24 shows the format of this type of disk address.

(3) For compatibility with possible existing programs that may call WRTK directly, WRTK will accept a disk mode (for example, CL2'P', etc.) in place of the pointer to the active disk table; ADTLKP is called in this case to find the equivalent table.

EXIT CONDITIONS:

Normal Return

R15 = 0 (and condition-code = 0)

Error Returns

R15 = 1: No Active-Disk-Table block found by ADTLKP where mode-letter was supplied (see Note 3 above)

R15 = 2: Permanent I/O error writing disk

R15 = 3: Permanent SIO error attempting to write disk

R15 = 4: Attempt to write on system disk (mode = S)

R15 = 5: Disk-address = 0 (a programming bug by caller)
or not within disk limits
(Sense-byte = X'81' on error analysis)

R15 = 6: Attempt to write on read-only disk

CALLS TO OTHER ROUTINES:

ADTLKP, FREE, FRET, WAIT, IOERR

CALLED BY (where known):

FINIS, READMFD, TFINIS, UPDISK, WRBUF, plus disk resident routines

OPERATION: WRTK determines whether or not the calling program wishes to write a sixteenth track. If it does not, WRTK checks to see if the byte count exceeds 800 bytes, implying more than one 800-byte record to be written. If so, free storage is obtained, and chained CCW's are used insofar as practical to write as many records as possible with one SIO to the disk. If not more than 800, no free storage is necessary, as a standard CCW package (in DIOSECT) is used. In any event, whether or not repeated SIO's are necessary, the data is written from the designated buffer, free storage is returned if necessary, and return is made to the caller. If the byte count exceeds 800, it need not necessarily be an exact multiple of 800.

If the calling program wishes to write a sixteenth track, WRTK obtains 800 bytes of buffer space into which to read the quarter track (in which the sixteenth track (200 bytes) of data is to be placed). It then reads that quarter track into the buffer. (The disk address for the quarter track is pointed to by the first disk address in the list.) Next, WRTK moves 200 bytes of data from the buffer pointed to by the parameter list into the appropriate sixteenth track location in the buffer containing the quarter track. WRTK then writes the updated quarter track back onto the disk at its original location, releases the 800 bytes it used as a buffer, and returns to the calling program.

Note: If the 200-byte sixteenth-track is the first 200 bytes of an 800-byte buffer and the remaining 600 bytes are not used (see note accompanying Figure 24.) WRTK writes 200 bytes directly from the designated core-area instead of the read, move, and write procedure described above.

## DIRECT I/O

Input/Output activity to disk may be initiated by a specially coded DIAGNOSE instruction if:

(1) CMS is running on a version of CP-67 which supports the I/O via Diagnose feature
AND
(2) a read or write of 800 bytes (or less) is performed.

If both of these apply, the I/O is performed as follows:

Instead of CMS doing the SIO, checking for success, then calling WAIT, and processing the interrupt thru INTIO, back to WAIT & back to DISKIO, and checking the CSW for successful completion, CMS does the following:

a diagnose instruction with a code X'18' is issued to signal CP-67 to perform the designated I/O operation. R4 contains the device-address, and R8 points to the standard CCW chain:

```
SEEK CCHH
SEARCH CCHHR
TIC *-8
READ/WRITE
NOP
DC X'CCHHR'
```

If the operation is successful, CP returns to CMS with a condition-code 0; this indicates that the operation was successfully started AND completed. DISKIO then continues as if the regular procedure had been followed, at the point AFTER the SIO would have been performed, WAIT had been called, the CSW had been checked for errors, and a check had been made for the handling of a request of more than 800 bytes.

Upon return to CMS, the following condition codes and error codes are returned:

condition-code (CC) = 0: I/O complete with no errors.

CC = 1: SIO failed, CSW stored.
(CSW+4 & CSW+5 returned to user)

CC =2: Either an attempt to write on a read-only disk
(program-check returned)
or
other I/O error on completion
CSW (8 bytes) returned
(sense bytes available if a 'SENSE' is issued)

CC =3: Not attached; neither 2314 nor 2311; or invalid DIAGNOSE call

Error-code returned in R15, as follows:

1 = Not attached
2 = Device is neither 2314 nor 2311
3 = Pointer to user's CCW-string not dbl-word aligned
4 = SEEK/SEARCH arguments not within user core
5 = Read/write CCW neither read (06) nor write (05)
6 = Read/write byte-count = 0
7 = Read/write byte-count greater than 4096
8 = Read/write buffer not within user core
9 = Condition-code 2 (busy) on actual SIO as attempted by CP
10 = Condition-code 3 (not operational) on actual SIO as attempted by CP

DISKIO will then invoke the IOERR error recovery procedures to attempt to rectify the operation.

Unsuccessful Disk I/O: When an I/O operation is not started successfully or does not complete successfully, RDTK or WRTK branches to the I/O error recovery program, IOERR. The calling parameter list — described under "Input/Output Service Routines" — is built in free storage by RDTK or WRTK. RDTK or WRTK requests IOERR to perform recovery on the specific DASD device, and IOERR returns the status of the completed recovery procedure. Figure 39, "Relationship of IOERR to RDTK/WRTK", shows the actions taken by RDTK or WRTK on the various errors from IOERR.

| IOERR Return Code | RDTK and WRTK Action |
|---|---|
| 1. E10CC and R15 = Successful Retry (X'00') | 1. Release free storage<br>2. Return to calling program with successful return code (X'00') |
| 2. E10CC and R15 = Device now available (X'AF') | 1. Release free storage<br>2. Retry I/O operation |
| 3. E10CC and R15 = Unknown Device (X'8F')<br>= Error in IOERR (X'9F')<br>= Device not available (X'BF')<br>= Invalid Parameter List (X'CF')<br>= Unknown I/O error (X'DF') | 1. Release free storage<br>2. Return to calling Program with I/O error Reading/Writing (X'02') or start I/O (X'03') error code |
| 4. E10CC and R15 = RDTK/WRTK error code | 1. Release free storage<br>2. Return to calling program with this code (X'02', X'03', X'05', or X'06'). |

Figure 39. Relationship of IOERR to RDTK/WRTK

## UNIT RECORD HANDLING FUNCTION PROGRAMS

The following text describes the routines that handle I/O operations or unit record devices (printer and card reader/punch).

### CARDIO

FUNCTION: To read cards from the card reader and punch cards on the card punch

ATTRIBUTES: Nucleus resident

CALLING SEQUENCE:

```
      LA    1,PLIST
      SVC   X'CA'
      .
      .
      .
PLIST DC    CL8' /CARDRD\
                 \CARDPH/
      DC    X'flag'
      DC    AL3 (buffer)
      DC    H' number of bytes to read'
      DC    H'0' number of bytes read
```

EXIT CONDITIONS:

Normal Return:

R15 = 0

Error Return

Card Reader

| | |
|---|---|
| R15 = 1 | End of File |
| R15 = 2 | Never Read |
| R15 = 3 | Unknown Error |
| R15 = 4 | Not Operational |
| R15 = 5 | Count Not Equal to Requested Count |

Card Punch:

| | |
|---|---|
| R15 = 2 | Unrecoverable Unit Check |
| R15 = 3 | Unknown Error |
| R15 = 4 | Not Operational |

CALLS TO OTHER ROUTINES:

None

CALLED BY (where known):

OFFLINE

OPERATION: CARDIO first determines if a card read or card punch is to be executed. If a read is to be executed, CARDIO checks the high-order byte of the second word in the parameter list to see if it is an extended PLIST. If flag is a X'80', the extended PLIST is in effect and CARDIO reads the 2 high-order bytes in the third word of the parameter list to determine the number of bytes to be read. It then scans the device table (DEVTAB) in the nucleus constant area for the entry RDR1, the symbolic name of the reader. When it finds this entry, it uses the device entry contained to construct a CCW. It then issues a start I/O operation. If the operation is started successfully, CARDIO calls the WAIT function program to wait for an interruption from the card reader. When the interruption is received, CARDIO stores the number of bytes read in the two low order bytes of the third word in the parameter list. If the flag in the PLIST is not X'80', then 80-byte record length is assumed. With successful operation completion, CARDIO returns to the caller. If the operation was not successfully completed, CARDIO signals an error and returns to the caller.

If, after the SIO is issued, the CSW is stored, CARDIO checks for an error. If an error exists, CARDIO signals it and returns to the caller. If the CSW was stored because of a busy condition, CARDIO waits (via WAIT), if necessary for the device and condition, and then retries the operation. If an end-of-file condition caused the CSW to be stored, CARDIO indicates this and returns to the caller.

If the reader was busy when it issued the SIO, CARDIO calls the WAIT function program to wait for the condition to clear; it then retries the operation. The operation to punch cards is the same as above with the exception of the entended PLIST (for a special reader). The 80-byte record length is in effect for card punch.

## PRINTR - PRINTIO

Filename: PRINTIO nucleus-resident, re-entrant code

Entry Point: PRINTPR

FUNCTION: To print a line on the printer with appropriate carriage control.

CALLING SEQUENCE:

```
        LA     1, PLIST
        SVC    X'CA'
        .
        .
        .
PLIST   DC     CL8'PRINTR'
        DC     A(        )     Address of buffer containing line to be printed
        DC     A(        )     size of buffer
```

EXIT CONDITIONS:

Normal Return:

R15 = 0 Successful Operation

Error Returns:

| | |
|---|---|
| R15 = 1 | Unrecoverable Unit Check |
| R15 = 2 | Invalid Parameter List from Caller |
| R15 - 3 | Not Operational |
| R15 = 5 | Unknown Error |

CALLS TO OTHER ROUTINES:

FREE, FRET, IOERR

CALLED BY (WHERE KNOWN):

OFFLINE

OPERATION: The operation of PRINTIO depends upon the nature of the carriage control character in the first byte of the output buffer.

First Character Blank: If the first character in the buffer is a blank, indicating that the caller wishes to print a line and space a line, PRINTIO issues a start I/O instruction to print the buffer and space a line. With a successfully started I/O operation, PRINTIO calls the WAIT function programs to wait for the completion of the operation. Upon return to PRINTIO with a successful operation, PRINTIO returns with a normal return code to the calling program.

First Character Zero: If the first character in the buffer is a zero (hex 'F0'), indicating that the caller wishes to space a line, print the line in the buffer, then space another line, PRINTIO issues a SIO instruction to space one line. With a successful SIO, PRINTIO calls the WAIT function program to wait for an interrupt from the printer. Upon return to PRINTIO with a successful operation, PRINTIO proceeds to print the contents of the buffer and space a line in the same manner as when the first character in the buffer is a blank.

First Character One: If the first character in the buffer is a one (hex 'F1'), indicating that the caller wishes to eject to a new page, print a line, and space a line, PRINTIO issues a start I/O to eject to a new page. It then proceeds in the same manner as it does when the first character in the buffer is a zero.

First Character Machine Code: After determining that the first character is a machine code -- by comparing to the entries in a table containing all possible machine codes -- PRINTIO executes a channel program containing the user machine code as the operation code of the first channel command words. With a successful SIO instruction, PRINTIO calls the WAIT function program to wait for an interrupt from the printer. Upon return to PRINTIO with a successful operation, PRINTIO returns to the calling program with the normal return code.

First Character Extended USA Carriage Control Character: After determining that the first character is an extended USA carriage control character -- by comparing it to the valid USA character -- PRINTIO translates it into machine code. PRINTIO then executes a channel program containing the translated control character as the operation code of the first channel command word. With a successful SIO, PRINTIO calls the WAIT function program to wait for an interrupt from the printer. Upon return to PRINTIO with a successful operation, PRINTIO checks to see if the caller desired only a carriage control operation (i.e., the caller did not provide any data in the buffer). If so, PRINTIO returns to the calling program with a normal return code. If the caller desired a line to be printed, rather than only a carriage control operation, PRINTIO proceeds to print the contents of the buffer and space a line as it does when the first character in the buffer is a blank.

First Character Undefined: If the character is not a blank, zero, one, machine code, or extended USA carriage control character, PRINTIO substitutes a "Write, Space 2 after Print" operation code for the undefined user code -- i.e., it prints the contents of the buffer and spaces one line in the same manner as it does when the first character in the buffer is a blank.

Unsuccessful I/O Operations: When an I/O operation is not started successfully or does not complete successfully, PRINTIO branches to the I/O error recovery program, IOERR with the calling parameter list described under "Input/Output Service Routines". This PLIST is built in free storage by PRINTIO. PRINTIO thereby requests IOERR to perform recovery on the printer, and IOERR returns to PRINTIO the status of the recovery procedure. Figure 40 relates the return code from IOERR and the corresponding action taken by PRINTIO.

| IOERR Return Code | |
|---|---|
| 1. E10CC = Successful Retry (X'00')<br>and<br>R15 | 1. Release free storage<br>2. Return to calling program with successful return code (X'00') |
| 2. E10CC = Device Now Available (X'AF')<br>and<br>R15 | 1. Release free storage<br>2. Retry I/O operation |
| 3. E10CC = Unknown Device Type (X'8F')<br>and<br>R15 = Error in IOERR (X'9F'<br>= Device Not Available (X'BF')<br>= Invalid Parameter List (X'CF')<br>= Unknown I/O Error (X'DF') | 1. Release free storage<br>2. Return to calling program with unknown return code (X'05') |
| 4. E10CC = PRINTIO's error code<br>and<br>R15 | 1. Release free storage<br>2. Return to calling program with this code (X'02', X'03', or X'05') |

Figure 40. Relationship of IOERR to PRINTIO

## TAPE HANDLING FUNCTION PROGRAM

The following text describes the program that performs I/O operations to tape.

### TAPEIO

FUNCTION: To (1) read or write tape records, (2) rewind the tape, and (3) write an end-of-file marker on tape.

CALLING SEQUENCE:

```
      LA    1, PLIST
      SVC   X'CA'
      .
      .
      .
PLIST DC    CL8'TAPEIO'
               \ READ     )
      DC    CL8' ) WRITE   (  '
               ) REWIND   (
               ( WRITEOF  \
      DC    CL4'        '      symbolic tape name
      DC    A(          )      buffer address for read/write
      DC    F'          '      buffer size
      DS    1F                 number of bytes actually read
                               (returned to caller by TAPEIO)
```

EXIT CONDITIONS:

Normal Return

| | |
|---|---|
| R15 = 0 | Successful Operation |

Error Return

| | |
|---|---|
| R15 = 1 | Invalid Function |
| R15 = 2 | End-of-file or End-of-tape |
| R15 = 3 | Permanent I/O Error |
| R15 = 4 | Illegal Symbolic Unit |
| R15 = 5 | Tape is not attached |
| R15 = 6 | Tape is file protected |
| R15 = 7 | Serious tape error |
| R15 = 8 | Channel busy on start of I/O operation |

CALLED BY (WHERE KNOWN):

Disk resident routines

OPERATION: TAPEIO starts an I/O operation (read, write, rewind or write end-of-file) appropriate to the calling program's request (READ, WRITE, REWIND, or WRITEOF). It then calls the WAIT function program to wait for an interruption from the tape device. When an interruption occurs, control is returned to TAPEIO, which analyzes it. If a unit exception (end-of-file) caused the interruption, TAPEIO places a code indicating this condition into register 15 and returns (via SVCINT) to the calling program. If a unit check caused the interruption, TAPEIO retries a maximum of ten times to correct the error. (For each retry, TAPEIO backspaces the tape, starts the I/O, and calls the WAIT function program.) If unsuccessful, it signals the error and returns to the caller. If a channel end caused the interruption, the operation was successful, and TAPEIO returns to the caller.

Note: When running CMS under CP/67, the user must request the operator to mount his tape on an available drive and attach that drive to this configuration at the address that corresponds to the specified symbolic name, "TAPn".

ERROR RECOVERY and HANDLING

Once the SIO to the particular tape address is given, a specific branch is taken, depending on the condition code:

0. the I/O has started, and a call to CMS WAIT is made to handle the terminating interrupt. When the interrupt occurs, the CSW is examined to determine if a unit check, unit exception, or no device end was present. For no device end, WAIT is reentered until the DE is received. Unit exception will return to the calling routine with a code of 2 in register 15. For a unit check, a sense is issued to the device. If a data error caused the unit check, the operation is repeated 10 times. If intervention is required, a message is typed indicating this. If file is protected, again a message is typed. If the operation is successful, control is returned to the calling program.

1. CSW stored — the unit check process described above is executed. If a weird permanent error is encountered, operation is terminated.

2. busy — wait is entered until device is free.

3. not operational — a message is typed and control returned to the caller.

## TERMINAL HANDLING FUNCTION PROGRAMS

The following text describes the programs that perform I/O operations involving user terminals. These function programs include CONWAIT, TYPLIN, TYPE, and WAITRD.

### CONWAIT

FUNCTION: To place the computer in the wait state until all terminal I/O requests have been satisfied.

CALLING SEQUENCE:

```
        LA           1, PLIST
        SVC          X'CA'
         .
         .
         .
PLIST   DC           CL8'CONWAIT'
```

OPERATION: CONWAIT checks the number of read/write requests (NUMRDWRT) remaining in the read/write stack. If there are some, it calls the WAIT function program to place the computer in the wait state until a terminal I/O request is completed. Upon return from WAIT, CONWAIT again checks to see if there are any remaining read/write requests. If there are, it calls WAIT. CONWAIT repeats this procedure until all terminal read/write requests have been satisfied. At this time, it returns (via SVCINT) to the calling program.

### TYPLIN – CONWRITE

FILENAME – CONWRITE

ENTRY POINT – TYPLIN

FUNCTION: To write an output line on the console with terminal blanks deleted and an automatic carriage return added.

CALLING SEQUENCE:

```
        LA           1, PLIST
        SVC          X'CA'
         .
         .
         .
PLIST   DC           CL8'TYPLIN'
        DC           AL1(1)               terminal number
        DC           AL3(MSG)             address of output line
        DC      C'{B}'             see note
                  {K}

        DC      AL3(EMSG-MSG)      message length
        .
        .
        .
MSG     DC      C'                 message to be typed
EMSG    EQU     *
```

Note: Write codes —

Not implemented

OPERATION: TYPLIN obtains the output line and truncates it after the last nonblank character. (If the kill-right-end flag is set, truncation will occur at or before 72 characters.) If the kill-typing flag is set, TYPLIN exits without typing the line. If the kill-typing flag is not set but there is not enough room to insert the output line into the read/write stack area contained in the nucleus, TYPLIN creates room by calling the WAIT function program until the number of read/write entries is reduced to one. TYPLIN then constructs a CCW package for this output line and inserts it into the read/write stack. If at this time there are no requests in the read/write stack, TYPLIN branches to the start I/O routine in CONINT to initiate the I/O operation. On return to TYPLIN with a successful SIO, TYPLIN returns to the caller. If there are requests in the read/write stack, TYPLIN does not start the write operation; it links the CCW package into the read/write stack as the last entry and returns to the calling program.

## TYPE

FUNCTION: To write an output line on the console without terminal blanks being deleted and without an automatic carriage return being added.

CALLING SEQUENCE: See that for TYPLIN

OPERATION: TYPE operates in essentially the same manner as TYPLIN, except that TYPE does not truncate the line after the last nonblank character and does not provide an automatic carriage return.

## WAITRD — CONREAD

FILENAME: CONREAD

ENTRY POINT: WAITRD

FUNCTION: To read an input line from the terminal and make it available to the caller.

CALLING SEQUENCE:

```
        LA       1, PLIST
        SVC      X'CA'
         .
         .
         .
PLIST   DC       CL8'WAITRD'
        DC       AL1(1)
        DC       AL3(INPBUF)          address of 130-byte input buffer
                  ( U )
                  | V |
        DC       C'{ S }              see note
                  | T |
                  ( X )
        DC       AL3(    )            byte count of input message stored here
         .
         .
         .
INPBUF  DS       130C                 input buffer
```

Note: Read codes —

U = perform clean-up, uppercase translation, and blank-fill

V = perform clean-up and uppercase translation

S = perform clean-up and blank-fill

T = perform clean-up only

X = leave input line exactly as is

OPERATION: WAITRD checks the validity of the read code. If invalid, it signals the error and returns to the caller. If valid, WAITRD checks to see if there are any finished reads. If there are, the input line pointed to by the first CCW package in the finished read stack is obtained and edited according to the read code, the CCW package is removed from the stack, the pointer to the first finished read (FSTFINRD) is updated to point to the next CCW package (if any) in the stack, the number of finished reads counter (NUMFINRD) is decremented, and the line is made available to the requester.

If there are no finished reads, the pending read stack is checked to see if there are any pending reads. If there are, WAITRD calls the WAIT function program to wait until a pending read finishes. (An interruption will mark the completion of the read, and the CCW package in the pending read stack will be moved by the interrupt handler to the finished read stack.) When the read is finished, WAITRD proceeds as if there had been an entry in the finished read stack when the read request was made.

When a request for an input line is made and there are neither finished reads nor pending reads, WAITRD checks to see if there is more than one entry in the read-write stack. If there is, WAITRD waits, by calling WAIT, until either a read is finished or until only one entry remains in the read-write stack. (At this point, all entries in the read-write are for write requests. As each of these writes is completed, an interruption occurs and the next write in the stack is started; eventually, the read-write stack will be reduced to a single entry. This will terminate the wait. A finished read will terminate the wait only if, during the depletion of the writes in the read-write stack, the user hits the attention button, indicating that he wishes to enter a new command. If this is the case, the terminal interruption program will place a read CCW package into the pending read stack, place a similar package at the start of the read/write stack, and start the read operation. When the operation is complete, the interrupt handler will move the CCW package from the pending read stack to the finished read stack.)

If a finished read terminates the wait, processing proceeds as if there had been a finished read at the time the calling program made the read request. If the wait is terminated by the depletion of the read/write stack to a single entry, WAITRD constructs a CCW package for a read and places it into the pending read stack as the only entry. If, by this time, the last write in the read-write stack has been completed (that is, there are no entries in the read-write stack), the CCW package is also made the first (only) entry in the read-write stack. WAITRD then starts a read and waits for its completion. When the read is finished, WAITRD proceeds as if there had been an entry in the finished read stack when the read request was made. If the last write in the read-write stack has not been completed, the CCW package is chained onto the read-write stack as the last (in this case the second) entry. WAITRD then waits for the read to finish. (The terminal interruption program will actually start the read and move the CCW package from the pending read stack to the finished read stack.) When the read is finished, WAITRD proceeds as if there had been an entry in the finished read stack when the read request was made.

## USER PROGRAMMED DEVICE HANDLING ROUTINE

FUNCTION: To perform I/O operations for an I/O device.

CALLING SEQUENCE:

User defined

EXIT CONDITIONS:

User defined

CALLS TO OTHER ROUTINES:

User defined

CALLED BY:

User defined

OPERATION: A description of the operation of user programmed device handling routines is given in "User Input/Output Operation" under "I/O Operations".

INPUT/OUTPUT SERVICE ROUTINES

There are two I/O service routines in CMS. These allow both the user and the system to synchronize I/O operations (via the WAIT function program), and allow the system to have centralized error recovery (via the IOERR function program).

SYNCHRONIZE ROUTINE

The WAIT function program allows both the user and the system to synchronize I/O operations.

## WAIT

FUNCTION: To place the computer in the wait state until the completion of an I/O operation on a particular device.

CALLING SEQUENCE:

```
LA      1, PLIST
SVC     X'CA'
.
.
.
```

```
PLIST  DC    CL8'WAIT'
       DC    CL4'    '    symbolic device name (CON1, DSK1,
                          PCH1, etc.)
       DC    CL4'    '    symbolic device name (CON1, DSK1,
                          PCH1, etc.)
        .
        .
        .
       DC    F'0'         signifies end of device list
       DS    1F           symbolic name of interrupting device
                          is stored here (by WAIT) after IOINT
                          returns to WAIT.
```

OPERATION: WAIT looks up the first symbolic device name in the user-defined interrupt table (if any - as initialized by the HNDINT function program), or in the device table (DEVTAB) in the nucleus constant area (NUCON). If the device is not in either table, WAIT returns to the caller with error code 1.

If the device was found, WAIT puts the pseudo-wait bit in the corresponding device table entry on. WAIT repeats this procedure for each symbolic device name on the parameter list. It then loads a PSW with the wait bit on. This causes the system to enter the wait state until an I/O operation on one of the specified devices is completed. After the I/O operation is completed, IOINT returns control to WAIT, which places the symbolic name of the interrupting device (that is, the device on which the I/O operation was completed) in the last entry in the parameter list and returns control to the caller.

In some cases, some special logic is used where the interrupt-handling was initialized by the HNDINT function program. See NONTERMINAL I/O — HNDINT for a full description of these special procedures.

WAIT can be called via SVC, or can also be called by BALR if the caller's system-mask is x'00'. For disk resident programs doing their own I/O operations WAIT should be called via SVC.

## CENTRALIZED ERROR RECOVERY PROGRAM

### IOERR

Purpose: IOERR is a table processing program providing CMS with centralized I/O error recovery. The devices for which error recovery is provided are the 2311 and 2314 direct access storage devices and the 1403 printer.

Modules: IOERRSUP, IOERREC, IOERRMES,
IONUTAB

Called By: DISKIO, PRINTIO

General Operation: When either PRINTIO or DISKIO (i.e., the system routines that perform the I/O operations on the devices supported by error recovery) encounters an I/O error, a parameter list is built in free storage and IOERRSUP (the error recovery interface) is called. IOERRSUP initializes its free storage (containing a trace table for debugging, a WAIT parameter list to synchronize the error recovery I/O operations, and a read list for bringing a copy of IONUTAB into free storage if necessary). IOERRSUP then replaces the device's standard interrupt processing routine with ERPRINTD (the error recovery interrupt processing routine).

If the error occurred during the initialization of the operation, SIOERRRT is entered. If the error occurred during the completion of the operation, CIOERRRT is entered.

SIOERRRT contains three routines corresponding to the three possible errors during I/O initiation. SIOBUSY waits for the channel or subchannel to become available and returns to either PRINTIO or DISKIO. SIONOTOP (not operational) calls IOERRMES to type an error message, and returns to either PRINTIO or DISKIO with a not-operational return code in register 15. SIOCSW (channel status word stored) does one of four things depending on the condition causing the CSW to be stored. If the device is busy, SIOBUSY is entered. If the device contains a pending interrupt from a previously completed operation, SIOCSW initializes the return code to indicate to PRINTIO or DISKIO that the device is now available and enters CLEANUP. If the started operation caused an error, CIOERRRT is entered. If the started operation is an unchained immediate command and the immediate operation is error free, normal processing continues; however, if the immediate command caused an error, the routine containing the I/O operation enters IOERRSUP.

CIOERRRT contains four routines. CIOQUIES (quiesce I/O) analyzes the machine's state on entry into error recovery. CIONUCK (unit check) performs a sense operation if a unit check has occurred. RTLOADER initializes the device type table (IONUTAB for direct access, the copy of IONUTAB in free storage called IOERTAB for the printer), and passes control to IOERRREC. IOERREC searches the device type table for the error and executes the routine for the recovery action. When completed, IOERREC exits to RTLOADER. When a permanent I/O error has occurred, RTLOADER calls IOERRMES. With a successful recovery, RTLOADER exits to CLEANUP. CLEANUP releases free storage and returns to either PRINTIO or DISKIO with register 15 containing the status of the operation and/or device

During error recovery, interrupts from the device in error are processed by ERPINTD.

DISKIO and PRINTIO

Purpose: I/O operations directed to the 2311 or 2314 direct access storage device are processed by DISKIO. Upon a start or completion error, DISKIO builds a parameter list for IOERR to attempt recovery. On return from IOERR, DISKIO analyzes the return code. PRINTIO handles error I/O operations directed to the 1403 printer in the same way.

Entry Conditions (when calling IOERR)

| | |
|---|---|
| R1: | Address of PLIST |
| R14: | Return address in Routine Containing the I/O operation in error |
| R15: | Address of IOERRSUP |

Exit Conditions (on return from error recovery)

R15: Return Code

| | |
|---|---|
| X'00': | Successful Retry |
| X'8F': | Unknown Device Type |
| X'9F': | Error in IOERR |
| X'AF': | Device Now Available |
| X'BF': | Device Not Available |
| X'CF': | Invalid Parameter List |
| X'DF': | Unknown I/O Error |
| Other: | Permanent I/O Error |

Operation: When either a start or completion I/O error occurs, DISKIO (or PRINTIO) issues the EIOPL macro to (1) obtain free storage for the parameter list, (2) build the parameter list by using the CSW, I/O old PSW, and information from the device table in NUCON, (3) set a flag indicating if this is start or completion error, and (4) enter IOERRSUP.

On return from IOERRSUP, DISKIO (or PRINTIO) analyzes the return code. If the recovery was successful, DISKIO (or PRINTIO) releases free storage and returns to the calling program. If register 15 contains X'AF', DISKIO (or PRINTIO) releases free storage and retries the operation. If register 15 indicates a permanent I/O error DISKIO (or PRINTIO) releases free storage and returns to the calling program with register 15 indicating a permanent I/O error.

IOERRSUP

Purpose: IOERRSUP is (1) the interface between I/O error recovery and either DISKIO or PRINTIO, and (2) the supervisor of error recovery.

Modules: BEGERP, CLEANUP, CIOERRRT, STOERRRT

Operation: IOERRSUP controls the processing of the I/O error recovery procedures, and communicates with the calling routine (i.e., either DISKIO or PRINTIO.

Module 1: BEGERP

Purpose: BEGERP performs the initialization for the I/O error recovery.

Operation: BEGERP establishes addressability and saves registers 0 to 15. The plist is then checked. If it is invalid, BEGERP exits to CLEANUP. If the parameter list is valid, BEGERP initializes free storage for the work area by the ERPTRWT macro, and for the error queue with the ERPERRQ macro. BEGERP then turns on the IOERR active flag in the nucleus, and sets up registers that IOERR will use. BEGERP activates ERPRINTD. If the error occurred on a state I/O, BEGERP exits to SIOERRRT; if the error occurred on an I/O completion, the exit is to CIOERRRT.

Module 2: SIOERRRT

Purpose: SIOERRRT processes errors that occur on a start I/O operation.

Entry Points: SIOBUSY, SIONOTOP, SIOCSW

Operation: SIOERRRT examines the condition code and branches to one of the three entry points depending on its contents. If the condition code is 3, SIOERRRT branches to SIONOTOP; if 2, to SIOBUSY; and if 1, to SIOCSW.

SIOBUSY: SIOBUSY executes on a busy condition during a start I/O operation. SIOBUSY initializes ERPWT (the WAIT list) with the device name(s), then enters the WAIT function program. On return from WAIT, SIOBUSY indicates to the calling program that the device is available, and exits to CLEANUP.

SIONOTOP: SIONOTOP executes on a not operational condition during a start I/O operation. SIONOTOP calls QUEINSER to insert the I/O error into the error queue, and exits to RTLOADER to indicate the error to the user.

SIOCSW: SIOCSW executes on a CSW stored condition during a start I/O operation. If the device or control unit is busy, SIOCSW exits to SIOBUSY. If a pending interrupt is indicated in the device field of the CSW (either busy plus attention, busy plus device end, busy plus control unit end, busy plus channel end, or busy plus channel and device end), SIOCSW sets the return code to indicate that the device is available, and exits to CLEANUP. If the error is an I/O error indicated by other information in the device and channel fields of the CSW, SIOCSW calls QUEINSER to enter the I/O error information into the error queue, and exits to CIOERR1 (an entry point in CIOERRRT).

Module 3: CIOERRRT

Purpose: CIOERRRT processes errors that occur on the completion of an I/O operation.

Entry Points: CIOQUIES, CIOUNCK, CIOERR1, RTLOADER

Operation: CIOERRRT contains the code to process I/O errors encountered during the completion of an I/O operation. After QUEINSER is called to insert the I/O error into the error queue, CIOERRRT branches to CIOQUIES.

CIOERR1: CIOERR1 is entered from SIOCSW. It branches to CIOQUIES.

CIOQUIES: CIOQUIES exists to provide a hook whereby additional processing logic may be added to IOERR.

CIOUNCK: CIOUNCK obtains the sense information for a unit check error. It the error is not a unit check error, CIOUNCK branches to RTLOADER. If the I/O error is (1) a unit check, (2) a successful sense operation, and (3) not channel 9 on the printer, CIOUNCK moves the sense information from ERPSENIN (the input area) to ERPSEN (the sense field in the error queue), then branches to RTLOADER. If the I/O error is (1) unit check, (2) successful sense operation, and (3) channel 9 on the printer, CIOUNCK sets the return code to indicate a successful error recovery, and exits to CLEANUP. If the I/O error is a unit check and an unsuccessful sense operation, the sense operation is retried 10 times. If the retry is unsuccessful, an internal IOERR sense operation failure is indicated, and CIOUNCK exits to RTLOADER. If the retry is successful and it is not channel 9 on the printer, the sense information from ERPSENIN is moved to ERPSEN and CIOUNCK exits to RTLOADER. If the retry is successful and it is channel 9 on the printer, the return code is set to indicate a successful error recovery and CIOUNCK exits to CLEANUP.

RTLOADER: RTLOADER initializes the recovery table (IONUTAB or IOFRTAB) and passes control to IOERREC and/or IOERRMES. RTLOADER first determines which copy of the recovery tables is to be used. If the error is on a DASD device, IONUTAB is used. If the error is not on a DASD device but there is also an I/O error while accessing the disk resident IOFRTAB, typing and I/O error message, or other internal error, IONUTAB is used. Otherwise, if it is not a DASD error, free storage is obtained for IOFRTAB.

If the I/O error has had a successful recovery, RTLOADER calls IOERREC, then exits to CLEANUP. If the I/O error has had an unsuccessful recovery, RTLOADER calls IOERRREC, then calls IOERRMES, then exits to CLEANUP. If an INTERVENTION

REQUIRED occurs during a retry operation, RTLOADER calls IOERRREC, then IOERRMES. On return, if the recovery is successful, RTLOADER exits to CLEANUP; if unsuccessful, it enters IOERRMES, then CLEANUP.

Module 4: CLEANUP

Purpose: CLEANUP terminates the I/O error recovery procedure.

OPERATION: If the error device is the console and write operations exist in the read/write stack, a restart console is executed, and operation continues as follows: In all other instances, execution of CLEANUP begins at this point. The error device's normal interrupt processing routine replaces ERPINTD, and, if free storage was used by IOFRTAB, it is returned along with all other free storage. The IOERR active flag in the nucleus is turned off, the user's registers are restored, and CLEANUP returns to DISKIO (or PRINTIO).

IOERRREC

Purpose: IOERRREC locates the entry for the error device in IONUTAB (or IOFRTAB) for the address of the recovery procedure and branches to the recovery procedure. On return, IOERRREC exits to RTLOADER.

OPERATION: IOERRREC uses the LOCATE macro to obtain the address of the device type table, then uses LOCATE again to find the error entry within the table. IOERRREC then branches to the address of the recovery procedure then exits to RTLOADER. If the error entry is not located, a non-identifiable error is indicated, and IOERRREC exits to RTLOADER.

IOERRMES

Purpose: IOERRMES (the message routine) uses information in IONUTAB (or IOFRTAB) to create a unique error message. The error return code is placed in register 15. For INTERVENTION REQUIRED and DEVICE NOT OPERATIONAL, the terminal user is notified.

OPERATION: IOERRMES uses the information from IONUTAB (or IOFRTAB) to format a 320 character error message for all unrecoverable errors. If the user is not to be notified of the error, IOERRMES exits to RTLOADER.

If the message is either (1) INTERVENTION REQUIRED or (2) NON-EXISTENT DEVICE, the user is notified with an abbreviated form of the formatted message. IOERRMES types the message either by using TYPLIN or issuing an SIO to the console.

Using TYPLIN: The message constants and variables are moved to the output message area in free storage. If the message is NON-EXISTENT DEVICE, the message is typed via TYPLIN, and IOERRMES exits to RTLOADER. If the message is INTERVENTION REQUIRED, the message is typed via TYPLIN, WAIT is called to wait for device end on the device.

Using SIO: The message constants and variables are moved to the message area in free storage. The console's standard interrupt routine is replaced by ERPINTD. The message is typed via an SIO to the console. WAIT is called to wait for the message to complete typing (and for INTERVENTION REQUIRED, to also wait for device end on the device requiring intervention). The console's interrupt processing routine replaces ERPINTD, and IOERRMES exits to RTLOADER. If the message does not type successfully, the address and length of the message is passed to the calling program along with an error return code. IOERRMES then exits to RTLOADER.

## IONUTAB — IOFRTAB

Purpose: IONUTAB is a table containing the recovery procedures and error messages by device type and error.

OPERATION: There are two copies of IONUTAB. One copy, IOFRTAB, is disk resident and is brought into free storage when errors are encountered on devices other than DASD devices. IONUTAB is nucleus resident, and is used for DASD errors.

## NORMAL/ERROR OVERRIDE FUNCTION PROGRAMS

The following text describes the programs that store trace information when the CMS SVC is used to transfer control from one routine to another. These function programs include .RDERR, .RESNRM, .RESERR, .RESUME, .RTLERR, .STEROV, and .STNOV.

### .RDERR

FUNCTION: To move either the normal or error override information saved by SVCINT to a work area.

CALLING SEQUENCE:

```
        LA      1,PLIST
        SVC     X'CA'
        .
        .
        .
PLIST   DC      CL8'.RDERR'
        DC      A(          )     address of 56-word work area
```

OPERATION: .RDERR first checks the address of the work area to ensure that it is legitimate. If not, it signals the error and returns (via SVCINT) to the calling program. If the address of the work area is legitimate, it checks the first byte of the error override save area. If this byte indicates that the save area does, in fact, contain valid error override information, .RDERR moves the contents of this save area to the work area and returns to the calling program. (SVCINT will set the first byte of the error override save area to indicate that it contains valid error-override information only when error overriding is in effect and an error was encountered during execution of the called program.) If the error override save area does not contain valid error override information, .RDERR similarly checks the normal override save area. (SVCINT will set the first byte of the normal override save area to indicate that it contains valid normal-override information only when normal overriding is in effect and no error was encountered during execution of the called program.) If the normal override save area does contain valid normal-override information, .RDERR moves this information to the work area and returns to the calling program. If the normal override save area does not contain valid normal override information, .RDERR similarly checks the error save area. If this save area does contain valid error information, .RDERR moves it to the work area and returns to the caller. If it does not, .RDERR signals an error and returns to the caller.

Note: The check of the error save area is done to allow a user access to saved error information in his non-override error program. For example, if the user has provided an address constant after the SVC X'CA' and if (1) error overriding is not in effect, and (2) an error is encountered during execution of the called program, SVCINT will pass control to the portion of code indicated in the address constant. If, in this portion of code, the user wishes to analyze the saved error information, he can call .RDERR and obtain it.

## .RESNRM

FUNCTION: To make a normal return from a user-written normal override program to a program that called another via an SVC X'CA'.

CALLING SEQUENCE:

```
         LA     1, PLIST
         SVC    X'CA'
         .
         .
         .
PLIST    DC     CL8'.RESNRM'
         DC     A(      )         address of new normal override
                                  program, zero(0), or one(1)
         DC     A(      )         address of new error override
                                  program, zero(0), or one(1)
```

OPERATION: .RESNRM checks the first byte of the normal override save area to determine whether it contains valid normal override data. If it does not, .RESNRM generates a program interruption that causes control to be passed to the DEBUG command program. If it contains valid data, .RESNRM sets register 15 to zero to ensure that SVCINT will make a normal return to the calling program. (Upon return from the called program, register 15 may or may not be zero.) It then checks the normal return address saved by SVCINT when the SVC X'CA' was executed to make sure it is legitimate. (This address is contained in the normal save area.) If not legitimate, .RESNRM generates a program interruption. If legitimate, it checks to see whether the kill-override flag has been turned on. (This will have been turned on by the terminal interrupt handling program, CONSI, if the user had entered a KO request.) If it is on, .RESNRM sets the normal and error override switches saved by SVCINT to zero. This will inhibit subsequent overriding. It then moves the data in the normal override save area into the normal save area, restores the override switches (now zero) to their original locations (NRMOVR and ERROVR), and returns to SVCINT, which will make the normal return to the caller.

If the kill-override switch is not on, .RESNRM checks the second parameter in the parameter list. If it is zero, indicating that the caller wishes to inhibit further normal overriding, .RESNRM sets the saved normal override switch to zero. If it is one, indicating that the caller wishes to continue normal overriding and use the same normal override program, .RESNRM does not alter the saved normal override switch. If the second parameter is neither zero nor one, indicating that the caller wishes to continue normal overriding but use a new normal override program, .RESNRM stores the address of that program in the saved normal override switch. .RESNRM then repeats a a similar procedure for the third parameter that indicates how the caller wishes to treat subsequent error overriding. .RESNRM then moves the data in the normal override save area to the normal save area, restores the appropriately set override switches to their original locations and returns to SVCINT.

## .RESERR

FUNCTION: To make an error return from a user-written error override program to the appropriate error location specified by a program that called another via an SVC X'CA'.

CALLING SEQUENCE:

```
         LA     1,PLIST
         SVC    X'CA'
         .
         .
         .
PLIST    DC     CL8'.RESERR'
         DC     A(        )        address of new normal
                                   override program, zero(0),
                                   or one(1)
         DC     A(        )        address of new error override
                                   program, zero(0), or one(1)
```

OPERATION: .RESERR checks the first byte of the error override save area to determine whether it contains valid error override data. If it does not, .RESERR generates a program interruption. If it contains valid data, .RESERR checks the contents of register 15, as returned by the called program. If register 15 contains a zero, indicating no errors, .RESERR assumes that a normal return is to be made and proceeds as described for the .RESNRM function program starting at the point where .RESNRM checks the validity of the normal return address. If register 15 is not zero, .RESERR checks to ensure that the error return address is valid. If it is not valid, it generates a program interruption. If the error return address is valid, .RESERR proceeds as described for .RESNRM, starting at the point where .RESNRM checks the kill-override flag.

## .RESUME

FUNCTION: To make either a normal or error return from either a normal or error override program to a program that called another via an SVC X'CA'.

CALLING SEQUENCE:

```
         LA     1, PLIST
         SVC    X'CA'
         .
         .
         .
PLIST    DC     CL8'.RESUME'
         DC     A(      )       address of new normal override program,
                                zero(0), or one(1)
         DC     A(      )       address of new error override program,
                                zero(0), or one(1)
```

OPERATION: .RESUME checks the validity of the error override information. If valid, it proceeds in the same manner as described for .RESERR, starting at the point where .RESERR checks the contents of register 15. If the error override information is invalid, .RESUME checks the validity of the normal override information. If invalid, .RESUME generates a program interruption. If valid, it proceeds in the same manner as described for .RESERR, starting where .RESERR checks the contents of register 15. (In this case, register 15 will contain a zero and a normal return will be made.)

## .RPLERR

FUNCTION: To replace either the normal override information in NOVSAV, the error override information in ERVSAV, or the error information in ERRSAV with the data supplied by a calling program in a 56-word area. (The data in the 56-word area should be arranged in the same manner as it is in NOVSAV, ERVSAV, or ERRSAV.

CALLING SEQUENCE:

```
         LA     1, PLIST
         SVC    X'CA'
         .
         .
         .
PLIST    DC     CL8'.RPLERR'
         DC     A(         )    address of 56-word area containing
                                replacement data
```

OPERATION: . RPLERR first checks to ensure that the address of the 56-word area is legitimate. If not, it signals the error and returns (via SVCINT) to the caller. If the address is valid, . RPLERR checks to ensure that the first byte of the area contains either 1 (for valid error information), 2 (for valid normal override information), or 3 (for valid error override information). If the first byte does not contain 1, 2, or 3, . RPLERR signals an error and returns to the caller. If the first byte contains either a 1, 2, or 3, . RPLERR checks to ensure that the normal return address provided in the 56-word area is legitimate. If not, it signals an error and returns to the caller. If the normal return address is legitimate, . RPLERR similarly checks the error return address. If this is invalid, . RPLERR signals the error and returns to the caller. Then, if the information provided is error information, . RPLERR moves it to the error save area (ERRSAV) and returns to the calling program. If the information provided is normal override information, . RPLERR moves it to the normal override save area (NOVSAV) and returns to the caller. If the information provided is error override information, . RPLERR moves it to the error override save area (ERVSAV) and returns to the calling program.

## .STEROV

FUNCTION: To set the error override switch (ERROV) in SVCINT to the address of an error override handling program.

CALLING SEQUENCE:

```
         LA     1, PLIST
         SVC    X'CA'
         .
         .
         .
PLIST    DC     CL8'.STEROV'
         DC     A(      )          address of error override handling program
```

OPERATION: . STEROV checks the address provided in the address constant to ensure that it is a valid one. If the address is invalid, it signals the error and returns (via SVCINT) to the calling program. If the address is valid, . STEROV moves it to the error override switch (ERROVR) within SVCINT and returns to the calling program.

Note 1: The SETOVER and SETERR command program calls . STEROV and passes it the address of the CMS error override handling program (HNDLERR). However, a user may call it and pass the address of his own error override handling program.

Note 2: If the address constant is zero, the error override switch will be turned off, thus inhibiting error overriding activity.

## .STNOV

FUNCTION: To set the normal override switch (NRMOVR) in SVCINT to the address of a normal override handling program.

CALLING SEQUENCE:

```
         LA       1, PLIST
         SVC      X'CA'
         .
         .
         .
PLIST    DC       CL8'.STNOV'
         DC       A(       )       address of normal override handling program
```

OPERATION: .STNOV checks the address provided in the address constant to ensure that it is a valid one. If the address is invalid, it indicates the error and returns (via SVCINT) to the calling program. If the address is valid, .STNOV moves it to the normal override switch (NRMOVR) within SVCINT and returns to the calling program.

Note 1: The SETOVER command program calls .STNOV and passes it the address of the CMS normal override handling program (HNDLNRM). However, a user may call it and pass the address of his own normal override program.

Note 2: If the address constant is zero, the normal override switch will be turned off, thus inhibiting normal override activity.

DEBUGGING FUNCTION PROGRAM

DEBDUMP Function

FUNCTION: DEBDUMP enables a user to dump his virtual core from within an executing program.

CALLING SEQUENCE:

```
         L        R15 = V(DMPEXEC)
         LA       R1, DUMPLIST       Dumplist DSECT in caller program
         BALR     R14, R15
```

Note: DUMPLIST can be found in the routine called 'GENSECT'

ENTRY REQUIREMENTS: Register 1 points to the following 'PLIST'.

```
DUMPLIST   DC    A (GPRS)          Address of Reg Save Area (0-15)
           DC    A (LOWCORE)       Address of Lowcore Save Area (0-256)
           DC    A (BEGIN)         Starting Dump Location (IN HEX)
           DC    A (END)           Ending Dump Location (IN HEX)
           DC    A (FPREGS)        Address Floating Point Reg Save Area 0-6
           DC    A (DUMPTITLE)     Address of DUMPTITLE
```

Register 0 contains the PSW location requested for the dump.

EXIT CONDITIONS: There are no error codes supplied by DEBDUMP

CALLS TO OTHER ROUTINES: None

OPERATION: DEBDUMP first obtains the address of DUMPTITLE and sends the title to the virtual printer. It then obtains the addresses of the general-purpose register save area, the floating-point register save area, the PSW requested in register 0, the CSW and the CAW. The contents of the save areas are sent to the virtual printer with appropriate headings.

The address of lowcore is then obtained. If this starting location is less than X'100' that portion or all of lowcore from 0 to X'100' up to and including the ending dumping location is sent to the virtual printer. The printer is closed after the last line is dumped. Control is passed back to the caller program.

Note: If the title is not filled in by the user a default TITLE of 'DUMP REQUESTED BY USER' is used. The user can specify a title up to 132 bytes in length.

## CONTROL AND SERVICE FUNCTION PROGRAMS

The control and service function programs provide facilities for using the interval timer, for searching libraries for undefined symbols, for closing libraries, for generating a copy of the CMS Nucleus onto disk, for IPL'ing CMS from the disk, for placing an input line into a parameter list format, and for waiting for an I/O completion from a specified device. These function programs included GETCLK, LDRLIBE, PRTCLK, IPLDISK, SCAN, SETCLK, WAIT, and BARE67.

### GETCLK

Filename: CMSTIME, Nucleus

FUNCTION: To calculate the time elapsed since SETCLK set the timer.

CALLING SEQUENCE:

```
         L      15, = V(GETCLK)
         BALR   14,15
```

OPERATION: GETCLK subtracts the current value of the timer from that saved by SETCLK in TIME. It then converts the result to hundredths of a second, stores it in registers 0 and 1, and returns to the calling program.

### KILLEXF

FUNCTION: To kill execution if disk is full

ATTRIBUTES: Nucleus resident. Entry point — KILLEXF in Log

CALLING SEQUENCE:

R2 = pointer to active disk table from WRBUF via R15.

OPERATION:

1. Calls CONWAIT to wait for any stacked up or busy typeouts.
2. Types '*** 2-DISK (CCU) IS FULL ***' (with mode letter and number filled in).
3. Then does same as KILLEX.

## LIBEPACK

Filename: LDRLIBE, Nucleus

FUNCTION: To search libraries for undefined symbols, and to close libraries.

CALLING SEQUENCE:

```
        L       15, = V(LIBEPACK)
        BALR    14,15
```

OPERATION: If LIBEPACK is not entered to close libraries, it calls TXTLIB to obtain a list of the entry points contained in each open text library. Each undefined symbol specified by the calling routine is checked against the lists of entry points. If a match is found, POINT is called to set the read item number in the file status table for the library containing the matching entry point. LIBEPACK then returns control to the caller.

If LIBEPACK is entered to close text libraries, it issues calls to FINIS, passing each time a library name as a parameter. When all text libraries are closed , LIBEPACK returns to the caller.

## PRTCLK

Filename: CMSTIME, Nucleus

FUNCTION: To calculate and type at the terminal the time elapsed since SETCLK set the timer.

CALLING SEQUENCE:

```
        L       15, = V(PRTCLK)
        BALR    14,15
```

OPERATION: PRTCLK subtracts the current value of the timer from that saved by SETCLK in TIME. It then converts the result to hundredths of a second, calls the TYPLIN function program to type the elapsed time at the terminal, and returns to the calling program.

## IPLDISK-INITIPL

Filename: INITIPL, Nucleus-Resident

Entry point: IPLDISK

FUNCTION: To write and read an IPL-able copy of the absolute core image of the CMS nucleus.

CALLING SEQUENCE: The nucleus routine INITSUB will transfer control to IPLDISK whenever the nucleus is IPL'ed from the card-reader using the NUCLEUS text deck. Upon subsequent IPL'ing of the CMS Nucleus from disk, the IPL sequence gives control to IPLDISK, which passes control back to INITSUB, after the nucleus is read into core.

OPERATION: Upon entry to IPLDISK, the DO flag will determine whether a read (R) or a write (W) of the nucleus should be executed. For a write, questions are asked to determine device address and starting cylinder; a sense operation will determine the device type (2311, 2314). Onto cylinder zero is then written an IPL control sequence of 24 bytes, followed by the IPLDISK program itself with the DO switch set to read. Now, all CCW string and search arguments are initialized. The routine SETCORLC will increment arguments as the I/O operation progresses. The routine DIOCS will execute it actual I/O commands. Onto disk will be written core pages 0-11 (hex) and pages 3D-3F in 820-byte blocks, one SIO execution per track. After writing, control is returned to INITSUB.

When the disk is subsequently IPL'ed to get a copy of the nucleus, the IPLDISK program is read in, receives control, and proceeds to read the blocks of nucleus data. The address of the IPL'ed device is saved in IPLDEV within NUCON. The Read phase constructs and executes its own CCW string. Control is passed to INITSUB upon completion of reading in core pages 0-11 and loader maps into pages 1D-1F.

## SCAN (Nucleus)

FUNCTION: To place an input line into the format of a parameter list.

CALLING SEQUENCE:

```
LA      1, BUFF            address of buffer containing input
                           line in register 1.
L       15, = A(SCAN)
BALR    14, 15
```

Note: The first word of the buffer contains a count of the number of characters in the input line. The remainder of the buffer contains the input line.

OPERATION: SCAN scans the input line for strings of nonblank characters and places them into successive doublewords in the command buffer (COMBUF) included with the SCAN subroutine. (The first string is placed into the first doubleword of COMBUF, the second into the second, etc.) If a particular string exceeds eight characters, SCAN truncates it at eight. If a string is less than eight characters, SCAN fills out the right portion of the corresponding doubleword in COMBUF with blanks. SCAN flags the end of the resultant parameter list by placing a doubleword with all bits on after the last applicable doubleword in COMBUF. SCAN returns the address of the resultant parameter list (that is, COMBUF) in register 1 and the number of bytes in the parameter list in register 0.

## SETCLK

FUNCTION: To save the current value of the timer for subsequent use.

CALLING SEQUENCE:

```
L       15, = V(SETCLK)
BALR    14,15
```

OPERATION: SETCLK stores the current value of the timer, which is located at address 80, in the storage call called TIME and returns to the calling program.

## BARE67-INITB67

Filename: INITB67

Entry point: BARE67

FUNCTION: To initialize CMS execution on a real machine (not under CP virtual machine).

CALLING SEQUENCE: BARE67 is given control by INITSUB when no chronological device, X'0FF' is present — no virtual machine.

OPERATION: BARE67 will wait for an attention interrupt from the system console. It will then request the date in the format 'SPECIFY DATE: ("mm/dd/yy"). . . . When the date has been entered, the time will be requested in the format 'SPECIFY TIME: ("hh. mm. ss"). . . . With the completion of date and time, it will proceed to determine whether the operator wishes to redefine device addresses (that is, the CMS Device Table located in NUCON). If affirmative, BARE67 will request from the user the real device addresses and set these values into DEVTAB.

The next logical step is a request to determine whether the CP-SAVE function for CMS will be executed. This is the next step whether or not there has been a redefiniter of CMS Device Addresses. If the CP-SAVE function is to be executed, it will set the real system disk address into SYSDEV located in NUCON. Then BARE67 will return control to INITSUB.

## DEFTFLV

Purpose: To provide a dummy table for file pointers.

Usage: Used by CMS FORTRAN library routines to hold file pointers. User does not have access to it. DEFINE uses it to check for too many files defined, by comparing next pointer to last.

External References: None

## CMS LOADER

The CMS Loader consists of the in-core relocating loader (LDR), the disk and type output program (LDRIO), and the library search program (LIBE). The loader reads specified files from the user's disk, searches specified libraries for missing subroutines, loads them into core storage, and establishes proper linkages.

When entered via the LOAD or $ commands, the CMS loader starts loading at location 12000 (hex) or at a user-specified location. When performing a USE or REUSE operation, loading resumes at the next available location after the previous LOAD, USE, REUSE, or LOADMOD.

The loader reads in the entire user's program, which consists of one or more control sections, each defined by a type 0 ESD card. Each control section contains a type 1 ESD card for each entry point, and may contain type 2 ESD cards for external references to other control sections, type 4 ESD cards for private code, type 5 ESD cards for common references, and type 6 ESD cards for pseudo-registers. User programs also contain:

- RLD (relocation dictionary) cards that indicate the positions of symbols within a control section; if the symbol contains a value defined in some other control section, the RLD points to that control section.
- END cards, which mark the end of each control section.
- An LDT (load terminate) card that marks the end of the entire user's program.
- REP (replace) cards with which the user makes corrections or additions to his program.
- ICS (include segment) cards that define the name of control sections.
- An SLC (set location counter) card that specifies an absolute address or a symbolic name whose assigned location is used by the loader.
- TXT cards that contain the instructions and constants of the program and the starting address at which the first byte of text is to be loaded.
- Control cards: LIBRARY and ENTRY. LIBRARY specifies entries that should not be processed by automatic library processing. ENTRY specifies "start of execution" address.

Once the user's program is in core, the loader begins to search his files for library subprograms called by the program. The loader reads the library subprograms into core, relocating and linking them as required. To relocate programs, the loader analyzes information on the SLC, ICS, ESD, TXT, and REP cards. To establish linkages, it operates on ESD, RLD, and REP cards. Information for end-of-load transfer of control is provided by the END and LDT cards, the ENTRY control card or START command.

The loader also analyzes the options specified on the LOAD, USE, and REUSE commands. In response to specified options, the loader can:

- Zero the load area before loading (CLEAR option).
- Load the program at a specified location (SLC option).
- Suppress creation of the LOAD MAP file on disk (NOMAP option).
- Print out the load map on-line (TYPE option).
- Suppress the printing of invalid card images in the load map (SINV option).
- Suppress the printing of REP card images in the load map (SREP option).
- Load program into "transient area" (TRANS option).
- Specify libraries to be searched (LIBE option).
- Suppress TXTLIB search (SLIBE option).
- Suppress TEXT file search (SAUTO option).

During its operation, the loader uses a Reference Table (REFTBL), an External Symbol Identification Table (ESIDTB), and a location counter (LOCCT). The Reference Table contains the names of control sections and entry points, their current location, and the relocation factor. (The relocation factor is the difference between the compiler-assigned address of a control section and the address of the core location where it is actually loaded.) The ESIDTB contains pointers to the entries in REFTBL for the control section currently being processed by the loader. The loader uses the location counter to determine where the control section is to be loaded. Initially, the loader obtains from the nucleus constant area the address (LOCCNT) of the next location at which to start loading. This value is subsequently incremented by the length indicated on an ESD (type 0), END, or ICS card, or it may be reset by an SLC card.

The loader contains a distinct routine for each type of input card. These routines perform calculations using information contained in the nucleus constant area, the location counter, the ESIDTB, the Reference Table, and the input cards. Other loader routines perform initialization, read cards into core, handle error conditions, provide disk and typewritten output, search libraries, convert hexadecimal characters to binary, handle end of file conditions, and begin execution of programs in core.

## OVRLD — Initial and Resume Loading Routine

FUNCTION - This routine performs initialization and is reentered during loader operation to prepare for each new loading operation.

OPERATION — At initial entry, the routine saves the address of the parameter list, obtains free storage, initializes registers, counters, flags, and save areas, obtains constants from the nucleus constant area, and sets up I/O operations. The routine also:

- Clears core before loading (optionally).
- Reads cards from disk into the SPEC buffer.

- Compares card image parameters with a parameter list (PLIST) and branches to the appropriate routine when a match is found.

- Closes files at end of file and opens next input file if more have been specified.

SLC Card Routine

FUNCTION — This routine sets the location counter (LOCCT) to the address specified on a SLC card, or to the address assigned (in the REFTBL) to a specified symbolic name.

ENTRY — The routine is entered at the first instruction when it receives control from the Initial and Resume Loading routine. It is entered at ORG2 whenever a loader routine requires the current address of a symbolic location specified on an SLC card.

OPERATION — This routine determines which of the following six situations exists, and takes the indicated action:

1. The SLC card contains neither an address nor a symbolic name. The SLC Card routine branches, via BADCRD in the Reference Table Search routine, to the Disk and Type output routine (LDRIO), which generates an error message.

2. The SLC card contains an address only. The SLC Card routine sets the location counter (LOCCT) to that address and returns to RD, in the Initial and Resume Loading routine, to read another card.

3. The SLC card contains a name only, and there is a Reference Table entry for that name. The SLC Card routine sets LOCCT to the current address of that name (at ORG2) and returns to the Initial and Resume Loading routine to get another card.

4. The SLC card contains a name only, and there is no Reference Table entry for that name. The SLC Card routine branches via ERRSLC to the Disk and Type Output routine (LDRIO), which generates an error message for that name.

5. The SLC card contains both an address and a name. If there is a REFTBL entry for the name, the sum of the current address of the name and the address specified on the SLC card is placed in LOCCT; control returns to the Initial and Resume Loading routine to get another card. If there is no REFTBL entry for the name, the SLC Card routine branches via ERRSLC to the Disk and Type Output routine, which generates an error message for the name.

## ICS Card Routine — C2AE1

FUNCTION — This routine establishes a Reference Table entry for the control-segment name on the ICS card if no entry for that name exists, adjusts the location counter to a fullword boundary, if necessary, and adds the card-specified control-segment length to the location counter if necessary.

ENTRY — This routine has one entry point, location C2AE1. The routine is entered from the Initial and Resume Loading routine when it finds an ICS card.

OPERATION —

1. The routine begins its operation with a test of card type. If the card being processed is not an ICS card, the routine branches to the ESD Card Analysis Routine; otherwise, processing continues in this routine.

2. The routine tests for a hexadecimal address on the ICS card. If an address is present, the routine links to the HEXB Subroutine to convert the address to binary; otherwise the routine branches via BADCRD to the Disk and Type Output routine (LDRIO).

3. The routine next links to the REFTBL Search Routine, which determines whether there is a Reference Table entry for the card-specified control-segment name. If such an entry is found, the REFTBL Search Routine branches to the Initial and Resume Loading Routine; otherwise, the REFTBL Search Routine places the control-segment name in the Reference Table, and processing continues.

4. The routine determines whether the card-specified control-segment length is zero or greater than zero. If the length is zero, the routine places the current location counter value in the Reference Table entry as the control segment's starting address (ORG2), and branches to the Initial and Resume Loading Routine. If the length is greater than zero, the routine sets the current location counter value at a fullword boundary address. The routine then places this adjusted current location counter value in the Reference Table entry, adjusts the location counter by adding the specified control-segment length to it, and branches to RD in the Initial and Resume Loading Routine to get another card.

## ESD Card Analysis Routine — C3AA1

FUNCTION — This routine determines whether the card being processed is an ESD card, determines which type of ESD card it is, and branches to the proper ESD card processing routine.

ENTRY — This routine has one entry point, location C3AA1.

OPERATION —

1. If dynamic loading has been indicated, a call is made to DYNALOAD for each segment to GETMAIN sufficient space and set LOCCT appropriately.

2. The routine begins its operation with a test of card type. If the card being processed is not an ESD card, the routine branches to the TXT Card Routine; otherwise, processing continues in this routine.

3. The routine tests for ESD Type 0 card. If the card is an ESD Type 0, the routine branches to the ESD Type 0 Card Routine; otherwise, processing continues in this routine.

4. The róutine then branches to location ESD00 in the ESD Type 2 Card Routine if an absolute loading process is indicated; otherwise, processing continues in this routine. The routine at location ESD00 determines whether there is more than one entry on the ESD Type 2 card. If there is another entry, the ESD00 routine returns control to location CA3A1 for further processing (repeating operation 2, above); if not, the ESD00 routine branches to location RD in the Initial and Resume Loading Routine to get another card.

EXITS — This routine exits to several routines. These are:

1. Location C4AA1, an entry point in the TXT Card Routine. This exit occurs when the card being processed is not an ESD card.

2. Location C3AA3, an entry point in the ESD Type 0 Card Routine. This exit occurs when the card being processed is an ESD Type 0 card.

3. The ESD Type 1 Card Routine. This exit occurs when the card being processed is an ESD Type 1 card.

4. Location ESD00, an entry point in the ESD Type 2 Card Routine. This exit occurs when the card being processed is not an ESD Type 0 card and the loading process is absolute.

5. Location C3AH1, an entry point in the ESD Type 2 Card Routine. This exit occurs when the card being processed is an ESD Type 2 card.

6. Location PRVESD in the ESD Type 5 and 6 Card Routine, when the card is a pseudo-register.

7. Location COMESD in the ESD Type 5 and 6 Card Routine, when the card is type common.

8. Location BADCRD in the REFTBL Search Routine, when the card is invalid.

ESD Type 0 Card Routine — C3AA3

FUNCTION — This routine makes Reference Table and ESID Table entries for the card-specified control section.

ENTRY — This routine has one entry point, location C3AA3. The routine is entered from the ESD Card Analysis Routine.

OPERATION —

1. This routine first determines whether a Reference Table (REFTBL) entry has already been established for the card-specified control section. To do this, the routine links to the REFTBL Search Routine. The ESD Type 0 Card Routine's subsequent operation depends on whether there already is a REFTBL entry for this control section. If there is such an entry, processing continues with operation 4, below; if there is not, the REFTBL Search Routine places the name of this control section in REFTBL, and processing continues with operation 2, below.

2. The routine obtains the card-specified control section length and performs operation 3, below.

3. The routine links to location C2AJ1 in the ICS Card Routine and returns to C3AD4 to obtain the current storage address of the control section from the REFTBL entry, inserts the REFTBL entry position (N - where this is the Nth REFTBL entry) in the card-specified ESID Table location, and calculates the difference between the current (relocated) address of the control section and its card-specified (assembled) address. This difference is the relocation factor; it is placed in the REFTBL entry for this control section.

4. The entry found in the REFTBL is examined to determine whether it had been defined by a COMMON. If so, it is converted from a COMMON to a CSECT and performs operation 3 above.

5. If the entry had not been defined previously by an ESD Type 0, processing continues at step 3.

6. If the entry had been defined previously other than as COMMON, LDRIO is called via ERRORM to print a warning message, "DUPLICATE IDENTIFIER". The entry in the ESID table is set negative so that the CSECT will be skipped (that is, not loaded) by the TXT and RLD processing routines.

## ESD TYPE 1 Card Routine — ENTESD

FUNCTION — This routine establishes a Reference Table entry for the entry point specified on the ESD card, unless such an entry already exists.

ENTRY — This routine is entered from the ESD Card Analysis Routine.

OPERATIONS —

1. This routine first links to the REFADR routine in order to obtain the relocation factor of the card-specified entry point's control section. If the corresponding CSECT's ESD type 0 has not been processed yet, a warning is printed and a relocation factor is assumed. This can occur if the ENTRY card precedes the START or CSECT card.

2. The routine then adds the relocation factor and the card-specified entry point address; the sum is the current storage address of the entry point.

3. The routine links to the REFTBL Search Routine to find whether there is already a REFTBL entry for the card-specified entry point name. If such an entry exists, the routine performs operation 4, below. If there is no entry, the routine performs operation 5, below.

4. Upon finding a REFTBL entry that has been previously defined for the card-specified name, the routine then compares the REFTBL-specified current storage address with the address computed in operation 2, above. If the addresses are different, the routine branches to the LDRIO routine (Duplicate Symbol Error); if the addresses are the same, the routine branches to location RD in the Initial and Resume Loading Routine to read another card. Otherwise, it is assumed that the REFTBL entry was created as a result of previously encountered external references to the entry. The ADDEF routine is called to resolve the previous external references and adjust the REFTBL entry. The entry point name and address are printed by calling LDRIO.

5. If there is no REFTBL entry for the card-specified entry point name, the routine makes such an entry and branches to the LDRIO routine.

## ESD Type 2 Card Routine — C3AH1

FUNCTION — This routine makes the proper ESID Table entry for the card-specified external name and places that name's assigned address (ORG2) in the reference table relocation factor for that name.

ENTRIES — This routine has two entry points, location C3AH1 and location ESD00. Location C3AH1 is entered from the ESD Card Analysis Routine; this occurs when an ESD Type 2 card is being processed. Location ESD00 is entered from:

1. The ESD Card Analysis Routine, when the card being processed is an ESD Type 2, and an absolute loading process is indicated.

2. The ESD Type 0 Card Routine and ESD Type 1 Card Routine, as the last step in each of these routines.

OPERATION –

1. When this routine is entered at location C3AH1, it first links to the REFTBL Search Routine to determine whether there is a REFTBL entry for the card-specified external name. If none is found, the REFTBL Search Routine branches to the LDRIO routine, which generates an error message; otherwise, the ESD Type 2 Card Routine continues processing with operation 2, below.

2. The routine next places the REFTBL entry's position-key in the ESID Table. If the entry has already been defined by means of an ESD Type 0, 1, 5, or 6, processing continues at step 4. Otherwise, continue at step 3.

3. The relocated address is placed in the RELFAC entry in the external name's REFTBL entry.

4. The ESD Type 2 Card Routine then determines (at location ESD00) whether there is another entry on the ESD card. If there is another entry, the routine branches to location CA3A1 in the ESD Card Analysis Routine for further processing of this card; otherwise, the routine branches to location RD in the Initial and Resume Loading Routine.

EXITS – This routine exits to location CA3A1 in the ESD Card Analysis Routine if there is another entry on the ESD card being processed, and exits to location RD in the Initial and Resume Loading Routine if the ESD card requires no further processing.

## ESD Type 4 Routine – PC

FUNCTION – This routine makes Reference Table and ESIDTAB entries for private code CSECT.

OPERATIONS – The ESD Type 4 Card Routine:

1. The routine LDRSYM is called to generate a unique character string number of the form 00000001, which is left in the external data area NXTSYM; it is greater in value than previously generated symbol.

2. The CSECT is then processed as a normal Type 0 ESD with the above assigned name.

## ESD Types 5 and 6 Card Routine — PRVESD and COMESD

FUNCTION — This routine makes Reference Table and ESIDTAB entries for common and pseudo-register ESD's.

OPERATION — The ESD Type 5 and 6 Card routine:

1. Links to ESIDINC in the ESD Type 0 Card Routine, to update the number of ESIDTAB entries.

2. Links to the REFTBL Search Routine to determine whether a Reference Table (REFTAB) entry has already been created. If there is no entry, the REFTBL Search Routine places the name of the item in the REFTBL.

3. If the REFTBL Search Routine had to create an entry for the item, the ESD Type 5 and 6 Card Routine indexes it in the ESIDTAB, enters the length and alignment in the entry, indicates whether it is a PR or common, and branches to ESD00 in the ESD Type 2 Card Routine to determine whether the card contains additional ESD's to be processed. If the entry is a PR, the ESD Type 5 and 6 Card Routine enters its displacement and length in the REFTBL before branching to ESD00.

4. If the REFTBL already contained an entry, the ESD Type 5 and 6 Card Routine indexes it in the ESIDTAB, checks alignment and branches to ESD00.

Note: The PR alignment is coded and placed into the REFTBL. It is an error to encounter more restrictive alignment PR than previously defined.

## TXT Card Routine — C4AA1

FUNCTION — This routine has two functions: address inspection and placing text in storage.

ENTRIES — This routine has three entry points: location C4AA1, which is entered from the ESD Card Analysis Routine, and locations REPENT and APR1, which are entered from the REP Card Routine for address inspection.

OPERATIONS —

1. This routine begins its operation with a test of card type. If the card being processed is not a TXT card, the routine branches to the REP Card Routine; otherwise, processing continues in this routine.

2. The routine then determines how many bytes of text are to be placed in storage, and finds whether the loading process is absolute or relocating. If the loading process is absolute, the routine performs operation 4, below; if relocating, the routine performs operation 3.

3. If the ESIDTAB entry was negative, this is a duplicate CSECT and processing branches to RD. Otherwise, the routine links to the REFADR Routine to obtain the relocation factor of the current control segment.

4. The routine then adds the relocation factor (0, if the loading process is absolute) and the card-specified storage address. The result is the address at which the text must be stored. This routine also determines whether the address is such that the text, when loaded starting at that address, will overlay the loader or the Reference Table. If a loader overlay or a Reference Table overlay is found, the routine branches to the LDRIO Routine. If neither condition is detected, the routine proceeds with address inspection.

5. The routine then determines whether an address has already been saved for possible use as the end-of-load branch address. If an address has been saved, the routine performs operation 7; if not, the routine performs operation 6.

6. The routine determines whether the text address is below location 128. If the address is below location 128, it should not be saved for use as a possible end-of-load branch address, and the routine performs operation 7; otherwise the routine saves the address and then performs operation 7.

7. The routine then stores the text at the address specified (absolute or relocated) and branches to location RD in the Initial and Resume Loading Routine to read another card.

EXITS — The routine exits to two locations, as follows:

1. The routine exits to location RD in the Initial and Resume Loading Routine if it is being used to process a TXT card.

2. The routine exits to location APRIL in the REP Card Routine if it is being used for REP card address inspection.

## REP Card Routine — C4AA3

FUNCTION — This routine places text corrections in storage.

ENTRY — This routine has one entry point, location C4AA3. The routine is entered from the TXT Card Routine.

OPERATION —

1. This routine begins its operation with a test of card type. If the card being processed is not a REP card, the routine branches to the RLD Card Routine; otherwise, processing continues in this routine.

2. The routine then links to the HEXB Conversion Routine to convert the REP card-specified correction address from hexadecimal to binary.

3. The routine then links to the HEXB Conversion Routine again to convert the REP card-specified ESID from hexadecimal to binary.

4. The routine then determines whether the two-byte correction being processed is the first such correction on the REP card. If it is the first correction, the routine performs operation 5, below; otherwise, the routine performs operation 6, below.

5. When the routine is processing the first correction, it links to location REPENT in the TXT Card Routine, where the REP card-specified correction address is inspected for loader overlay and for end-of-load branch-address saving; in addition, if the loading process is relocating, the relocated address is calculated and checked for Reference Table overlay. The routine then performs operation 7, below.

6. When the correction being processed is not the first such correction on the REP card, the routine branches to location APR1 in the TXT Card Routine for address inspection.

7. The routine then links to the HEXB Conversion Routine to convert the correction from hexadecimal to binary, places the correction in storage at the absolute (card-specified) or relocated address, and determines whether there is another correction entry on the REP card. If there is another entry, the routine repeats its processing from operation 4, above; otherwise, the routine branches to location RD in the Initial and Resume Loading Routine.

EXITS — This routine exits, when all the REP-card corrections have been processed, to location RD in the Initial and Resume Loading routine.

## RLD Card Routine — C5AA1

FUNCTION — This routine processes RLD cards, which are produced by the assembler when it encounters address constants within the program being assembled. This routine places the current storage address (absolute or relocated) of a given defined symbol or expression into the storage location indicated by the assembler. The routine must calculate the proper value of the defined symbol or expression and the proper address at which to store that value.

ENTRY — This routine has one entry point, location C5AA1. The routine is entered from the REP Card Routine.

OPERATION —

1. This routine begins its operation with a test of card type. If the card being processed is not an RLD card, the routine branches to the END Card Routine; otherwise, processing continues in this routine.

2. The routine next determines whether the loading process is absolute or relocating. If absolute, the routine branches to location RD in the Initial and Resume Loading routine; if relocating, the routine continues processing the RLD card.

3. The routine uses the Relocation Header (RH ESID) on the card to obtain the current address (absolute or relocated) of the symbol referred to by the RLD card. This address is found in the Relocation Factor section of the proper Reference Table entry. If the RH ESID is 0, the routine branches to the LDRIO Routine (Invalid ESD).

4. The routine uses the Position Header (PH ESID) on the card to obtain the relocation factor of the control segment in which the Define Constant assembler instruction occurred. If the PH ESID is 0, the routine branches to BADCRD in the REFTBL Search Routine (Invalid ESID). If the ESIDTAB entry is negative (Duplicate CSECT), the RLD entry is skipped.

5. The routine next decrements the card-specified byte count by 4 and tests it for 0. If the count is now 0, the routine branches to location RD in the Initial and Resume Loading Routine; otherwise, processing continues in this routine.

6. The routine determines the length, in bytes, of the address constant referred to in the RLD card. This length is specified on the RLD card.

7. The routine sets up an instruction (which will place the specified address value in storage at the specified address) by inserting the proper number in the Move instruction. This number is the address constant's byte-length.

8. The routine then adds the relocation factor obtained in operation 4, above (relocation factor of the control segment in which the current address of the symbol must be stored), and the card-specified address. The sum is the current address of the location at which the symbol address must be stored.

9. The routine then computes the arithmetic value (symbol address or expression value) that must be placed in storage at the address calculated in operation 8, above, and places that value at the indicated address. If the value is undefined, the routine branches to location APOINT, where the constant is added to a string of constants that are to be defined later.

10. The routine again decrements the byte count (of information on the RLD card) and tests the result for zero. If the result is zero, the routine branches to location RD in the Initial and Resume Loading Routine; otherwise, processing continues in this routine.

11. The routine next checks the continuation flag, a part of the data placed on the RLD card by the assembler. If the flag is on, the routine repeats its processing for a new address only; the processing is repeated from operation 5, above. If the flag is off, the routine repeats its processing for a new symbol; the processing is repeated from operation 3, above.

EXITS — This routine exits to location RD in the Initial and Resume Loading Routine.

## END Card Routine — C6AA1

FUNCTION — This routine saves the END card address under certain circumstances, and initializes the loader to load another control segment.

ENTRY — This routine has one entry point, location C6AA1. The routine is entered from the RLD Card Routine.

OPERATION —

1. This routine begins its operation with a test of card type. If the card being processed is not an END card, the routine branches to the LDT Card Routine; otherwise, processing continues in this routine.

2. The routine then determines whether the END card contains an address. If the card contains no address, the routine performs operation 7, below; otherwise, the routine performs operation 3.

3. The routine next checks the end-address-saved switch. If this switch is on, an address has already been saved, and the routine performs operation 7. If the switch is off, the routine performs operation 4.

4. The routine determines whether the loading process is absolute or relocating. If the loading process is absolute, the routine performs operation 6; otherwise, the routine performs operation 5.

5. The routine links to the REFADR routine to obtain the current relocation factor, and adds this factor to the card-specified address.

6. The routine stores the address (absolute or relocated) in area BRAD, for possible use at the end-of-load transfer of control to the problem program.

7. The routine then clears the ESID Table, sets the absolute load flag on, and branches to the location specified in a general register (see Exits).

EXITS — This routine exits to the location specified in a general register. This may be either of two locations, as follows:

1. Location RD in the Initial and Resume Loading Routine. This exit occurs when the END Card Routine is processing an END card.

2. The location in the LDT Card Routine that is specified by that routine's linkage to the END Card Routine. This exit occurs when the LDT Card Routine entered this routine to clear the ESID Table and set the absolute load flag on.

## LDT Card Routine — C6AC1

FUNCTION — This routine determines the address at which the loaded program is to begin execution and transfers control to that address, terminating the loading process.

ENTRY — The routine is entered at C6AC1 from the END Card Routine.

OPERATIONS — The LDT Card Routine:

1. Tests the card type. If the card is not an LDT card, the routine branches to the Control Card Routine; otherwise, processing continues in this routine.

2. Determines whether the LDT card contains a name. If there is a name, the routine performs operation 3; otherwise, the routine proceeds to operation 4.

3. Links to the Reference Table Search Routine to determine whether there is an entry in REFTBL for the name specified in the LDT card. If there is no entry for the name, the Reference Table Search Routine branches to the Disk and Type Output routine, which generates an error message. If an entry is found, the address of the entry is returned to the LDT Card routine, which stores it in the end of load branch address save area (BRAD).

4. Determines (at LIBGO) whether the library search has been suppressed. If the search has not been suppressed and there are missing subroutines to be read in, control passes to the LIBE program, which searches the specified libraries. If a needed entry is found in the library, processing continues at location NXTRD. Otherwise, processing continues at step 6.

5. If the ENTRY control card had been encountered, the start address is set to the specified entry-address (if it was found). Otherwise, the previously determined start address is used.

6. If any undefined symbols were referenced without specification as nonobligatory on a LIBRARY card, the names are printed as a warning to the user.

7. Tests (at N03) for the XEQ option. If it was specified, control is passed to the Begin Execution Routine (XEQQ). If it was not specified, the libraries are closed (at N03), free storage is released, and control is returned to the loaded program.

## Control Card Routine — CTLCRD1

FUNCTION — This routine handles the ENTRY and LIBRARY control cards.

ENTRY — This routine has one entry point, location CTLCRD1. The routine is entered from the LDT Card Routine.

OPERATIONS —

1. The CMS function SCAN is called to parse the card.

2. If the card is not an ENTRY or LIBRARY card, the routine determines whether the SINV option (suppress printing of invalid card images) was specified. If printing is suppressed, control passes to RD in the Initial and Resume Loading Routine, where another card is read. If printing is not suppressed, control passes to the Disk and Type Output Routine (LDRIO), where the invalid card image is printed in the load map. If the card is a valid control card, processing continues.

ENTRY Card:

3. If the ENTRY name is already defined in REFTBL, its REFTBL address is placed in ENTADR. Otherwise, a new entry is made in REFTBL, indicating an undefined external reference (to be resolved by later input or library search), and this REFTBL entry's address is placed in ENTADR.

4. The control card is printed by calling LDRIO via CTLCRD; it then exits to RD.

LIBRARY Card:

5. Only nonobligatory reference LIBRARY cards are handled; any other form is considered an invalid card.

6. Each entry-point name is individually isolated and is searched for in the REFTBL. If it has already been loaded and defined, nothing is done and the next entry-point name is processed. Otherwise, the nonobligatory bit is set in the flag byte of the REFTBL entry.

7. Processing continues at step 4.

## REFADR Routine

FUNCTION — This routine computes the storage address of a given entry in the Reference Table.

ENTRY — This routine has one entry point, location REFADR. The routine is entered from several of the routines within the loader.

OPERATION —

1. The routine first obtains, from the indicated ESID Table entry, the position (n) of the given entry within the Reference Table (where the given entry is the nth REFTBL entry).

2. The routine then multiplies n by 16 (the number of bytes in each REFTBL entry) and subtracts this result from the starting address of the Reference Table. The starting address of the Reference Table is held in area TBLREF; this address is the highest address in storage, and the Reference Table is always built downward from that address.

3. The result of the subtraction in operation 2, above, is the storage address of the given Reference Table entry. If there is no ESD for the entry, the routine prints a warning by calling LDRIO via INVESD and assumes "next available location" as CSECT address; otherwise, this routine returns to the location specified by the calling routine.

## PRSERCH Routine

FUNCTION — This routine compares each Reference Table entry name with the given name, determining first, whether there is an entry for that name and second, what the storage address of that entry is.

ENTRY — This routine is initially entered at PRSERCH, and subsequently at location SERCH. The routine is entered from several routines within the loader.

OPERATION —

1. This routine begins its operation by obtaining the number of entries currently in the Reference Table (this number is contained in area TBLCT), the size of a Reference Table entry (16 bytes), and the starting address of the Reference Table (always the highest address in storage, contained in area TBLREF).

2. The routine then checks the number of entries in the Reference Table. If the number is 0, the routine performs operation 5, below; otherwise, the routine performs operation 3.

3. The routine next determines the address of the first (or next) Reference Table entry to have its name checked, increments by one the count it is keeping of name comparisons, and compares the given name with the name contained in that entry. If the names are identical, PRSERCH branches to the location specified in the routine that linked to it. PRSERCH then returns the address of the REFTBL entry; otherwise, PRSERCH performs operation 4.

4. The routine then determines whether there is another Reference Table entry to be checked. If there is none, the routine performs operation 5; if there is another, the routine decrements by one the number of entries remaining and repeats its operation starting with operation 3.

5. If all the entries have been checked, and none contains the given name for which this routine is searching, the routine increments by one the count it is keeping of name comparisons, places that new value in area TBLCT, moves the given name to form a new Reference Table entry, and returns to the calling program.

EXITS — This routine exits to either of two locations, both of which are specified by the routine that linked to this routine. The first location is that specified, in the event that an entry for the given name is found; the second location is that specified, in the event that such an entry is not found.

## HEXB Conversion Subroutine

FUNCTION — This subroutine converts a specified number of characters of hexadecimal data to binary form.

ENTRIES — This routine has one entry point, location HEXB. The routine may be entered from:

1. The REP Card Routine, for conversion of an address or of a text correction, or

2. Any of several routines within the loader, for conversion of hexadecimal data.

OPERATION —

1. In a series of tests beginning at location L1, the routine determines whether the first hexadecimal character to be converted is a valid numeric, a valid alphabetic, or an invalid character.

2. If operation 1 indicates that the character is a valid numeric character (0-9), the routine converts the hexadecimal character to binary by clearing its high-order (leftmost) four bits. The routine then performs operation 5, below.

3. If operation 1 indicates that the character is a valid alphabetic character (A-F), the routine converts the hexadecimal character to binary by subtracting a constant from the character. The routine then performs operation 5, below.

4. If operation 1 indicates that the character is invalid, the routine branches to location BADCRD in the REFTBL Search Routine.

5. When the routine has converted a valid numeric or valid alphabetic character to binary form, it shifts the general register in which it returns the entire converted number four bits to the left and inserts the converted digit in the vacant low-order (rightmost) four bits of that register.

6. The routine then determines whether it has converted all the characters that were passed to it for conversion. If it has not, it branches within itself to location L1 and repeats the conversion process for the next character; otherwise, it returns to the calling routine.

EXIT — This routine has two exits:

1. If the routine encounters an invalid hexadecimal character, it exits to BADCRD, in the REFTBL Search Routine.

2. If the routine encounters no invalid hexadecimal character during the process of converting the entire specified number, it exits to the address contained in location RETT. This address is the return address specified by the loader routine or problem program that linked to this routine.

## Start Execution Routine — XEQQ

FUNCTION — This routine begins execution of programs loaded into core. It is called by the START command or the LOAD, USE, or REUSE commands, if the XEQ option was specified with them.

OPERATIONS — The Start Execution Routine receives control from the START command or the LDT Card Routine if the XEQ option was specified. The Start Execution routine:

1. Initializes free storage.

2. Calls the Disk and Type Output program (LDRIO) to print a storage map header for COMMON, if the REFTBL contains any common entries.

3. Searches the REFTBL for entries with a nonzero flag (COMMON, CXD, PR, and references to undefined symbols).
   (a) A pointer to a PR entry is placed in a separate table depending upon PR alignment: byte, halfword, fullword, or doubleword.

(b) A CXD entry is marked "undefined" for later processing and a pointer placed in special CXD table.

(c) A COMMON is defined at the next available load location and the location counter (LOCCT) is increased by the length of the COMMON. References to the COMMON are processed by calling ADDEF. The name, address, and length of the COMMON are printed by calling LDRIO via CMVAL.

(d) A reference to an undefined symbol is replaced by a value of zero by calling ADDEF.

4. After scanning entire REFTBL, the Disk and Type Output program (LDRIO) is called to print a storage map header for PR's, if there are any pseudo-register entries in the REFTBL. The name, value, and length of each PR (they were defined earlier when ESD was first encountered) are printed as well as the CXD request, which can only be determined after all PR's have been assigned.

5. The starting address for execution is then determined: by entry-point name if supplied, by START command or by the start execution address (in BRAD) set by first program loaded, END card, LDT card, or ENTRY control card.

6. (a) If actual execution is suppressed, for example, by specifying START (NO) command, the library is closed, load map closed, free storage released, and control returned to caller. Otherwise, execution is performed by step 6(b).

   (b) The message "EXECUTION BEGINS" is printed, GETMAIN/FREEMAIN storage is initialized, save area set in register 13, and control transferred to loaded program via LPSW.

## Disk and Type Output Routine – LDRIO

FUNCTION – The LDRIO routine creates the load map on disk and types it out at the terminal; it performs disk and typewriter output for the LDR.

OPERATION – The LDRIO routine is entered by several CMS routines. At each entry, it writes data for the map on disk (unless the NOMAP option has been specified by the user). If the user has specified the TYPE option, LDRIO types out the map at the user's terminal.

At the end of loading, LDRIO closes the map file and returns control to the user.

## Library Search Routine — LIBEPACK

FUNCTION — The LIBEPACK routine has several entries to set up core-resident library directories, release these directories, and search the directories.

OPERATION —

Initialization Entry:

1. A special storage area is allocated and initialized.

2. The number of libraries specified either by GLOBAL TXTLIB command or LIBE option is determined and the names processed.

3. For each library, sufficient storage is allocated to hold its directory and then its directory is read into main storage.

Release Entry:

For each library, steps 1 and 2 are performed.

1. The TXTLIB file is closed via FINIS.

2. The main storage area that holds a copy of its directory is released via FRET.

3. The LIBEPACK special storage area is released via FRET.

Search Entry:

1. The reference table (REFTBL) is searched for occurrences of undefined references; if none are found, it returns to caller.

2. If undefined reference is found, each library's core-resident directory is searched for the required entry-point. If entry cannot be found, control passes to step 1 continuing the search of REFTBL, otherwise, to step 3.

3. The file read pointer and parameter list are set to point to the library and card that correspond to the beginning of the CSECT deck that contains the desired entry-point. Control is returned to caller via special return that continues the loading process from specified file.

## Relevant Loader Data Bases

NAME — SWS Flag Byte

| Bit Number | Meaning |
|---|---|
| 0 (80) | Absolute load |
| 1 (40) | TXT cards processed |
| 2 (20) | SLC card being processed |
| 3 (10) | END card processed |
| 4 (08) | REP card being processed |
| 5 (04) | not processing first ESD field on ESD card |
| 6 (02) | do not print illegal cards (SINV) |
| 7 (01) | do not print REP cards (SREP) |

NAME — CONS Flag Byte

| Bit Number | Meaning |
|---|---|
| 1 (40) | no map option specified (NOMAP) |
| 2 (20) | PR entries exist |
| 3 (10) | COMMON entries exist |
| 4 (08) | online load map (TYPE) |
| 5 (04) | do not erase old load map (USE, REUSE, START) |

NAME — FLAGS Flag Byte

| Bit Number | Meaning |
|---|---|
| 4 (08) | LIBEPACK has been initialized |
| 5 (04) | Close LIBE call in process |
| 6 (02) | Undefined references exist in REFTBL |
| 7 (01) | LIBE search suppressed. |

The next 3 bytes are a pointer to LIBEPACK storage area if it has been initialized (bit 4).

NAME — ESD Card Codes (col. 25. . .)

| Code | Meaning |
|---|---|
| 00 | SD (CSECT or START) |
| 01 | LD (ENTRY) |
| 02 | ER (EXTRN) |
| 04 | PC (Private Code) |
| 05 | CM (COMMON) |
| 06 | XD (Pseudo-Register) |

## ESIDTB Entry

The ESD ID Table (ESIDTB) is constructed separately for each TEXT deck processed by the loader. The ESIDTB produces a correspondence between ESD ID numbers (used on RLD cards) and entries in the loader reference table (REFTBL) as specified by the ESD cards. Thus, the ESIDTB is constructed while processing the ESD cards. It is then used in order to process the TXT and RLD cards later in the TEXT deck.

The ESIDTB is treated as an array and is accessed by using the ID number as an index. Each ESIDTB entry is 16 bits long:

| Bits | Meaning |
|---|---|
| 0 | If 1, this entry corresponds to a CSECT that has been previously defined. All TXT cards and RLD cards referring to this CSECT in this TEXT deck should be ignored. |
| 1 | If 1, this entry corresponds to a CSECT definition (SD). |
| 2-3 | Unused. |
| 4-15 | REFTBL entry number (e.g. 1,2,3, ...) |

Bit 1 is very crucial since it is necessary to use the VALUE field of the REFTBL if the ID corresponds to an ER, CM, or PR; but, the INFO field of the REFTBL entry must be used if the ID corresponds to an SD.

## NAME — REFTBL Entry

| Byte | Name | Meaning |
|---|---|---|
| 0-7 | NAME | Symbolic Name |
| 8 | FLAG | Flag Byte |
| 9-11 | INFO | Necessary Information |
| 12 | -- | Unused (must be zero) |
| 13-15 | VALUE | Value of Symbol |

NAME – REFTBL Entry Flag Byte Codes

| Code | Translation | Action Routine | Meaning |
|---|---|---|---|
| 7C | 00 | XBYTE | PR byte align |
| 7D | 01 | XHALF | PR halfword align |
| 7E | 03 | XFULL | PR fullword align |
| 7F | 07 | XDBL | PR doubleword align |
| 80 | 05 | XUNDEF | Undefine symbol |
| 81 | 04 | XCXD | CXD |
| 82 | 02 | XCOMSET | COMMON |
| 90 | 06 | N1BLK | Nonobligatory undefined symbol |

NAME — REFTBL Info and Value Fields

| Symbol Type | INFO Field | VALUE Field |
|---|---|---|
| SD (CSECT or START) | Relocation-factor | Absolute Address |
| LD (ENTRY) | zero | Absolute Address |
| CM (COMMON) | Length (max) | Absolute Address |
| PR (Pseduo-Register) | Length (e.g. 4) | Assigned Value (starting from 0) |
| PC (Private Code) | Relocation-factor | Absolute Address |

Entries may be created in the loader reference table prior to the actual defining of the symbol. For example, an entry is created for a symbol if it is referenced by means of an EXTRN (ER) even if the symbol has not yet been defined nor even its type known. Furthermore, Common (CM) is not assigned absolute addresses until immediately prior to the start of execution by the START command.

These circumstances are determined by the setting of the Flag Byte; if the symbol's value has not yet been defined, the VALUE field specifies the address of a Patch Control Block (PCB).

NAME — Patch Control Block (PCB)

These are allocated from free storage and pointed at from REFTBL entries or other PCB's.

| Byte | Meaning |
|---|---|
| 0 - 3 | Value of constant [for example, A(EXTRN+4) results in constant = 4 ] |
| 4 - 7 | Pointer to first such address constant. |
| 8 - 11 | Pointer to last such address constant. |
| 12 | Flag byte |
| 13 - 15 | Pointer to next PCB for same external symbol but different constant, or zero. |

All address constant locations in loaded program for undefined symbols are placed on PCB chains.

NAME — LIBEPACK Storage Area

| Byte | Meaning |
|---|---|
| 0 - 3 | Number of libraries (present max = 8) |
| 4 - 7 | TBLCNT |
| 8 - 11 | Updated TBLCNT during search |
| 12 - 15 | TBLREF, address of REFTBL |
| 16 - 23 | Library name |
| 24 - 27 | Length of dictionary in doublewords |
| 28 - 31 | First location of dictionary in core |
| 32 - 35 | "12" [constant] |
| 36 - 39 | Last location of dictionary in core |
| 40 - 111 | Field 16-39 repeated 3 times for other 3 libraries |
| 112-175 | Register save area |

NAME — ESD Card Format

| Column | | Meaning |
|---|---|---|
| 1 | | 12-2-9 punch |
| 2-4 | | ESD |
| 5-10 | | Blank |
| 11-12 | | Variable field count |
| 13-14 | | Blank |
| 15-16 | | ESDID for first SD, XD, CM, PC, or ER |
| 17-64 | | Variable field, repeated 1 to 3 times |
| | (17-24) | Name |
| | (25) | ESD type code |
| | (26-28) | Address |
| | (29) | Alignment for XD, otherwise blank |
| | (30-32) | Length, LDID, or blank |

NAME — TXT Card Format

| Column | Meaning |
|---|---|
| 1 | 12-2-9 punch |
| 2-4 | TXT |
| 5 | Blank |
| 6-8 | Relative address of first data on card |
| 9-10 | Blank |
| 11-12 | Byte count |
| 13-14 | Blank |
| 15-16 | ESDID |
| 17-72 | 56-byte field |

NAME — RLD Card Format

| Column | | Meaning |
|---|---|---|
| 1 | | 12-2-9 punch |
| 2-4 | | RLD |
| 5-10 | | Blank |
| 11-12 | | Data field count |
| 13-16 | | Blank |
| 17-72 | | Data field |
| | (17-18) | Position ESDID |
| | (19-20) | Relocation ESDID |
| | (21) | Flag byte |
| | (22-24) | Address to be relocated |

## SECTION 6: CMS BATCH MONITOR

The CMS Batch Monitor is a method of providing a high-speed background batch job environment in a CMS machine. A job stream, consisting of batch control cards (BCC), source decks to be compiled and/or object decks to be loaded and executed, and data, is placed in the input device - the tape at address 185, or the card reader at 00C. Assemble, Fortran, PL/I and any legitimate CMS command may be executed in the batch environment.

The batch nucleus is constructed by replacing the DEBUG text deck in the normal CMS nucleus with text decks of:

| | | |
|---|---|---|
| BATCH | - | main control routine |
| BATBOMB | - | ABEND routine |
| BATDECC | - | decode standard OS JCL |
| BATJCB | - | CSECT of standard system values |
| BATLIST | - | data set management |
| BATPRES | - | data set management |
| BATSCTL | - | main I/O interface management |
| IPL | - | initial-program-load command |

A batch machine should be set up in CP's virtual machine directory to consist of:

normal unit record equipment,
core size desired,
terminal,
normal CMS system disk 190,
normal CMS user disk 191,
small work disk for batch nucleus only.

The batch virtual machine is logged into CP as any other virtual CMS machine. The difference is solely in the construction of the nucleus and the subsequent activity.

The batch nucleus must be available on a disk other than the normal system disk. Once the batch nucleus is IPL'ed, disk-resident commands are read from the normal CMS System Disk, 190.

After IPL'ing the batch nucleus, BATCH responds with READY. The START command must then be given from the console. BATCH transfers to BATSCTL to determine the input device - tape 185, if ready; if not ready, it defaults to the card reader. If there is a job stream in the input device it will be processed until the input device indicates an EOF, at which time the message "END OF JOB STREAM" is typed.

During processing, if a program interrupt occurs, BATBOMB receives control and initiates a core dump to the output device.

Between each // JOB card, the batch nucleus re-IPL's itself, and issues a LOGIN NO_UFD command to its P-disk. Notice, no data files will be retained on the P-disk during job transition.

Since all activity is initiated by control cards, no terminal input is necessary, and the normal CMS conversational routines are not entered.

At the end of the job stream, Batch will enter an enabled wait state. It waits for an interrupt from its input device. A device-end interrupt on the virtual card reader may be caused by XFER'ing a card deck to batch virtual machine. In other words, jobs may be sent from any user's virtual machine to Batch.

It is also possible to run Batch in the CP disconnect mode. To supplement the above discussion, refer to the CP-67/CMS Installation Guide (GH20-0857) and CP-67/CMS User's Guide (GH20-0859) for an explanation of DISCONNECT and XFER.

The routine BATCH is entered from INITSUB, if the value V(BATCH) is nonzero. This is a method used throughout the CMS nucleus to determine whether BATCH is operating. Another method is to test the $BAT-bit in the word EXECSWT in SWITCH. BATCH sets this bit on (X'01') to signal that the Batch Monitor nucleus is operating.

On the initial start-up, a READY message is typed. Upon receiving a START command from the console, the routine BATSCTL is entered to determine the input device. BATSCTL does all I/O processing peculiar to Batch; card I/O, printer output, etc. CMS will still manipulate the basic disk file I/O.

From the 00C the function card is scanned, and the appropriate routine given control - ASSEMBLY, FORTRAN, LOAD, GO, DATA, PRINT/PUNCH, etc. For a COMMAND card, the operand becomes the subject of an SVC to CMS; that is the operand is assumed to be any valid CMS command. Within the CMS normal I/O packages, tests are made to determine whether Batch is active. If so, for any I/O, control is passed to BATSCTL.

The routine BATBOMB will output a core dump if a program check occurs; BATLIST will print or punch the CMS file, which is the operand of a PRINT or PUNCH control card. BATJCB contains program dependent constants; for example, a time limit per job may be set. BATPRES will write a CMS file onto the Batch P-disk from data sets in the input device.

## APPENDIX A: CMS FILE NAMING CONVENTIONS

To make the filenames of CMS source routines more meaningful to the system programmer responsible for maintaining CMS, a naming convention has been established. A routine's function and relationship to other routines will consequently be identified by prefixes and suffixes. The naming conventions are:

- device dependent routines prefixed by a code denoting the physical device type: e.g., DISK, CONsole, etc.
- device interrupt handlers suffixed by "INT": CONINT
- software interrupts prefixed by "INT": INTSVC
- simulators of OS functions prefixed by "SO": SOQSAM
- miscellaneous routines of similar functions prefixed by the same code: FREESYS, FREEXTND
- device input/output executors suffixed by "IO": TAPEIO

The following chart will show:

1. the filename of the source SYSIN deck
2. the internal entry point(s) or START card label if dissimilar from the filename
3. how the routine is used:

   N - nucleus resident
   NI - nucleus resident during the IPL procedure only - then no longer needed
   NS - nucleus resident, sharable code
   D - disk resident module
   DC - component of a disk resident module
   T - transient module

4. relationship between Version 3 Level 0 and Version 3 Level 1 with respect to the individual routine:

   S - same, routine is unchanged
   U - updated, the update deck that was used to generate the current version of the routine will be supplied.
   UN - updated and renamed. The newly named update deck will be supplied. Note: the Version 3 Level 0 source and the Version 3 Level 1 source - prior to applying the update - are identical
   R - replacement, complete substitute for the mentioned Version 3 source routines

RN - rename, the filename of the Version 3 Level 0 source deck was changed.
N - new

5. a brief comment about each function

| External Filename | Internal Entry Point | Usage | Relation vis a vis Ver 3.0 | Functions |
|---|---|---|---|---|
| $ | — | T | U | execution initiator |
| ABBREV | — | N | S | abbreviation processor |
| ACTLKP | — | NS | U | determine active files |
| ADTLKP | — | NS | U | determine available disks |
| ALTER | — | T | U | change file identification |
| ASMDIRT | — | DC | R-ASMDIRT | ASSEMBLER auxiliary directory |
| ASSEMBLE | — | D | R-ASSEMBLE,<br>ASMREAL<br>ASMA01,<br>ASMA02<br>ASMA03,<br>ASMAFIND | ASSEMBLER interface |
| BATBOMB | — | N | S | batch monitor component |
| BATCH | — | N | U | batch monitor component |
| BATDECC | — | N | S | batch monitor component |
| BATJCB | — | N | S | batch monitor component |
| BATLIST | — | N | S | batch monitor component |
| BATPRES | — | N | S | batch monitor component |
| BATSCTL | — | N | S | batch monitor component |
| CARDIO | CARDRDPH | N | S | reader/punch executor |
| CEDIT | — | D | U | editor for large files |
| CLOSIO | — | N | S | close executor for Unit Record Equipment |
| CMSCARE | — | T | N | auxiliary command module |
| CMSCONF | CPFUNCTN | T | U | virtual CP console function executor |
| CMSFORM | FORMAT | D | U | disk formattor |
| CMSIPL | IPL | T | S | CMS IPL invoker |
| CMSTIME | — | NS | R-CMSTIME | virtual time accounter |
| CNVTFV | CVTFV | D | RN-CVTFV | convert fixed/variable |
| CNVT26 | — | D | S | convert 026/029 |
| COMBINE | — | D | R-COMBINE | file manipulator |
| COMPARE | — | D | U | file matchmaker |
| CONATTN | ATTN | NS | UN-ATTN | attention handler |
| CONINT | CONSI | NS | UN-CONINT | console interrupts |
| CONREAD | WAITRD | NS | UN-WAITRD | console input |
| CONWAIT | — | NS | U | console wait |
| CONWRITE | TYPE | NS | UN-TYPLIN | console write |

|  | Internal |  | Relation |  |
|---|---|---|---|---|
| External | Entry |  | vis a vis |  |
| Filename | Point | Usage | Ver 3.0 | Functions |
| DEBDUMP | — | N | N | debug dump executor |
| DEBUG | — | N | U | problem determination aide |
| DISK | — | D | U | disk utility |
| DISKINT | — | N | UN-DIOSECT | disk interrupts |
| DISKIO | RDTK/WRTK | NS | UN-DIO | disk I/O executor |
| DUMPRST | DUMPREST | D | S | dump/restore utility |
| DUMPD | — | D | S | dump disk utility |
| DUMPF | — | D | S | dump file utility |
| ECHO | — | D | S | terminal tester |
| EDIT | — | D | U | editor |
| EDITDUAL | — | NS | U | file eradicator |
| ERASE | — | NS | U | file eradicator |
| EXEC | — | NS | U | exec bootstrap |
| EXECTOR | — | D | U | exec work module |
| FILEDEF | — | T | R-FILEDEF | define file routine |
| FINIS | — | NS | U | file deactivator |
| FORDIRT | — | DC | R-FORTDIRT | FORTRAN auxiliary directory |
| FORTRAN | — | D | R-FORTRAN,<br>FORTIO, | FORTRAN interface |
| FREESYS | FREE/FRET | NS | UN-CMSFREE | system free storage |
| FREEXTN | EXTEND | N | UN-CMSEXTND | system free storage |
| FSTLKP | — | NS | U | file lookup |
| FUNCTAB | — | NS | R-FUNCTAB | internal function table |
| GENDIRT | — | T | S | auxiliary direct generator |
| GENMOD | LOADMOD/<br>GENMOD | NS | U | module manipulator |
| GLOBAL | — | T | R-GLOBAL | library governess |
| HNDINT | — | T | S | handle I/O interrupts |
| HNDSVC | — | T | S | handle SVC interrupts |
| IADT | — | NS | U | init Active Disk Tables |
| INIT | — | N | U | initializer |
| INITIPL | TRANSAR,<br>IPLDISK | NI | R-LAST,<br>IPLDISK | reads CMS from disk |
| INITB67 | BARE67 | NI | R-BARE67 | bare CPU initial processor |
| INITSUB | — | NI | U | sub-initializer |
| INITSYS | SYSGEN | NI | RN-SYSGEN | initializes S-disk |
| INTEXT | EXTINT | NS | UN-CMSEXTIT | external interrupt |
| INTIO | IOINT | NS | UN-CMSIOIT | I/O interrupt |
| INTMACH | MCHINT | NS | N | machine interrupt |
| INTPROG | PRGINT | NS | R-CMSPRGIT | program interrupt |
| INTSECT | — | N | N | interrupts work area |
| INTSVC | SVCINT | NS | R-CMSSVCIT | SVC interrupt |
| IOFRTAB | — | N | N | disk tables |
| IOERR | — | N | N | I/O error processor |
| IONUTAB | — | D | N | I/O error tables |

| External Filename | Internal Entry Point | Usage | Relation vis a vis Ver 3.0 | Functions |
|---|---|---|---|---|
| IXCBLTP | — | DC | S | FORTRAN library |
| IXCCMS | — | DC | U | FORTRAN library |
| IXCDEF | — | DC | S | FORTRAN library |
| IXCDEFTB | — | DC | S | FORTRAN library |
| IXCDSD | — | DC | U | FORTRAN library |
| IXCFREM | — | DC | S | FORTRAN library |
| IXCGETP | — | DC | S | FORTRAN library |
| IXCRENM | — | DC | S | FORTRAN library |
| IXCRERD | — | DC | S | FORTRAN library |
| IXCTAP | — | DC | S | FORTRAN library |
| IXECMS | — | DC | R-IXECMS | PL/I library |
| IXECLOK | — | DC | R-IXECLOK | PL/I library |
| IXEFILE | — | DC | R-IXEFILE | PL/I library |
| LDR | — | NS | U | relocatable loader |
| LDRIO | — | N | R-LDRIO | loader I/O executor |
| LDRLIBE | — | N | R-LDRLIBE | loader library processor |
| LDRSUBS | — | N | R-LDRSUBS | loader subroutine |
| LDRSYM | GETSYM | N | RN-GETSYM | loader symbols |
| LISTF | — | T | S | list files |
| LOAD | — | N | U | load initiator |
| LOG | — | N | U | maintain disk direct |
| LOGIN | — | T | U | login a disk |
| MACLIB | — | D | U | macro library |
| MAPPRT | — | D | U | nucleus map |
| MODMAP | — | T | U | module map |
| NUCON | — | N | R-NUCONTS | nucleus constants |
| NUDVEXT | — | N | N | I/O device tables |
| NUSECT | — | N | N | nucleus work section |
| OFFLINE | — | T | R-OFFLINE | file utility |
| OSTAPE | — | D | S | tape-file utility |
| OVERRIDE | — | D | U | override executor |
| OVERSUB | — | DC | UN-JASOVER | override executor |
| PLI | — | D | R-PL/I | PL/I interface |
| PLIDIRT | — | DC | U | PL/I auxiliary directory |
| POINT | — | NS | S | sys file manipulator |
| PRINTF | — | T | S | print files |
| PRINTIO | PRINTR | NS | UN-PRINTR | printer executor |
| QQTRK | — | NS | S | track manager |
| RDBUF | — | NS | U | basic read executor |
| READFST | — | DC | U | login module |
| READMFD | — | DC | U | login module |
| RELEASE | — | T | U | release user disk |
| RELUFD | — | DC | U | login module |
| SCAN | — | N | U | decipher input lines |
| SCSFOR | — | DC | S | script module |
| SCSLNK | — | DC | S | script module |

| External Filename | Internal Entry Point | Usage | Relation vis a vis Ver 3.0 | Functions |
|---|---|---|---|---|
| SCSPRT | SCRIPT | D | S | script processor |
| SOABEND | N | N | ABSTD | *sim of OS ABEND |
| SOBDAM | — | NS | N | *sim of BDAM |
| SOBSAM | — | NS | R-RDWR | *sim of BSAM |
| SOCNTRL | — | NS | R-NTPT, CHECK | *sim of NOTE/POINT/CHECK |
| SOEOB | — | NS | N | *sim of end-of-block |
| SOLINKS | — | NS | UN-LINKAGE | *sim of control transfers |
| SOMAIN | — | NS | RN-STORAGE | *sim of SVC 4, 5, 10 |
| SOOPCL | — | NS | R-OPEN | *sim of SVC 19, 20, 22, 23 |
| SOQSAM | — | NS | R-GET | *sim of GET, PUT |
| SORT | — | D | U | sort data within files |
| SORTREE | — | DC | S | sort module |
| SORTSRCH | — | DC | S | sort module |
| SOSVCNU | — | NS | N | *sim of misc SVC in nucleus |
| SOSVCTR | — | T | R-SVCCARE | *sim of misc SVC in transient |
| SOSVCT2 | — | T | N-SVCCARE | *sim of misc SVC in transient (con' |
| SPLIT | — | D | U | file utility |
| START | — | N | S | commence execution |
| STATDSK | STAT | T | U | disk statistician |
| STATE | — | NS | U | file lookup |
| SYN | — | T | U | synonym processor |
| TAPE | — | D | U | tape utility |
| TAPEIO | — | T | S | tape I/O executor |
| TAPRINT | — | D | S | listing tape utility |
| TPCOPY | — | D | S | copy tapes |
| TRAP | — | N | S | interrupt trap |
| TRKLKP | — | NS | S | track management |
| TXTLIB | — | D | U | library processor |
| UPDATE | — | D | U | file utility |
| UPDISK | — | NS | U | disk management |
| USE | — | N | U | load initiator |
| WAIT | — | NS | S | I/O wait |
| WRBUF | — | NS | U | basic write executor |
| WRTAPE | — | D | U | tape write utility |

* "sim of" means "CMS simulation of the OS function . . ."

## APPENDIX B: CMS DIRECTORY

### DISK-RESIDENT ROUTINES

Each CMS MODULE is the result of LOAD'ing the TEXT (Object) deck of one or more source programs, and then executing the GENMOD command. The following is a brief description of CMS MODULE's and the TEXT routines they contain.

| MODULE | FUNCTION | TEXT | COMMENTS |
|---|---|---|---|
| ALTER | change file identification | ALTER | transient |
| ASSEMBLE | interface, I/O processor | ASSEMBLE,<br>IEUASM,<br>ASMDIRT | see "ASMGEND" |
| BRUIN | interface, language processor | BRUIN,<br>BRUINTXT | see BRUIN section in PLM |
| CEDIT | file editor | CEDIT<br>EDITDUAL | unlimited file size |
| CNVT26 | 026-029 converter | CNVT26 | |
| COMBINE | file concatenation | COMBINE | -- |
| COMPARE | file comparison | COMPARE | -- |
| CVTFV | fixed-variable converter | CNVTFV | |
| DISK | file utility | DISK | transient |
| DUMPD | disk dump | DUMPD | -- |
| DUMPF | file dump | DUMPF | -- |
| DUMPREST | dump/restore | DMPRST | can be stand-alone |
| ECHO | console test | ECHO | -- |
| EDIT | file editor | EDIT,<br>EDITDUAL | in-core editor |
| EXECTOR | EXEC work routine | EXECTOR | relocatable module |
| FORMAT | disk organizer | CMSFORM | -- |

| MODULE | FUNCTION | TEXT | COMMENTS |
|---|---|---|---|
| FORTRAN | interface, I/O handler | FORTRAN,<br>FORTOUT,<br>FORDIRT | see "FORTGEND" |
| GENDIRT | system programmer use only | GENDIRT | used to complete auxiliary directories (transient) |
| GLOBAL | designate libraries | GLOBAL | transient |
| HNDINT | alternate I/O interrupt processors | HNDINT | transient |
| HNDSVC | alternate SVC handling routines | HNDSVC | transient |
| IPL | disk load simulator | CMSIPL | transient |
| LISTF | display user disk contents | LISTF, | transient |
| MACLIB | macro library manipulator | MACLIB | -- |
| MAPPRT | nucleus load map | MAPPRT | -- |
| MODMAP | module load map | MODMAP | transient |
| OFFLINE | unit record utility | OFFLINE, | transient |
| OVERRIDE | SVC tracing processor | OVERRIDE,<br>OVERSUB | relocatable module |
| PLI | interface, I/O handler | PLI,<br>PLIDIRT | see "PLIGEND" |
| PRINTF | display file | PRINTF | transient |
| RELEASE | release disk<br>no longer needed | RELEASE | transient |
| SCRIPT | script processor | SCSPRT,<br>SCSLNK,<br>SCSFOR | -- |
| SNOBOL | string processor | SPL1 -<br>ASM, INT,<br>EXT, FRE,<br>IOS, CTL | -- |

| MODULE | FUNCTION | TEXT | COMMENTS |
|---|---|---|---|
| SORT | sort utility | SORT,<br>SORTREE<br>SORTSRCH | -- |
| SPLIT | file division | SPLIT | -- |
| STAT | disk accounting | STATDSK | transient |
| SOSVCTR | disk resident OS SVC simulation routines | SOSVCTR | transient |
| SOSVCT2 | disk resident OS SVC simulation routines | SOSVCT2 | transient |
| TAPE | tape utility | TAPE | -- |
| TAPEIO | tape I/O processor | TAPEIO | transient |
| TAPRINT | print tape assembly listings | TAPRINT | must have used LTAPE option |
| TPCOPY | tape duplicator | TPCOPY | -- |
| TXTLIB | text library manipulator | TXTLIB | -- |
| UPDATE | file changer | UPDATE | maintain SOURCE code |
| WRTAPE | create listings tape | WRTAPE | tape read only by TAPRINT |

Note: A series of EXEC procedures entitled "nnnGEND" is used to produce modules and overlay structures for all CMS commands and language processors:

| | | |
|---|---|---|
| CMSGEND | - | entire CMS command structure |
| ASMGEND | - | Assembler |
| FORTGEND | - | Fortran |
| PLIGEND | - | PL/I |

Also, each language processor will invoke other modules:

| | | |
|---|---|---|
| ASSEMBLER | - | all modules prefixed: "IEU" |
| FORTRAN | - | all modules prefixed: "IEY" |
| PLI | - | all modules prefixed: "IEM" |

NUCLEUS ROUTINES

The core-resident CMS nucleus consists of many routines. The object (TEXT) decks of all these routines are contained in the file "NUCLEUS type", where 'type' is the version identification (for example, 3.1). Filename is the CMS identifier of the Object deck.

START is the label on the internal START card for the routine. Filename names are only given if different from the START label.

| FILENAME (if dissimilar) | START | COMMENTS |
|---|---|---|
| - | ABBREV | synonym or short-name verification |
| ABSTD | ABEND | abnormal job termination |
| - | ACTLKP | active file table lookup |
| - | ADTLKP | active disk table lookup |
| - | ATTN | stack input lines |
| INITB67 | BAREMACH | bare machine initiator |
| - | BATCH | Batch monitor control |
| CARDIO | CARDRD | card I/O utility |
| - | CHCK | BSAM I/O function |
| - | CLOSIO | terminate unit record activity |
| CMSTIME | CMSTIMER | time statistics |
| - | CONSI | console, interrupt |
| - | CONWAIT | clear console activity |
| - | DEBUG | programmer aid |
| DISKINT | DISK | disk interrupt handler |
| - | DSKERR | disk error handler |
| - | ERASE | eradicate disk files |
| - | EXEC | command initiator |
| FREEXTN | EXTEND | system free storage handler |
| CMSEXTIT | EXTINT | external interrupt handler |
| - | FCBTAB | file control block table |
| - | FINIS | close out active I/O blocks |

| FILENAME (if dissimilar) | START | COMMENTS |
|---|---|---|
| FREESYS | FREE | system free storage |
| - | FREELIST | work area |
| - | FSTLKP | file search |
| - | FUNCTAB | index of nucleus commands |
| - | FVS | file management storage |
| - | GENMOD | write modules |
| - | GET | QSAM simulator |
| - | GETSYM | obtain variable symbol |
| - | IADT | initial active disk table |
| - | INIT | user/terminal interface |
| - | INITSUB | initial nucleus at login |
| INTIO | IOINT | I/O interrupt handler |
| - | IPLDISK | reads/writes loadable nucleus |
| LDR | XEQQ | relocatable loader |
| - | LDRIO | loader I/O utility |
| - | LDRSUBS | loader utility routines |
| LDRLIBE | LIBE | loader library search |
| SOLINKS | XCTL | simulation for SVC 6, 7, 8, 9, 3 |
| - | LOAD | loader initiator command |
| - | LOG | activate file maintenance |
| - | NTPT | BSAM I/O function |
| NUCON | NUCON | nucleus constants, SYSREF device table |
| - | OPEN | activate data control blocks |

| FILENAME (if dissimilar) | START | COMMENTS |
|---|---|---|
| - | POINT | CMS file management |
| INTPROG | PRGINT | program interrupt handler |
| - | PRINTR | printer I/O utility |
| - | QQTRK | assigns FCL blocks |
| - | RDBUF | basic file read operation |
| DISKIO | ADTK | CMS disk I/O executor |
| - | RDWR | BSAM simulator |
| - | READFST | read FST |
| - | READMFD | read MFD |
| - | RELUFD | release UFD |
| - | SCAN | align input line |
| - | STORAGE | simulation of SVC 4, 5, 10 |
| - | START | execution |
| - | STATE | verify file existence |
| - | SVCFREE | path-way to FREE |
| INTSVC | SVCINT | SVC interrupt handler |
| - | SVCSECT | data section for SVC calls |
| - | SWITCH | data control section |
| - | SYSGEN | system directory initializer |
| - | TPLIST | I/O data list for tape routines |
| - | TRAP | external interrupt user routines |
| INITIPL | TRANSAR | end-of-nucleus storage |
| - | TRKLKP | track allocation |
| - | TYPE | output terminal I/O |

| FILENAME (if dissimilar) | START | COMMENTS |
|---|---|---|
| - | UPDISK | update UFD onto disk |
| - | USE | loader continuation |
| - | WAIT | CMS I/O interrupt handler |
| - | WAITRD | issues terminal read |
| - | WAITREG | dummy section |
| - | OVERNUC | override initiator |
| - | WRBUF | basic file write routine |

TRANSIENT ROUTINES

| Module | Text | Function |
|---|---|---|
| ALTER | ALTER | change file identification |
| CPFUNCTN | CMSCONF | invoke virtual console function |
| DISK | DISK | disk-card utility |
| GENDIRT | GENDIRT | system programmer tool |
| GLOBAL | GLOBAL | alter library reference lists |
| HNDINT | HNDINT | specify alternate I/O interrupt routine |
| HNDSVC | HNDSVC | specify alternate SVC listing routine |
| IPL | CMSIPL | initial-program-load simulator |
| LISTS | LISTF | list use=disk contents |
| LOGIN | LOGIN | initiate disk processing |
| MODMAP | MODMAP | type load map |
| OFFLINE | OFFLINE | disk-card-printer utility |
| PRINTF | PRINTF | type file contents |

| Module | Text | Function |
|---|---|---|
| RELEASE | RELEASE | terminate disk processing |
| STAT | STATDSK | type disk statistics |
| SOSVCTR | SVCCARE | OS simulation routines |
| SOSVCT2 | SOSVCT2 | OS simulations routines |
| SYN | SYN | set user abbreviation |
| TAPEIO | TAPEIO | tape I/O handler |

## APPENDIX C: CMS FILE SYSTEM CROSS-REFERENCE LIST (PAGE 1 OF 3)

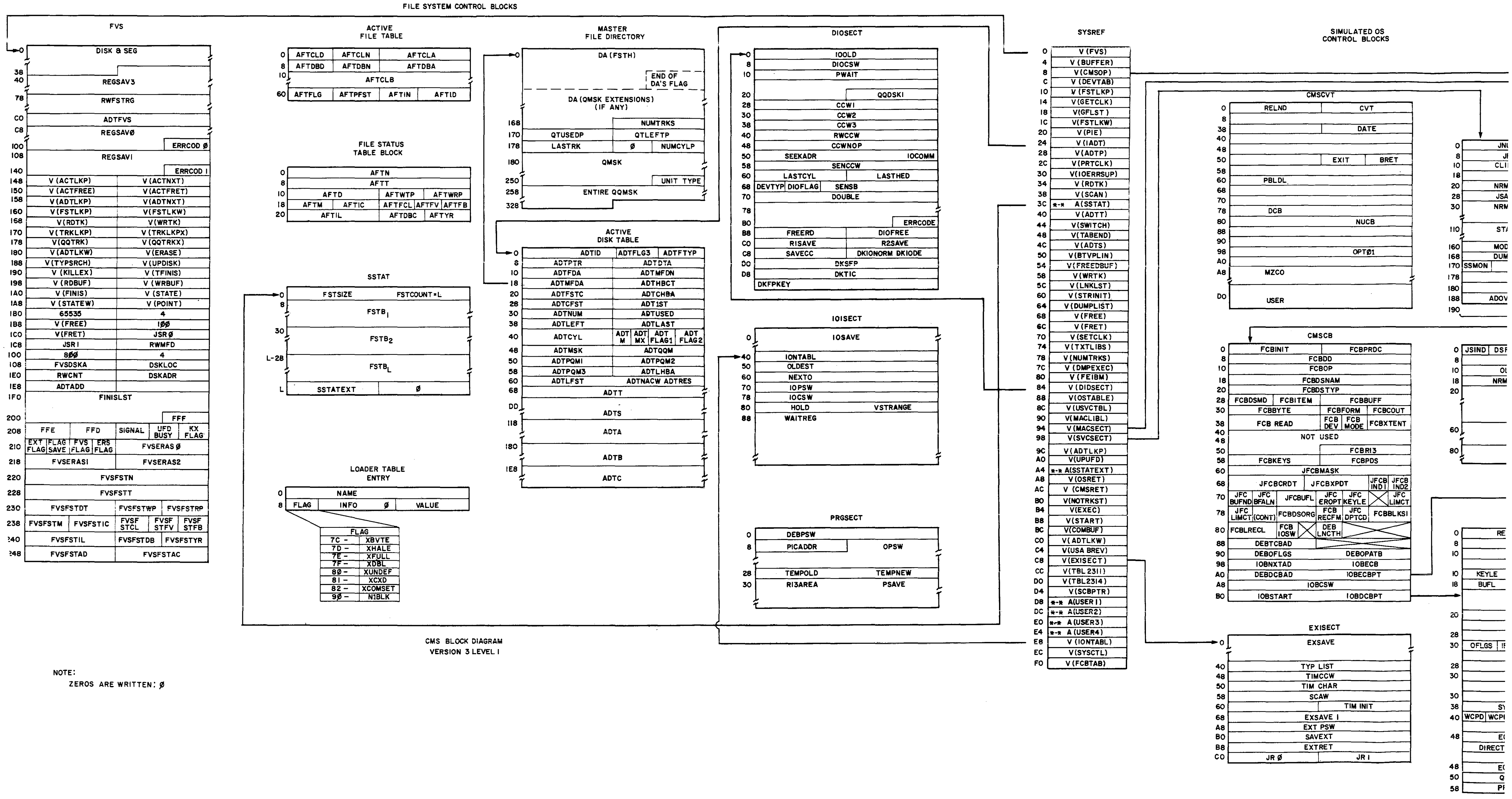
| NAME OF PROGRAM | ENTRY POINTS | I/O MACROS REFERENCED | CALLS | CALLED BY (WHERE KNOWN) |
|---|---|---|---|---|
| ACTLKP | ACTLKP | AFT | | ALTER,ERASE,FINIS,POINT,RDBUF, STATE,STATEW,TFINIS,WRBUF |
| | ACTNXT | AFT | | ERASE |
| | ACTFREE | AFT,ADT,FSTB | FREE | POINT,RDBUF,WRBUF |
| | ACTFRET | AFT | FRET | ERASE,FINIS,WRBUF |
| ADTLKP | ADTLKP | ADT | | ALTER,FORMAT,FSTLKP/FSTLKW,LISTF,LOGIN, RDTK/WRTK,RELEASE,STAT,TAPE,WRBUF |
| | ADTNXT | ADT | | FSTLKP/FSTLKW,LISTF,LOGIN,STAT |
| ALTER | ALTER | ADT,AFT,FSTB, FVS | ACTLKP,ADTLKP,FSTLKW, TFINIS,UPDISK | BRUIN,CEDIT,CNVT26,COMBINE,CVTFV, EDIT,MACLIB,TXTLIB,UPDATE |
| **DISKIO** | RDTK | ADT,DIOSCT, FVS | ADTLKP,DSKERR,FREE FRET,WAIT | ERASE,FINIS,FORMAT,RDBUF,READFST, READMFD,TFINIS,WRBUF |
| | WRTK | SAME AS RDTK | | FINIS,FORMAT,READMFD,TFINIS,UPDISK,WRBUF |
| DISK | DISK | FSTB,FVS | ERASE,FINIS,FSTLKP, RDBUF,STATE,UPDISK, WRBUF | USER |
| DSKERR | DSKERR | ADT,DIOSCT | | RDTK/WRTK |
| ERASE | ERASE | ADT,AFT, FSTB,FVS | ACTFRET,ACTLKP,ACTNXT, DISKDIE,FREE,FRET, FSTLKW,QQTRKX,RDTK, TFINIS,TRKLKPX,UPDISK | ASSEMBLE,BRUIN,CEDIT,CNVT26,COMBINE CVTFV,DISK,EDIT,FINIS,FORTRAN,GENMOD, LISTF,LOAD,MACLIB,MAPPRT,OFFLINE,PLI, SCRIPT,SORT,TAPE,TXTLIB,UPDATE |
| FINIS | FINIS | ADT,AFT, FSTB,FVS | ACTFRET,ACTLKP,DISKDIE, ERASE,FREE,FRET,FSTLKW, RDTK,UPDISK,WRTK | GENMOD,LOADMOD,LOGDISK,UPUFD, AND ALL COMMANDS WHICH USE RDBUF & WRBUF. |
| | TFINIS | SAME AS FINIS | ACTLKP,DISKDIE,FSTLKW, RDTK,WRTK,FRET | ALTER,ERASE |
| | DISKDIE | (NONE) | | ERASE,FINIS,TFINIS,UPDISK,WRBUF |

## APPENDIX C: CMS FILE SYSTEM CROSS-REFERENCE LIST (PAGE 2 OF 3)

| NAME OF PROGRAM | ENTRY POINTS | I/O MACROS REFERENCE | CALLS | CALLED BY (WHERE KNOWN) |
|---|---|---|---|---|
| CMSFORM | FORMAT | ADT,FVS | ADTLKP,FREE,FRET,RDTK RELUFD,UPDISK,WRTK | INIT,USER |
| FSTLKP | FSTLKP | ADT,FVS | ADTLKP,ADTNXT | DISK,INIT,POINT,RDBUF,STATE,TAPE |
| | FSTLKW | ADT,FVS | ADTLKP,ADTNXT,FREE | ALTER,ERASE,FINIS,STATEW, TAPE,TFINIS,WRBUF |
| GENMOD | GENMOD | FVS | ERASE,FINIS, START,WRBUF | USER, AND EXEC PROCEDURES WHICH GENERATE MODULES |
| | LOADMOD | FVS | FINIS,RDBUF,STATE | LINKAGE,SVCINT,$ |
| INIT | INIT | (NONE) | FORMAT,FSTLKP,INITSUB LOGIN,READFST,UPUFD | IPL CMS, IPL, IPL 190 |
| INITSUB | INITSUB | ADT,DIOSCT | IPLDISK,BAREMACH,SYSGEN | INIT |
| LISTF | LISTF | ADT,FVS | ADTLKP,ADTNXT,ERASE, FINIS,WRBUF | USER |
| LOG | LOGOUT | ADT,FVS | LOGDISK | USER |
| | KILLEX | ADT,FVS | LOGDISK | CONSI,DEBUG |
| | KILLEXF | ADT,FVS | LOGDISK | WRBUF |
| LOGIN | LOGIN | ADT,FVS | ADTLKP,ADTNXT,FREE, FRET,READFST,READMFD, RELUFD,UPDISK | INIT, OR USER |
| POINT | POINT | AFT,FVS | ACTFREE,ACTLKP,FSTLKP | ASSEMBLE,CEDIT,EDIT |
| QQTRK | QQTRK | ADT,FVS | TRKLKP,TRKLKPX | WRBUF |
| | QQTRKX | ADT,FVS | TRKLKPX | ERASE,WRBUF |
| RDBUF | RDBUF | AFT,FSTB,FVS | ACTFREE,ACTLKP,FREE, FRET,FSTLKP,RDTK | LOADMOD, AND ALL PROGRAMS WHICH READ CMS FILES |

## APPENDIX C: CMS FILE SYSTEM CROSS-REFERENCE LIST (PAGE 3 OF 3)

| NAME OF PROGRAM | ENTRY POINTS | I/O MACROS REFERENCED | CALLS | CALLED BY (WHERE KNOWN) |
|---|---|---|---|---|
| READFST | READFST | ADT,FVS. | FREE,FRET,RDTK,READMFD | INIT,LOGIN,SYSGEN |
| READMFD | READMFD | ADT,FVS | FREE,FRET,RDTK,WRTK | READFST,LOGIN |
| RELEASE | RELEASE | ADT,FVS | ADTLKP,CPFUNCTN,RELUFD | USER |
| RELUFD | RELUFD | ADT,FVS | FRET | FORMAT,LOGIN,RELEASE |
| STATDSK | STAT | ADT,FVS | ADTLKP,ADTNXT | USER |
| STATE | STATE | ADT,AFT,FSTB, FVS | ACTLKP,FSTLKP | LOADMOD, AND OTHER PROGRAMS WHICH CHECK THE CHARACTERISTICS OF A FILE |
| | STATEW | SAME AS STATE | ACTLKP,FSTLKW | OFFLINE |
| **INITSYS** | SYSGEN | ADTS (ADT) | FREE,READFST | INITSUB |
| TAPE | TAPE | ADT,FVS | ADTLKP,ERASE,FINIS FSTLKP,FSTLKW,RDBUF UPDISK,WRBUF | USER |
| TRKLKP | TRKLKP | ADT | | QQTRK,UPDISK,WRBUF |
| | TRKLKPX | ADT | | ERASE,QQTRK,QQTRKX,UPDISK,WRBUF |
| UPDISK | UPDISK | ADT,FVS | DISKDIE,FREE,FRET, TRKLKP,TRKLKPX,WRTK | ALTER,DISK,ERASE,FINIS, FORMAT,LOGDISK,LOGIN,TAPE |
| | UPUFD | FVS | FINIS | INIT,ASSEMBLE,FORTRAN,EXECTOR |
| | LOGDISK | ADT,FVS | FINIS,UPDISK | LOGOUT |
| WRBUF | WRBUF | ADT,AFT,FSTB, FVS | ACTFREE,ACTFRET,ACTLKP, ADTLKP,DISKDIE,FREE, FRET,FSTLKW,KILLEXF, QQTRK,QQTRKX,RDTK, TRKLKP,TRKLKPX,WRTK | GENMOD, AND ALL PROGRAMS WHICH WRITE CMS FILES |



FILE SYSTEM CONTROL BLOCKS

FVS

| | | | | | |
|---|---|---|---|---|---|
| 0 | DISK & SEG | | | | |
| 38 | | | | | |
| 40 | REGSAV3 | | | | |
| 78 | RWFSTRG | | | | |
| C0 | ADTFVS | | | | |
| C8 | REGSAVØ | | | | |
| 100 | ERRCOD Ø | | | | |
| 108 | REGSAVI | | | | |
| 140 | ERRCOD I | | | | |
| 148 | V (ACTLKP) | V (ACTNXT) | | | |
| 150 | V (ACTFREE) | V (ACTFRET) | | | |
| 158 | V (ADTLKP) | V (ADTNXT) | | | |
| 160 | V (FSTLKP) | V (FSTLKW) | | | |
| 168 | V (RDTK) | V (WRTK) | | | |
| 170 | V (TRKLKP) | V (TRKLKPX) | | | |
| 178 | V (QQTRK) | V (QQTRKX) | | | |
| 180 | V (ADTLKW) | V (ERASE) | | | |
| 188 | V (TYPSRCH) | V (UPDISK) | | | |
| 190 | V (KILLEX) | V (TFINIS) | | | |
| 198 | V (RDBUF) | V (WRBUF) | | | |
| 1A0 | V (FINIS) | V (STATE) | | | |
| 1A8 | V (STATEW) | V (POINT) | | | |
| 1B0 | 65535 | 4 | | | |
| 1B8 | V (FREE) | 1ØØ | | | |
| 1C0 | V (FRET) | JSR Ø | | | |
| 1C8 | JSR I | RWMFD | | | |
| 100 | 8ØØ | 4 | | | |
| 108 | FVSDSKA | DSKLOC | | | |
| 1E0 | RWCNT | DSKADR | | | |
| 1E8 | ADTADD | | | | |
| 1F0 | FINISLST | | | | |
| 200 | FFF | | | | |
| 208 | FFE | FFD | SIGNAL | UFD BUSY | KX FLAG |
| 210 | EXT FLAG | FLAG SAVE | FVS FLAG | ERS FLAG | FVSERAS Ø |
| 218 | FVSERASI | FVSERAS2 | | | |
| 220 | FVSFSTN | | | | |
| 228 | FVSFSTT | | | | |
| 230 | FVSFSTDT | FVSFSTWP | FVSFSTRP | | |
| 238 | FVSFSTM | FVSFSTIC | FVSF STCL | FVSF STFV | FVSF STFB |
| 240 | FVSFSTIL | FVSFSTDB | FVSFSTYR | | |
| 248 | FVSFSTAD | FVSFSTAC | | | |

ACTIVE FILE TABLE

| | | | | |
|---|---|---|---|---|
| 0 | AFTCLD | AFTCLN | AFTCLA | |
| 8 | AFTDBD | AFTDBN | AFTDBA | |
| 10 | AFTCLB | | | |
| 60 | AFTFLG | AFTPFST | AFTIN | AFTID |

FILE STATUS TABLE BLOCK

| | | | | | |
|---|---|---|---|---|---|
| 0 | AFTN | | | | |
| 8 | AFTT | | | | |
| 10 | AFTD | AFTWTP | AFTWRP | | |
| 18 | AFTM | AFTIC | AFTFCL | AFTFV | AFTFB |
| 20 | AFTIL | AFTDBC | AFTYR | | |

SSTAT

| | | |
|---|---|---|
| 0 | FSTSIZE | FSTCOUNT = L |
| 8 | $FSTB_1$ | |
| 30 | $FSTB_2$ | |
| L-28 | $FSTB_L$ | |
| L | SSTATEXT | Ø |

LOADER TABLE ENTRY

| | | | |
|---|---|---|---|
| 0 | NAME | | |
| 8 | FLAG | INFO Ø | VALUE |

| FLAG | |
|---|---|
| 7C – | XBVTE |
| 7D – | XHALE |
| 7E – | XFULL |
| 7F – | XDBL |
| 8Ø – | XUNDEF |
| 81 – | XCXD |
| 82 – | XCOMSET |
| 9Ø – | N1BLK |

MASTER FILE DIRECTORY

| | | | |
|---|---|---|---|
| 0 | DA (FSTH) | END OF DA'S FLAG | |
| | DA (QMSK EXTENSIONS) (IF ANY) | | |
| 168 | NUMTRKS | | |
| 170 | QTUSEDP | QTLEFTP | |
| 178 | LASTRK | Ø | NUMCYLP |
| 180 | QMSK | | |
| 250 | UNIT TYPE | | |
| 258 | ENTIRE QQMSK | | |
| 328 | | | |

ACTIVE DISK TABLE

| | | | | | |
|---|---|---|---|---|---|
| 0 | ADTID | ADTFLG3 | ADTFTYP | | |
| 8 | ADTPTR | ADTDTA | | | |
| 10 | ADTFDA | ADTMFDN | | | |
| 18 | ADTMFDA | ADTHBCT | | | |
| 20 | ADTFSTC | ADTCHBA | | | |
| 28 | ADTCFST | ADT1ST | | | |
| 30 | ADTNUM | ADTUSED | | | |
| 38 | ADTLEFT | ADTLAST | | | |
| 40 | ADTCYL | ADT M | ADT MX | ADT FLAG1 | ADT FLAG2 |
| 48 | ADTMSK | ADTQQM | | | |
| 50 | ADTPQM1 | ADTPQM2 | | | |
| 58 | ADTPQM3 | ADTLHBA | | | |
| 60 | ADTLFST | ADTNACW ADTRES | | | |
| 68 | ADTT | | | | |
| DD | ADTS | | | | |
| 118 | ADTA | | | | |
| 180 | ADTB | | | | |
| 1E8 | ADTC | | | | |

DIOSECT

| | | | |
|---|---|---|---|
| 0 | IOOLD | | |
| 8 | DIOCSW | | |
| 10 | PWAIT | | |
| 20 | QQDSKI | | |
| 28 | CCW1 | | |
| 30 | CCW2 | | |
| 38 | CCW3 | | |
| 40 | RWCCW | | |
| 48 | CCWNOP | | |
| 50 | SEEKADR | IOCOMM | |
| 58 | SENCCW | | |
| 60 | LASTCYL | LASTHED | |
| 68 | DEVTYP | DIOFLAG | SENSB |
| 70 | DOUBLE | | |
| 78 | | | |
| B0 | ERRCODE | | |
| B8 | FREERD | DIOFREE | |
| C0 | R1SAVE | R2SAVE | |
| C8 | SAVECC | DKIONORM DKIODE | |
| D0 | DKSFP | | |
| D8 | DKTIC | | |
| | DKFPKEY | | |

IO1SECT

| | | |
|---|---|---|
| 0 | IOSAVE | |
| 40 | IONTABL | |
| 50 | OLDEST | |
| 60 | NEXTO | |
| 70 | IOPSW | |
| 78 | IOCSW | |
| 80 | HOLD | VSTRANGE |
| 88 | WAITREG | |

PRGSECT

| | | |
|---|---|---|
| 0 | DEBPSW | |
| 8 | PICADDR | OPSW |
| 28 | TEMPOLD | TEMPNEW |
| 30 | R13AREA | PSAVE |

SYSREF

| | |
|---|---|
| 0 | V (FVS) |
| 4 | V (BUFFER) |
| 8 | V (CMSOP) |
| C | V (DEVTAB) |
| 10 | V (FSTLKP) |
| 14 | V (GETCLK) |
| 18 | V (GFLST) |
| 1C | V (FSTLKW) |
| 20 | V (PIE) |
| 24 | V (IADT) |
| 28 | V (ADTP) |
| 2C | V (PRTCLK) |
| 30 | V (IOERRSUP) |
| 34 | V (RDTK) |
| 38 | V (SCAN) |
| 3C | *-* A (SSTAT) |
| 40 | V (ADTT) |
| 44 | V (SWITCH) |
| 48 | V (TABEND) |
| 4C | V (ADTS) |
| 50 | V (BTVPLIN) |
| 54 | V (FREEDBUF) |
| 58 | V (WRTK) |
| 5C | V (LNKLST) |
| 60 | V (STRINIT) |
| 64 | V (DUMPLIST) |
| 68 | V (FREE) |
| 6C | V (FRET) |
| 70 | V (SETCLK) |
| 74 | V (TXTLIBS) |
| 78 | V (NUMTRKS) |
| 7C | V (DMPEXEC) |
| 80 | V (FEIBM) |
| 84 | V (DIDSECT) |
| 88 | V (OSTABLE) |
| 8C | V (USVCTBL) |
| 90 | V (MACLIBL) |
| 94 | V (MACSECT) |
| 98 | V (SVCSECT) |
| 9C | V (ADTLKP) |
| A0 | V (UPUFD) |
| A4 | *-* A (SSTATEXT) |
| A8 | V (OSRET) |
| AC | V (CMSRET) |
| B0 | V (NOTRKST) |
| B4 | V (EXEC) |
| B8 | V (START) |
| BC | V (COMBUF) |
| C0 | V (ADTLKW) |
| C4 | V (USA BREV) |
| C8 | V (EXISECT) |
| CC | V (TBL 2311) |
| D0 | V (TBL2314) |
| D4 | V (SCBPTR) |
| D8 | *-* A (USER 1) |
| DC | *-* A (USER2) |
| E0 | *-* A (USER3) |
| E4 | *-* A (USER4) |
| E8 | V (IONTABL) |
| EC | V (SYSCTL) |
| F0 | V (FCBTAB) |

SIMULATED OS CONTROL BLOCKS

CMSCVT

| | | | |
|---|---|---|---|
| 0 | RELND | CVT | |
| 8 | | | |
| 38 | | DATE | |
| 40 | | | |
| 48 | | | |
| 50 | | EXIT | BRET |
| 58 | | | |
| 60 | PBLDL | | |
| 68 | | | |
| 70 | | | |
| 78 | DCB | | |
| 80 | | NUCB | |
| 88 | | | |
| 90 | | | |
| 98 | | OPTØ1 | |
| A0 | | | |
| A8 | MZCO | | |
| D0 | USER | | |

CMSCB

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | FCBINIT | FCBPRDC | | | | | |
| 8 | FCBDD | | | | | | |
| 10 | FCBOP | | | | | | |
| 18 | FCBDSNAM | | | | | | |
| 20 | FCBDSTYP | | | | | | |
| 28 | FCBDSMD | FCBITEM | FCBBUFF | | | | |
| 30 | FCBBYTE | FCBFORM | FCBCOUT | | | | |
| 38 | FCB READ | FCB DEV | FCB MODE | FCBXTENT | | | |
| 40<br>48 | NOT USED | | | | | | |
| 50 | FCBR13 | | | | | | |
| 58 | FCBKEYS | FCBPDS | | | | | |
| 60 | JFCBMASK | | | | | | |
| 68 | JFCBCRDT | JFCBXPDT | JFCB IND1 | JFCB IND2 | | | |
| 70 | JFC BUFND | JFC BFALN | JFCBUFL | JFC EROPT | JFC KEYLE | | JFC LIMCT |
| 78 | JFC LIMCT | (CONT) | FCBDSORG | FCB RECFM | JFC DPTCD | FCBBLKSI | |
| 80 | FCBLRECL | FCB IOSW | | DEB LNCTH | | | |
| 88 | DEBTCBAD | | | | | | |
| 90 | DEBOFLGS | DEBOPATB | | | | | |
| 98 | IOBNXTAD | IOBECB | | | | | |
| A0 | DEBDCBAD | IOBECBPT | | | | | |
| A8 | IOBCSW | | | | | | |
| B0 | IOBSTART | IOBDCBPT | | | | | |

EXISECT

| | | |
|---|---|---|
| 0 | EXSAVE | |
| 40 | TYP LIST | |
| 48 | TIMCCW | |
| 50 | TIM CHAR | |
| 58 | SCAW | |
| 60 | | TIM INIT |
| 68 | EXSAVE I | |
| A8 | EXT PSW | |
| B0 | SAVEXT | |
| B8 | EXTRET | |
| C0 | JR Ø | JR I |

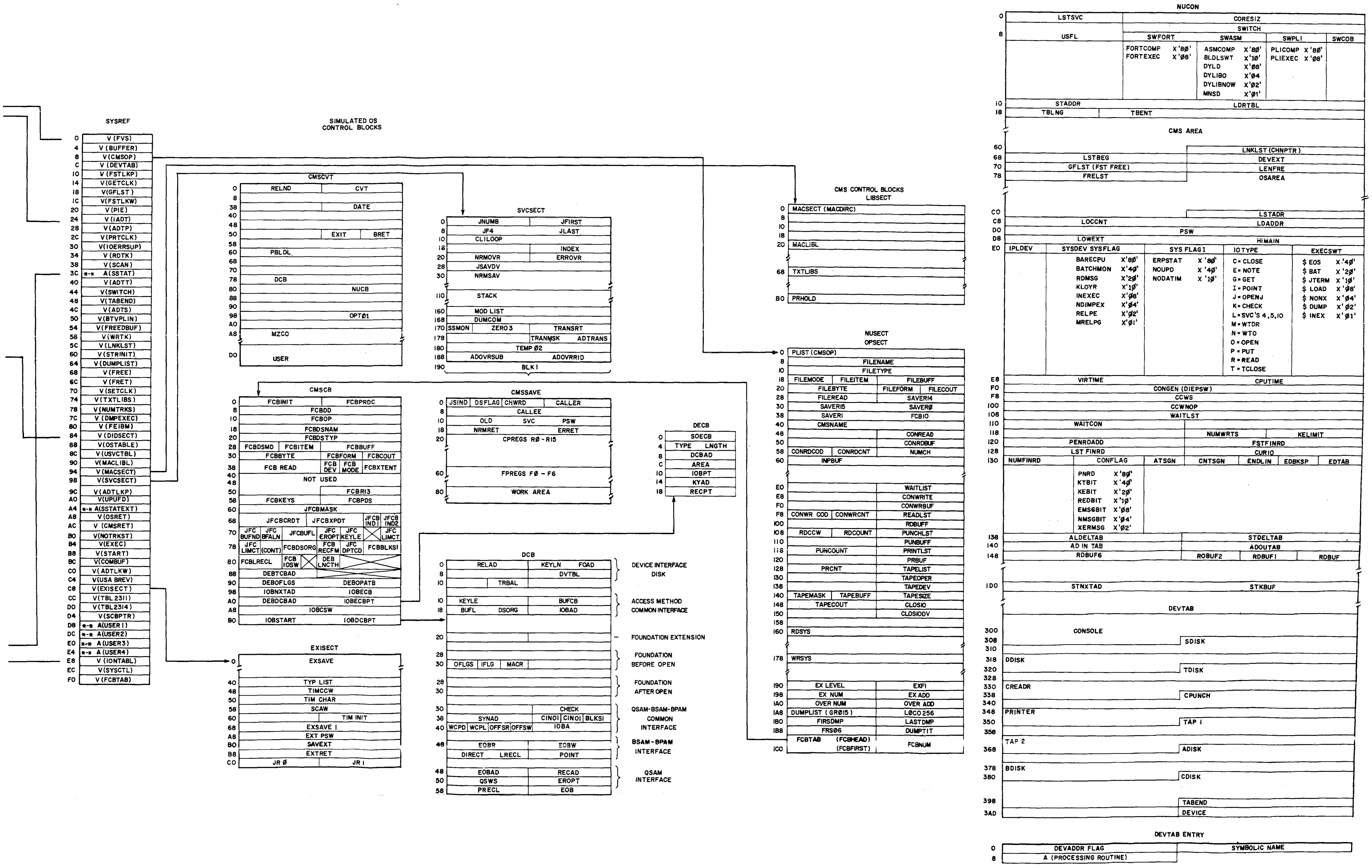
| | |
|---|---|
| 0 | JNI |
| 8 | JI |
| 10 | CLI |
| 18 | |
| 20 | NRM |
| 28 | JSA |
| 30 | NRM |
| 110 | STA |
| 160 | MOD |
| 168 | DUM |
| 170 | SSMON |
| 178 | |
| 180 | |
| 188 | ADOV |
| 190 | |

| | | |
|---|---|---|
| 0 | JSIND | DSF |
| 8 | | |
| 10 | OI | |
| 18 | NRM | |
| 20 | | |
| 60 | | |
| 80 | | |

| | | |
|---|---|---|
| 0 | RE | |
| 8 | | |
| 10 | | |
| 10 | KEYLE | |
| 18 | BUFL | |
| 20 | | |
| 28 | | |
| 30 | OFLGS | II |
| 28 | | |
| 30 | | |
| 30 | | |
| 38 | SY | |
| 40 | WCPD | WCPI |
| 48 | EI | |
| | DIRECT | |
| 48 | EI | |
| 50 | Q | |
| 58 | PI | |

CMS BLOCK DIAGRAM
VERSION 3 LEVEL I

NOTE:
ZEROS ARE WRITTEN: Ø



### SYSREF

| | |
|---|---|
| 0 | V (FVS) |
| 4 | V (BUFFER) |
| 8 | V(CMSOP) |
| C | V (DEVTAB) |
| 10 | V (FSTLKP) |
| 14 | V(GETCLK) |
| 18 | V(GFLST ) |
| 1C | V(FSTLKW) |
| 20 | V (PIE) |
| 24 | V (IADT) |
| 28 | V (ADTP) |
| 2C | V (PRTCLK) |
| 30 | V(IOERRSUP) |
| 34 | V (RDTK) |
| 38 | V (SCAN) |
| 3C | *-* A(SSTAT) |
| 40 | V (ADTT) |
| 44 | V(SWITCH) |
| 48 | V( TABEND) |
| 4C | V(ADTS) |
| 50 | V(BTVPLIN) |
| 54 | V(FREEDBUF) |
| 58 | V(WRTK) |
| 5C | V(LNKLST) |
| 60 | V(STRINIT) |
| 64 | V(DUMPLIST) |
| 68 | V(FREE) |
| 6C | V(FRET) |
| 70 | V(SETCLK) |
| 74 | V(TXTLIBS) |
| 78 | V(NUMTRKS) |
| 7C | V (DMPEXEC) |
| 80 | V (FEIBM) |
| 84 | V (DIDSECT) |
| 88 | V(OSTABLE) |
| 8C | V(USVCTBL) |
| 90 | V(MACLIBL) |
| 94 | V (MACSECT) |
| 98 | V(SVCSECT) |
| 9C | V(ADTLKP) |
| A0 | V(UPUFD) |
| A4 | *-* A(SSTATEXT) |
| A8 | V (OSRET) |
| AC | V (CMSRET) |
| B0 | V(NOTRKST) |
| B4 | V(EXEC) |
| B8 | V(START) |
| BC | V(COMBUF) |
| C0 | V( ADTLKW) |
| C4 | V(USA BREV) |
| C8 | V(EXISECT) |
| CC | V(TBL 2311) |
| D0 | V(TBL2314) |
| D4 | V(SCBPTR) |
| D8 | *-* A(USER1) |
| DC | *-* A(USER2) |
| E0 | *-* A (USER3) |
| E4 | *-* A (USER4) |
| E8 | V (IONTABL) |
| EC | V(SYSCTL) |
| F0 | V (FCBTAB) |

## SIMULATED OS CONTROL BLOCKS

### CMSCVT

| | | | |
|---|---|---|---|
| 0 | RELND | CVT | |
| 8 | | | |
| 38 | | DATE | |
| 40 | | | |
| 48 | | | |
| 50 | | EXIT | BRET |
| 58 | | | |
| 60 | PBLDL | | |
| 68 | | | |
| 70 | | | |
| 78 | DCB | | |
| 80 | | NUCB | |
| 88 | | | |
| 90 | | | |
| 98 | | OPTØ1 | |
| A0 | | | |
| A8 | MZCO | | |
| D0 | USER | | |

### SVCSECT

| | | | |
|---|---|---|---|
| 0 | JNUMB | JFIRST | |
| 8 | JF4 | JLAST | |
| 10 | CLILOOP | | |
| 18 | | INDEX | |
| 20 | NRMOVR | ERROVR | |
| 28 | JSAVDV | | |
| 30 | NRMSAV | | |
| 110 | STACK | | |
| 160 | MOD LIST | | |
| 168 | DUMCOM | | |
| 170 | SSMON | ZERO3 | TRANSRT |
| 178 | | TRANMSK | ADTRANS |
| 180 | TEMP Ø2 | | |
| 188 | ADOVRSUB | ADOVRRID | |
| 190 | BLK1 | | |

### CMSCB

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | FCBINIT | FCBPRDC | | | | | |
| 8 | FCBDD | | | | | | |
| 10 | FCBOP | | | | | | |
| 18 | FCBDSNAM | | | | | | |
| 20 | FCBDSTYP | | | | | | |
| 28 | FCBDSMD | FCBITEM | FCBBUFF | | | | |
| 30 | FCBBYTE | FCBFORM | FCBCOUT | | | | |
| 38 | FCB READ | FCB DEV | FCB MODE | FCBXTENT | | | |
| 40<br>48 | NOT USED | | | | | | |
| 50 | | FCBRI3 | | | | | |
| 58 | FCBKEYS | FCBPDS | | | | | |
| 60 | JFCBMASK | | | | | | |
| 68 | JFCBCRDT | JFCBXPDT | JFCB IND1 | JFCB IND2 | | | |
| 70 | JFC BUFND | JFC BFALN | JFCBUFL | JFC EROPT | JFC KEYLE | | JFC LIMCT |
| 78 | JFC LIMCT | (CONT) | FCBDSORG | FCB RECFM | JFC DPTCD | FCBBLKSI | |
| 80 | FCBLRECL | FCB IOSW | | DEB LNCTH | | | |
| 88 | DEBTCBAD | | | | | | |
| 90 | DEBOFLGS | DEBOPATB | | | | | |
| 98 | IOBNXTAD | IOBECB | | | | | |
| A0 | DEBDCBAD | IOBECBPT | | | | | |
| A8 | IOBCSW | | | | | | |
| B0 | IOBSTART | IOBDCBPT | | | | | |

### CMSSAVE

| | | | | |
|---|---|---|---|---|
| 0 | JSIND | DSFLAG | CHWRD | CALLER |
| 8 | CALLEE | | | |
| 10 | OLD | SVC | PSW | |
| 18 | NRMRET | ERRET | | |
| 20 | CPREGS RØ - R15 | | | |
| 60 | FPREGS FØ - F6 | | | |
| 80 | WORK AREA | | | |

### DECB

| | | |
|---|---|---|
| 0 | SOECB | |
| 4 | TYPE | LNGTH |
| 8 | DCBAD | |
| C | AREA | |
| 10 | IOBPT | |
| 14 | KYAD | |
| 18 | RECPT | |

### DCB

| | | | | | |
|---|---|---|---|---|---|
| 0 | RELAD | | KEYLN | FOAD | DEVICE INTERFACE DISK |
| 8 | | | DVTBL | | |
| 10 | | TRBAL | | | |
| | | | | | |
| 10 | KEYLE | | BUFCB | | ACCESS METHOD COMMON INTERFACE |
| 18 | BUFL | DSORG | IOBAD | | |
| | | | | | |
| 20 | | | | | FOUNDATION EXTENSION |
| | | | | | |
| 28 | | | | | FOUNDATION BEFORE OPEN |
| 30 | OFLGS | IFLG | MACR | | |
| | | | | | |
| 28 | | | | | FOUNDATION AFTER OPEN |
| 30 | | | | | |
| | | | | | |
| 30 | | | CHECK | | QSAM-BSAM-BPAM COMMON INTERFACE |
| 38 | SYNAD | | CINOI | CINOI BLKSI | |
| 40 | WCPD WCPL | OFFSR OFFSW | IOBA | | |
| | | | | | |
| 48 | EOBR | | EOBW | | BSAM-BPAM INTERFACE |
| | DIRECT | LRECL | POINT | | |
| | | | | | |
| 48 | EOBAD | | RECAD | | QSAM INTERFACE |
| 50 | QSWS | | EROPT | | |
| 58 | PRECL | | EOB | | |

### EXISECT

| | | |
|---|---|---|
| 0 | EXSAVE | |
| 40 | TYP LIST | |
| 48 | TIMCCW | |
| 50 | TIM CHAR | |
| 58 | SCAW | |
| 60 | | TIM INIT |
| 68 | EXSAVE 1 | |
| A8 | EXT PSW | |
| B0 | SAVEXT | |
| B8 | EXTRET | |
| C0 | JR Ø | JR 1 |

## CMS CONTROL BLOCKS

### LIBSECT

| | |
|---|---|
| 0 | MACSECT (MACDIRC) |
| 8 | |
| 10 | |
| 18 | |
| 20 | MACLIBL |
| 68 | TXTLIBS |
| 80 | PRHOLD |

### NUSECT OPSECT

| | | | | |
|---|---|---|---|---|
| 0 | PLIST (CMSOP) | | | |
| 8 | FILENAME | | | |
| 10 | FILETYPE | | | |
| 18 | FILEMODE | FILEITEM | FILEBUFF | |
| 20 | FILEBYTE | | FILEFORM | FILECOUT |
| 28 | FILEREAD | | SAVER14 | |
| 30 | SAVER15 | | SAVERØ | |
| 38 | SAVER1 | | FCBIO | |
| 40 | CMSNAME | | | |
| 48 | | | CONREAD | |
| 50 | | | CONRDBUF | |
| 58 | CONRDCOD | CONRDCNT | NUMCH | |
| 60 | INPBUF | | | |
| E0 | | | WAITLIST | |
| E8 | | | CONWRITE | |
| F0 | | | CONWRBUF | |
| F8 | CONWR COD | CONWRCNT | READLST | |
| 100 | | | RDBUFF | |
| 108 | RDCCW | RDCOUNT | PUNCHLST | |
| 110 | | | PUNBUFF | |
| 118 | PUNCOUNT | | PRINTLST | |
| 120 | | | PRBUF | |
| 128 | PRCNT | | TAPELIST | |
| 130 | | | TAPEOPER | |
| 138 | | | TAPEDEV | |
| 140 | TAPEMASK | TAPEBUFF | TAPESIZE | |
| 148 | TAPECOUT | | CLOSIO | |
| 150 | | | CLOSIODV | |
| 158 | | | | |
| 160 | RDSYS | | | |
| 178 | WRSYS | | | |
| 190 | EX LEVEL | | EXF1 | |
| 198 | EX NUM | | EX ADD | |
| 1A0 | OVER NUM | | OVER ADD | |
| 1A8 | DUMPLIST ( GRØ15 ) | | LØCO 256 | |
| 1B0 | FIRSDMP | | LASTDMP | |
| 1B8 | FRSØ6 | | DUMPTIT | |
| 1C0 | FCBTAB | (FCBHEAD)<br>(FCBFIRST) | FCBNUM | |

### NUCON

| | | | | | |
|---|---|---|---|---|---|
| 0 | LSTSVC | CORESIZ | | | |
| 8 | USFL | SWITCH | | | |
| | | SWFORT | SWASM | SWPLI | SWCOB |
| | | FORTCOMP X'8Ø'<br>FORTEXEC X'Ø8' | ASMCOMP X'8Ø'<br>BLDLSWT X'1Ø'<br>DYLD X'Ø8'<br>DYLIBO X'Ø4<br>DYLIBNOW X'Ø2'<br>MNSD X'Ø1' | PLICOMP X'8Ø'<br>PLIEXEC X'Ø8' | |
| 10 | STADDR | LDRTBL | | | |
| 18 | TBLNG | TBENT | | | |

CMS AREA

| | | |
|---|---|---|
| 60 | | LNKLST (CHNPTR) |
| 68 | LSTBEG | DEVEXT |
| 70 | GFLST (FST FREE) | LENFRE |
| 78 | FRELST | OSAREA |
| C0 | | LSTADR |
| C8 | LOCCNT | LDADDR |
| D0 | PSW | |
| D8 | LOWEXT | HIMAIN |

| | | | | | |
|---|---|---|---|---|---|
| E0 | IPLDEV | SYSDEV SYSFLAG | SYS FLAG1 | IOTYPE | EXECSWT |
| | | BARECPU X'8Ø'<br>BATCHMON X'4Ø'<br>RDMSG X'2Ø'<br>KLOYR X'1Ø'<br>INEXEC X'Ø8'<br>NDIMPEX X'Ø4'<br>RELPE X'Ø2'<br>MRELPG X'Ø1' | ERPSTAT X'8Ø'<br>NOUPD X'4Ø'<br>NODATIM X'1Ø' | C = CLOSE<br>E = NOTE<br>G = GET<br>I = POINT<br>J = OPENJ<br>K = CHECK<br>L = SVC'S 4,5,10<br>M = WTDR<br>N = WTO<br>O = OPEN<br>P = PUT<br>R = READ<br>T = TCLOSE | $ EOS X'4Ø'<br>$ BAT X'2Ø'<br>$ JTERM X'1Ø'<br>$ LOAD X'Ø8'<br>$ NONX X'Ø4'<br>$ DUMP X'Ø2'<br>$ INEX X'Ø1' |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| E8 | VIRTIME | | | CPUTIME | | | |
| F0 | CONGEN (DIEPSW) | | | | | | |
| F8 | CCWS | | | | | | |
| 100 | CCWNOP | | | | | | |
| 108 | WAITLST | | | | | | |
| 110 | WAITCON | | | | | | |
| 118 | | | | NUMWRTS | | KELIMIT | |
| 120 | PENRDADD | | | FSTFINRD | | | |
| 128 | LST FINRD | | | CURIO | | | |
| 130 | NUMFINRD | CONFLAG | ATSGN | CNTSGN | ENDLIN | EDBKSP | EDTAB |
| | | PNRD X'8Ø'<br>KTBIT X'4Ø'<br>KEBIT X'2Ø'<br>REDBIT X'1Ø'<br>EMSGBIT X'Ø8'<br>NMSGBIT X'Ø4'<br>XERMSG X'Ø2' | | | | | |
| 138 | ALDELTAB | | | STDELTAB | | | |
| 140 | AD IN TAB | | | ADOUTAB | | | |
| 148 | RDBUF6 | | | ROBUF2 | RDBUF1 | RDBUF | |
| 1D0 | STNXTAD | | | STKBUF | | | |

DEVTAB

| | | |
|---|---|---|
| 300 | CONSOLE | |
| 308 | | SDISK |
| 310 | | |
| 318 | DDISK | |
| 320 | | TDISK |
| 328 | | |
| 330 | CREADR | |
| 338 | | CPUNCH |
| 340 | | |
| 348 | PRINTER | |
| 350 | | TAP 1 |
| 358 | | |
| | TAP 2 | |
| 368 | | ADISK |
| 378 | BDISK | |
| 380 | | CDISK |
| 398 | | TABEND |
| 3AD | | DEVICE |

### DEVTAB ENTRY

| | | |
|---|---|---|
| 0 | DEVADDR FLAG | SYMBOLIC NAME |
| 8 | A (PROCESSING ROUTINE) | |

INDEX

# READER'S COMMENT FORM

Control Program-67/Cambridge Monitoring System GY20-0591-1

Program Logic Manual

Please comment on the usefulness and readability of this publication, suggest additions and deletions, and list specific errors and omissions (give page numbers). All comments and suggestions become the property of IBM. If you wish a reply, be sure to include your name and address.

## COMMENTS

fold fold

fold fold

- Thank you for your cooperation. No postage necessary if mailed in the U.S.A.
FOLD ON TWO LINES, STAPLE AND MAIL.

GY20-0591-1

## YOUR COMMENTS PLEASE...

Your comments on the other side of this form will help us improve future editions of this publication. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material.

fold fold

FIRST CLASS
PERMIT NO. 1359
WHITE PLAINS, N. Y.

**BUSINESS REPLY MAIL**

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY...

IBM Corporation
1133 Westchester Avenue
White Plains, N.Y. 10604

Attention: Technical Publications

fold fold

IBM

**International Business Machines Corporation**
**Data Processing Division**
**1133 Westchester Avenue, White Plains, New York 10604**
**[U.S.A. only]**

**IBM World Trade Corporation**
**821 United Nations Plaza, New York, New York 10017**
**[International]**

GY20-0591-1

Control Program-67/Cambridge Monitoring System PLM Printed in U.S.A. GY20-0591-1

IBM

**International Business Machines Corporation**
**Data Processing Division**
**1133 Westchester Avenue, White Plains, New York 10604**
**(U.S.A. only)**

**IBM World Trade Corporation**
**821 United Nations Plaza, New York, New York 10017**
**(International)**