Honeywell

LEVEL 64

GCOS

JOB CONTROL LANGUAGE USER GUIDE

SERIES 60 (LEVEL 64)

JOB CONTROL LANGUAGE USER GUIDE

SUBJECT

This Manual Complements the Level 64 Job Control Language Reference Manual and Is Intended to Serve as an Aid to the Use of the GCOS Operating System for Series 60 Level 64 Computers

SPECIAL INSTRUCTIONS

For users of Release 0400 this manual replaces Revision 0 dated July 1977, which remains valid for Release 0300 users. Because of extensive revision, change bars have not been used.

SOFTWARE SUPPORTED

Level 64 GCOS Release 0400

ORDER NUMBER

AQ11, Rev. 1

September 1978

Honeywell

Preface

This manual is intended for all users of the Level 64 GCOS operating system and should be used in conjunction with the Job Control Language Reference Manual (referred to in this manual as the JCL Reference Manual).

While there are many areas in common between the topics covered in this manual and the topics of the JCL Reference Manual, the latter is more concerned with statement format specifications and usage regulations; in this manual, the emphasis is placed on how to benefit from the various facilities available through the Job Control Language of Level 64 GCOS.

Section I introduces the basic concepts involved in the passage of a job through the system. The management of unit record input and output (for example, punched card input and line printer output) is described in Section II, and Section III is concerned with the assignment and allocation of disk, tape and cassette files. Sections IV and V respectively deal with the management of system resources, such as memory and devices, and the way in which stored JCL can be modified and executed. Section VI discusses how to change the processing order of JCL statements and deals with the manner in which errors are handled under Level 64 GCOS. Section VII describes the Job Occurrence Report. A description of the Mini-Editor is contained in Appendix A.

Each section of this document is structured according to the heading hierarchy shown below. Each heading indicates the relative level of the text that follows it.

Level	Heading Format
1	**ALL CAPITALS BOLD**
2	**Inital Capitals Bold**
3	ALL CAPITALS MEDIUM
4	Inital capitals Medium

The Level 64 Document Set follows. Many of the manuals may be referenced in the text.

LEVEL 64 DOCUMENT LIST

Order Number	Title
AQ02	*Series 100 Program Mode Operator Guide*
AQ03	*Series 100 Conversion Guide*
AQ04	*Series 200/2000 Conversion Guide*
AQ05	*System 360/370 Conversion Guide*
AQ09	*System Management Guide*
AQ10	*Job Control Language (JCL) Reference Manual*
AQ11	*Job Control Language (JCL) User Guide*
AQ13	*System Operation Operator Guide*
AQ14	*System Operation Console Messages*
AQ18	*Operator Reference Manual*
AQ20	*Data Management Utilities Manual*
AQ21	*Series 200/2000 Program Mode User Guide*
AQ22	*Series 200/2000 Program Mode Operator Guide*
AQ26	*Series 100 File Translator*
AQ27	*Series 200/2000 File Translator*
AQ28	*Library Maintenance Manual*
AQ40	*System 3 Conversion Guide*
AQ49	*Network Control Terminal Operation Manual*
AQ50	*Terminal Operations Manual*
AQ52	*Program Checkout Facility Manual*
AQ53	*Communications Processing Facility Manual*
AQ55	*TDS/64 Standard Processor Site Manual*
AQ56	*TDS/64 User Guide*
AQ57	*Standard Processor Programmer Reference Manual*
AQ59	*Unit Record Devices User Guide*
AQ63	*COBOL User Guide*
AQ60	*Interactive Operation Facility*
AQ64	*COBOL Language Reference Manual*
AQ65	*FORTRAN Language Reference Manual*
AQ66	*FORTRAN User Guide*
AQ67	*FORTRAN Mathematical Library*
AQ68	*RPG Language Reference Manual*
AQ69	*RPG User Guide*
AQ72	*Series 200/2000 COBOL to Level 64 COBOL Translator*
AQ73	*IBM COBOL Translator*
AQ82	*BFAS User Guide*
AQ83	*HFAS User Guide*
AQ84	*UFAS User Guide*
AQ85	*Sort/Merge Manual*
AQ86	*Catalog Management Manual*
AQ87	*Library Maintenance User Guide*
AQ88	*I-D-S/II User Guide, Volume 1*
AQ89	*I-D-S/II User Guide, Volume 2*
AQ90	*COBOL Reference Card*
AQ92	*Operator's Reference Card*
AQ93	*RPG Reference Card*
AQ94	*FORTRAN Reference Card*

CONTENTS

SECTION I

SECTION II

ILLUSTRATIONS

TABLES

1•Job Management

JOB MANAGEMENT

The execution of work input to the system by the user, is controlled by GCOS (General Comprehensive Operating Supervisor). A work package, which may consist of many separate jobs, is submitted to the system in the form of an input stream, usually via card reader. A job stream contains JCL (Job Control Language) statements which allows GCOS to handle each phase of a job through to successful execution.

In a multi-programming environment, jobs executing concurrently may compete for the same system resources. The information given to the operating system in the JCL job descriptions enables conflicts of this type to be resolved to optimize throughput by establishing an execution and scheduling hierarchy. JCL instructions and operator commands entered at run-time allow the user to make further optimizing interventions in the overall processing strategy.

Job structure

Each user job in an input stream is delimited by JCL $JOB. . . . . $ENDJOB statements, forming the major partition or enclosure of a complete job description. The system resource and utility requirements of each load module are defined by JCL statements and enclosed by $STEP. . . $ENDSTEP statements. Data in the input stream is separated from the input stream in a job enclosure by $INPUT. . . . . $ENDINPUT statements. The structure of an input stream therefore consists of three levels of enclosure (see Fig. 1-1). Their functions are to :

- make the job known to the system JOB enclosure
- describe the handling of each load module to the operating system ; STEP enclosure
- define data areas in a job stream ; INPUT enclosure.

Most of the JCL statements in a job description relate to a step enclosure, that is, they name a load module and define conditions and resources required for step execution. If a step involves the execution of a user written program, then the user must provide a complete JCL step description.

The execution of a system program, typically a compiler or dependent component program, does not require a detailed step description. It is sufficient to write only a single statement (for example $COBOL, $SORT) and supply the appropriate parameter information. At the time the job description is translated into internal format (see Translator below), the original single statement coded by the programmer is expanded to a full set of basic functions.

These expandable JCL statements, such as $COBOL, $SORT, are not considered as basic statements, although they are written in exactly the same way ; they are called Extended JCL. Thus JCL comprises of two types of statement :

- Basic JCL Statement, which requests a specific system function for example $STEP, $ASSIGN, $ENDSTEP
- Extended JCL Statement, which represents a set of basic JCL statements that themselves constitute one complete step description, for example $COBOL, $PRINT, $SORT.

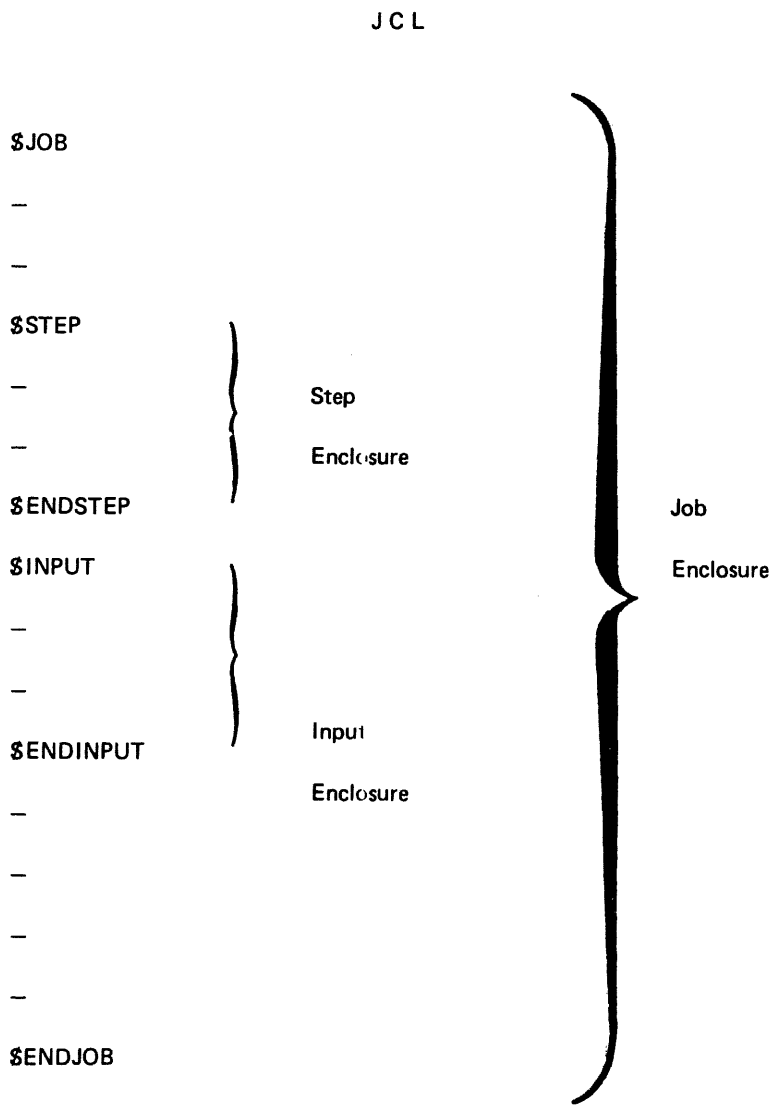


Figure 1-1. Job Description

The JCL Language Reference Manual describes in full the purpose of each Basic JCL statement and contains a list of Extended JCL statements together with the name of the associated manual.

A JOB RUN

The whole life cycle of a job, as opposed to the period of this cycle during which the job is actually being executed, is called a job run. During its run a job is uniquely identified by a number assigned to the job by the operating system, called a Run Occurrence Number (known as a «ron»). The Run Occurrence Number is a number of up to three digits, which is always preceded by the letter X. Sample rons are X23, X56, X112. Steps of a job are referenced by their physical order in the job description, that is, the step number. This number is printed in the banner page of the Job Occurrence Report produced by the system for identification purposes.

Stages of a Job Run

From the time the job is submitted, until it is output to the user-defined media, it is known to the system in different states or stages. Firstly the job is introduced to GCOS ; that is, a request is sent to GCOS to execute a given job in a job stream. It is at this time that a ron is assigned to a job (and the job is «known» to the system). The job description is then translated into an internal format suitable for execution. During translation the JCL is checked for syntax errors (see Section VII). From the time the job is submitted to the system at an input device until the JCL is translated, the job is said to be in an INTRODUCED state. The Input Reader supervises this stage of a job run.

The translated JCL is used by a system component known as the Job Scheduler, to establish an executing hierarchy for jobs currently «known» to the system. The Job Scheduler selects a job from the IN SCHEDULING queue that is next to be executed. The selected job enters the executing state. The EXECUTING state is followed by the OUTPUT state during which the Output Writer supervises the production of program output onto the user defined media. Two other job states exist, namely the HOLD state and the SUSPENDED state. Jobs in the HOLD state, although at the same stage as jobs IN SCHEDULING, are removed from the scheduler queue and are ignored by the Job Scheduler.

A job may be put in the HOLD state by use of the HOLD parameter in a $JOB statement, or by an operator command, HOLD JOB (HJ).

A job in the HOLD state can be put IN SCHEDULING by an operator command RELEASE JOB (RJ), or a RELEASE statement in another job. The HOLD option can be used to delay the execution of a job until a certain event has occurred for example, the termination of another job, or an operator decision.

EXECUTING jobs can be temporarily placed in the SUSPENDED state, by the operator HJ command. This can prove useful for the quick re-scheduling and execution of an urgent job, or to handle recurrent resource conflicts between two jobs, or to alleviate a situation of excessive system load.

The resources allocated to the a job can be made available by using the operator command HJ ron ENDSTEP ; the job is then suspended at the end of the current step.

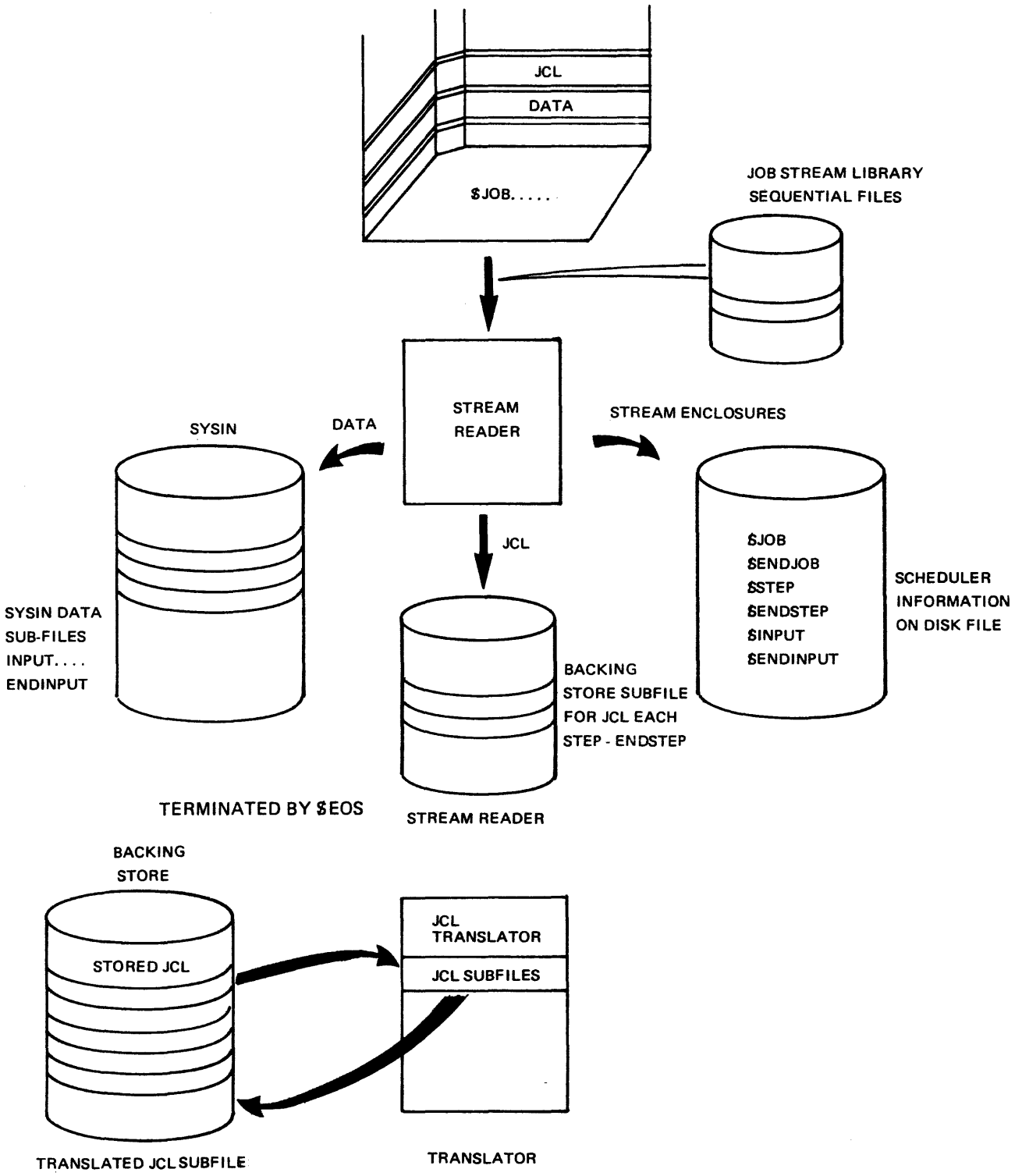


Figure 1-2. Stream Reader and JCL Translator

Input Reader

The input reader is initialized by an operator command or more usually by switching the card reader to READY. The Input Reader controls the introduction of a job stream into the system by reading, analyzing, translating and storing data and JCL in an internal format separately for later job scheduling and execution. The input reader consists of two separate processes :

- Stream Reader
- JCL Translator

STREAM READER

When the Stream Reader is requested to read an input stream (see Figure 1-2) it assigns the input device and reads the first record. For each job, the JCL statements in the input stream are stored, in source form, on a file in backing store. For each input enclosure the data is stored in a system file known as SYS.IN. In this way, the stream reader separates input enclosures from JCL job descriptions, yielding one JCL file in backing store and 'n' SYSIN sub-files for 'n' input enclosures. When the $ENDJOB statement is encountered, an entry is made in a queue accessible to the Translator. The Stream Reader then repeats its activity for the next job, until the end of the input stream is reached. At this point an end of input stream notification is sent to the Translator and the start of translation is initiated. The end of an input stream on cards is indicated by a $EOS statement, followed by at least one blank card. Normally the $EOS statement is an operator function rather than a programmer function.

JCL TRANSLATOR

When the Translator is notified by the Stream Reader that the end of an input stream has been reached, it starts to translate the JCL statements into a format suitable for execution by the Command Interpreter. For each JCL file produced by the Stream Reader's activity, the Translator opens a file into which the translated JCL statements of a job description are written. In addition, a listing of all JCL and Translator errors is stored for printing later in the Job Occurrence Report by the Output Writer.

Known Jobs Limit

KNOWN JOBS LIMIT

From the moment a job enters the system (INTRODUCED STATE) to the moment it is about to leave the system (OUTPUT STATE), the job is said to be «known» to the system. The maximum number of jobs known to the system is set at system configuration time and is 200. If this limit is exceeded the Stream Reader stops further jobs being introduced to the system.

Job Scheduling

When the translation is complete, provided that no errors have been detected, the job Scheduler determines when a job can be started.

JOB SCHEDULER

The Job Scheduler controls the system load at job level, and provides various facilities to the user for organization and planning of the workload in order to optimize system throughput. In addition, it provides the user with a set of commands (JCL and OPERATOR) to control the job flow.

There are pratical reasons for limiting the number of jobs simultaneously executing. Firstly, in the case where two executing steps would require the same system resources (for example each step needs three tape devices, and only three devices are available), the user can ensure that such jobs do not execute simultaneously. Secondly, if the number of programs executing was allowed to increase without restriction, a greater proportion of memory would be occupied by segments which must be resident throughout execution (e.g. process control structures). This would mean that virtual memory management must be increasingly involved with transferring the non-resident parts of the programs to and from backing store, thus degrading the overall system performance (see Section IV, $SIZE). For these reasons GCOS provides a set of functions to control the number and type of jobs simultaneously executing as well as the order in which they are executed.

At any time, the total number of executing jobs is limited by the System Multi-programming Level (the actual number of jobs is known as the System Load). Each job has an associated scheduling priority, indicating its urgency or priority of scheduling relative to other jobs known to the system. Jobs IN SCHEDULING are stored in a queue (the scheduler queue), which they enter at the end of JCL translation. The scheduler queue is put in sequence according to the «scheduling priority», assessed either from user specified values or by default value associated with the «job class» (see SCHEDULING AND JOB CLASSIFICATION below).

COMMAND INTERPRETER

When the scheduler selects a job from the queue for execution it notifies the Command Interpreter, which then accesses the job's translated JCL from backing store. For each JCL statement, the Command Interpreter's function is to initiate the appropriate Operating System action. Before execution of a step can begin, its requests in the JCL for resources must be analyzed and fulfilled, or the necessary queueing for resources performed. When all resources are available a user program or a utility can be executed.

Step Termination

At step termination, allocated resources are released and control again passes to the Command Interpreter. If the job contains another step, the process is repeated. Figure 1-3 illustrates the processing of a one-step job. Abnormal step termination may be caused by certain errors or abnormal conditions during step initiation or translation.

Job Termination

When the Command Interpreter encounters an $ENDJOB statement, job termination procedures are invoked to delete temporary files and subfiles and to collect final accounting information ; Output Writer activity for the job is ready to be initiated.

JOB DESCRIPTIONS

Introduction

The $JOB statement is the first statement of a job description, and provides identification and administrative information, such as job name, user name, project name and accounting identification. For example :

```
$JOB  RTSJOB,  USER = SSF,
        PROJECT= SSFT, BILLING = GPO ;
```

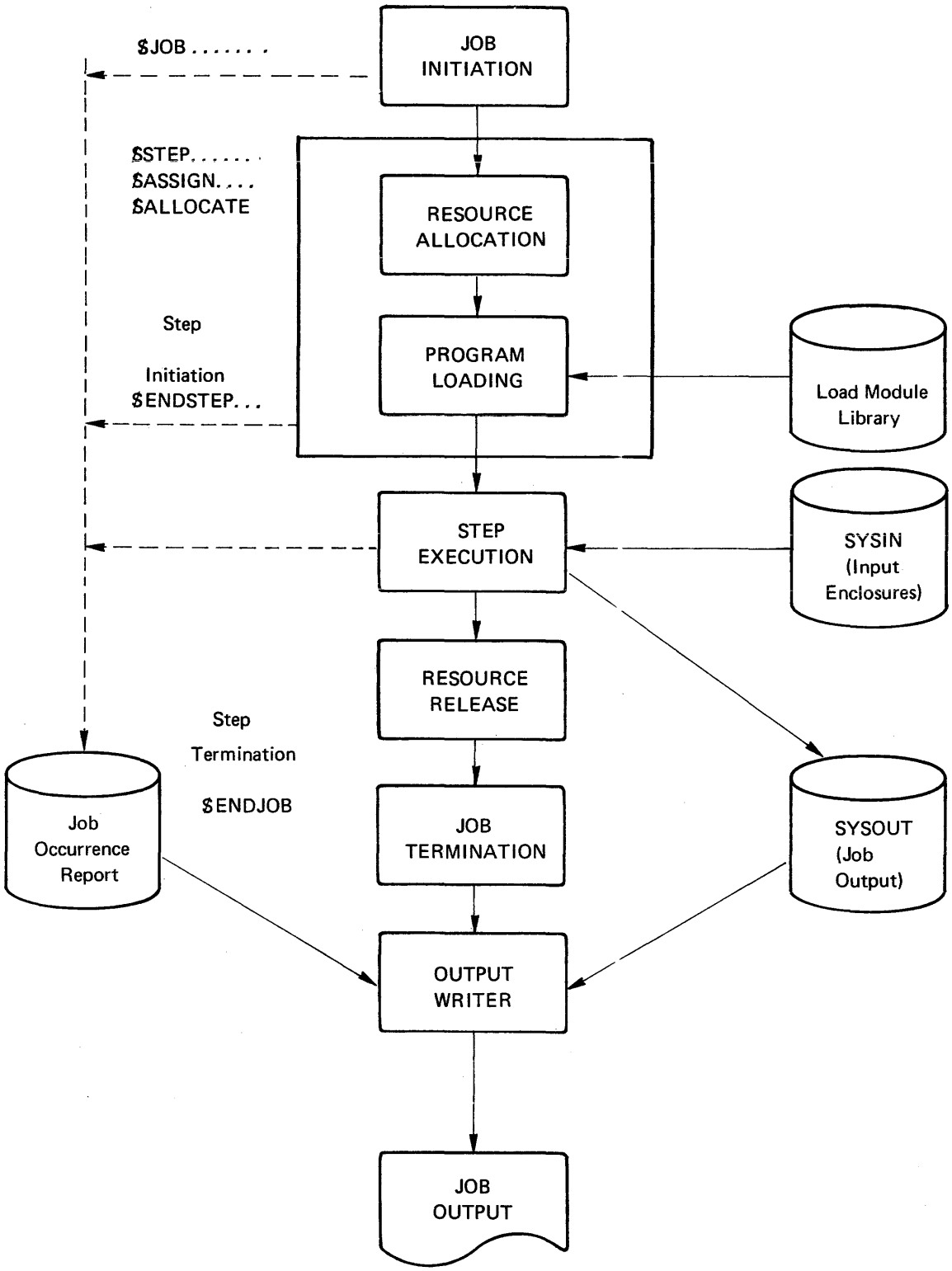


Figure 1-3. Step Execution for a Single Step Job

The job description is always terminated by $ENDJOB, which has no parameters. If it is available, the site catalog contains a list of users and associated project and billing information ; normally only the USER parameter need be used.

$STEP description

The purpose of the $STEP statement is to define to the system all the resources and facilities needed to execute the load module enclosed within the $STEP. . . . $ENDSTEP statements.

File assignment

The assignment of files to a step is performed using $ASSIGN statements. The $ASSIGN statement relates the internal-file-name (ifn), which is the name by which the file is known to the program, to the external-file (efn), which is the name by which the file is physically identified by the system. The second function of the statement is to allocate to the step, the resources (device, volume) that are associated with the file. The file assigned may be a permanent file (cataloged or uncataloged) or a temporary file which exists for the duration of the step only (or can be passed, see Section III) providing work space for the step. The most common uses of $ASSIGN are :

- For an input enclosure :

  ASSIGN ifn, *input-enclosure-name ;

- For a permanent cataloged file:

  ASSIGN ifn, efn ;

- For a permanent, uncataloged file on a resident disk:

  ASSIGN ifn, efn, RESIDENT ;

- For a permanent, uncataloged file on a non-resident disk:

  ASSIGN ifn, efn, DEVCLASS = device-class, MEDIA= volume-name ;

- For a temporary file :

  ASSIGN ifn, efn, TEMPRY ;

See Section III, File Assignment.

SCHEDULING AND EXECUTION

GCOS gives the user a considerable amount of control over the order in which jobs will be executed once they are known to the system. This control capability enables work loads to be planned to produce an efficient processing strategy. The user applies this control by use of a PRIORITY parameter in the $JOB (or $RUN) statements, which establishes an execution hierarchy for each job relative to all others in scheduling and those of the Operating System. The second parameter is job class, which establishes a job as of a certain category, and the system schedules the job according to the class mix of jobs already known to the system. The following paragraphs describe the use of these parameters to produce the desired job throughput.

Scheduling Priority

A job is assigned a scheduling priority, which indicates its urgency relative to other jobs. Scheduling priorities range from 0 (highest priority) to 7 (lowest priority). The scheduler queues the jobs according to the priority number and, for jobs of equal priority, on a first-in, first-out basis. Jobs are selected for execution according to their order in the queue, and the availability of the job class, one after another until the system multiprogramming limit is reached (see below). There is a limit applied also to the number of jobs of the same class which may execute simultaneously, (i.e., job class multiprogramming limit). This is discussed later. At this point, the remaining jobs are left in the IN SCHEDULING state until the termination of a job frees a

multiprogramming slot (see below).

The scheduling priority of a user job can be specified using the PRIORITY parameter in the $JOB or in the $RUN statement (see JCL Reference Manual). If the PRIORITY parameter is not present then a default value is assigned to a job according to its job class (see below).

NOTE : The operator MJ command (MODIFY JOB) can override any priority given by the user, or applying by default, for the duration of the job.

ENTERING THE SCHEDULER QUEUE

For a given scheduling priority, a job enters the scheduler queue just behind the last job with the same priority. An example is shown in Figure 1-4 (see the paragraph entitled Scheduling).

JOB ORDER in the SCHEDULER QUEUE

Jobs are placed in the scheduler queue in the following order.

- Forced job (i.e. urgent job).
  The operator command FJ positions a job at the head of the queue, regardless of its class or its status (e.g. even if the HOLD parameter is present in the $JOB statement). A forced job executes immediately, unless it has been introduced but not yet translated. Only one forced job can be executing in the system at any time.
- Suspended jobs
  Executing jobs can be put temporarily in the SUSPENDED state by the operator command HJ ron. This can prove useful for the quick rescheduling and execution of an urgent job or to handle recurrent resource conflicts between two jobs. The SUSPENDED job can be released by the operator RJ ron command, which then executed as soon as resources are available.

  A multiprogramming slot occupied by the SUSPENDED job is in general, not released. The slot can be released at the end of the currently executing step by use of HJ ENDSTEP (ron).
- Service Jobs
  Certain system components (e.g. JCL Translator, BTNS) are executed in the same way as ordinary user jobs with specific classes and priorities. These jobs are known as «service jobs» and are not subject to the scheduling rules but are started when required (provided the multiprogramming limit of their class has not been exceeded), and are never taken into account when calculating the system load.
- User jobs (not forced nor held)
  Jobs are in the scheduler queue according to their job scheduling priority and then to their Run Occurrence Number (i.e. jobs with the same priority are scheduled in the order of their introduction to the system). The scheduling priority can have a value between 0 and 7 inclusive, where 0 has the highest urgency and 7 the lowest.
- Held jobs
  A job in the scheduler queue may be put in the HOLD state by use of the HOLD parameter in $JOB statement, or by an operator command HJ (ron). A job in the HOLD state is ignored by the scheduler. Such a job can be released either by operator command RJ ron, or a RELEASE job-name, in a job other than the one in the HOLD state. The job is then returned to the scheduler queue at the end of the jobs of the same class already in the queue.

Activities

Once a job is selected from the scheduler queue for execution, it is known as an «activity», and as such occupies a multiprogramming slot. There is a maximum number of multiprogramming slots available to all jobs (user and service), set at system configuration time. This limit specifies the maximum number of activities (i.e. the total number of jobs), that can be executing or suspended at any time. If this limit is reached, every job in the scheduler queue waits in the IN SCHEDULING state until a slot is freed. Unlike the Maximum System Load (see below), the maximum number of activities is considered to be invariant and is not used by the operator for controlling the execution of jobs within the system.

Maximum System Load

The Maximum System Load (or System Multiprogramming Limit) is the maximum number of batch jobs i.e., in the EXECUTING or SUSPENDED states. In other words, it is the total number of multiprogramming slots available to user jobs. The Maximum System Load value is user configured and can never exceed the maximum of 30. It can be changed dynamically by the operator, depending on the current work load. The operator can control job throughput by use of this value.

Job Class

The programmer can influence the order in which jobs are executed through the assignment in the $JOB statement of a job class. A user job is assigned to one of sixteen job classes, denoted by a letter from A to P. Service jobs are assigned classes within the range Q to Z.

Class P is the default class for user jobs.

Job classes are a means of organizing the user's workload, so as to optimize the throughput of jobs in a multiprogramming environment. For example :

- Some jobs need to be executed serially (one after another) because of the installation dependancies or constraints.
- Some jobs must be executed at given periods of time (night shifts, etc...).
- To take the best advantage of multiprogramming, some job mixes (running simultaneously) are more favorable then others ; for example, central processor bound and I/O bound jobs, instead of groups of all central processor bound jobs.
- If several jobs use the same resources (for example tape units or other peripherals) to avoid conflicts, they should not be scheduled at the same time.

Thus, each user job entering the system belongs to a certain class associated with a series of attributes, through which the job mixes and workload planning can be tuned.

A class multiprogramming limit (or «maximum class load») controls the number of jobs which can be in EXECUTION or SUSPENDED state within a given class. The total number of jobs that are executing and/or suspended at a given time within a particular job class is known as the class load for that class. Associated with each job class are the default job scheduling and step execution priorities, as well as the default multiprogramming limit. These are shown in Table 1-1.

In addition, an operator can suspend or reactivate a job class. This provides further capabilities to select job for processing and manage the serial execution of jobs.

CLASS ATTRIBUTES

Three attributes are attached to each class and used to optimize the user's work-load. These attributes are never erased by a system shutdown or system crash, and are unaltered unless the permanent backing store is destroyed and are fixed at configuration time. They can be modified, though, by the operator, via the MC command.

The attributes are :

- Class multiprogramming limit :
  This is the maximum number of jobs in execution or suspended within a given class, including those of service jobs. The operator can change the value of the class multiprogramming limit to suit the current work load.
- Scheduling priority :
  A default value is automatically allocated to a job, if not specified in the $JOB or $RUN statements. The operator can also modify the scheduling priority using the MJ command.
- EXECUTION PRIORITY (DISPATCHING PRIORITY)
  A default value is allocated to all steps of all jobs in a given class for which the execution priority (XPRTY) is not specified in $STEP statement. The operator can modify the execution priority by use of MJ command (by default).

EXAMPLE OF WORKLOAD ORGANIZATION

The following example shows a possible organization of a user workload with the partition of jobs into classes according to their type (I/O bound etc.). It is assumed that scheduling and execution priorities are those defined as the class attributes, see Table 1-1.

TABLE 1-1. CLASS AVAILABILITY AND USE (DEFAULT VALUES)

JOB CLASS	DEFAULT SCHEDULING PRIORITY	DEFAULT DISPATCHING (EXECUTION) PRIORITY	DEFAULT MULTIPROGRAMMING LIMIT
A (NATIVE JOBS)	7	9	1
B (NATIVE JOBS)	7	9	1
C (NATIVE JOBS)	7	9	1
D - PROGRAM MODE (PM 100, 200)	1	5	1
E (NATIVE JOBS)	2	4	1
F (NATIVE JOBS)	3	7	1
G (NATIVE JOBS)	4	9	1
H - (COMMUNICATIONS)	6	1	1
I - (NATIVE JOB S)	7	9	1
J - (TDS/64 STANDARD PROCESSOR)	6	1	1
K (NATIVE JOBS)	7	9	1
L (NATIVE JOBS)	7	9	1
M (NATIVE JOBS)	7	9	1
N (NATIVE JOBS)	7	9	1
O (NATIVE JOBS)	7	9	1
P - DEFAULT CLASS NATIVE JOBS	7	9	5
SERVICE JOBS			
Q - IOF	7	4	10
R - READER	7	2	
S - BTNS	7	0	1
U - FTU	1	2	6
W WRITER	1	2	8
X JTRA	0	3	1

NOTE : The values stated in this table ase those of the reference system disk, but they can be modified using CONFIG, see System Management Guide.

Assume the following system configuration :

- Four tape drives
- Multiprogramming limit = four jobs

and the following work jobs :

Job Type	Resources and constraints	No. of jobs	Assigned class
I/O	3 tapes/job step	3	E
I/O	1 tape/job step	5	F
CPU	2 jobs max in memory	5	G
Communication (BTNS + User job)	Assume user job running all day	1	H
Background	2 tapes/job step	5	K
Tests	unknown	10	P

The class configuration intended to optimize the global throughput according to the different constraints of coexistence between jobs may be the following :

Class	Max Class Load within Constraints	Scheduling priority	Execution priority
E	1	2	4
F	1	3	4 (a)
G	2 (a)	4	9
H	1	6	1
K	2 (a)	7	9
P	5	6 (a)	9

a. Modifiable by operator MC command. Then valid until again modified.

All referenced classes are «started».

Suggested Work Schema :

- Communication job (Class H) is scheduled at the start of the day. The execution priority is higher than that of the Output Writer and Stream Reader in order to get an acceptable response time.
- Class E jobs are exclusive, in order not to create conflicts between tape drives.
- From the point of view of tape drive requirements, class F jobs are complementary to jobs of class E. They may be executed alongside class E jobs.
- Class G jobs are scheduled with previously referenced jobs, but only one at a time unless one of the above classes becomes empty. In order not to monopolize the common resources of virtual memory, the maximum number of jobs in parallel (in this class) is two.
- Class P jobs are scheduled when one of the previous classes is emptied.

– If there is enough room, the background jobs of class K are executed to avoid conflicts on the tape drives, with a maximum of two at the same time.

Job Selection from Scheduler Queue

Whenever the current system load is less than the maximum system load and the scheduler queue contains at least one job whose class load is currently less than the maximum class load for that class, a job will be selected from the scheduler queue for execution. The selection is based on the current order within the scheduler queue, the job classes of the jobs already executing, and the classes of the members of the queue.

– Figure 1-4 illustrates the selection of jobs from a scheduler queue. Note that after job PAUL terminates, job PETER cannot be selected because one job of class E is already executing. Therefore job JOHN is started, making it the second job in class A executing.

Step Execution

The scheduler notifies the Command Interpreter when a job has been selected for execution. The Command Interpreter reads and initiates the appropriate system action requested by JCL statements. Each time a $STEP statement is encountered, the Command Interpreter calls a «step initiation» routine which reads all of the statements of the step description and allocates the appropriate system resources, (or the step is queud until all necessary resources are available. The load module is loaded from the load module library and step execution begins when the $ENDSTEP for that step is encountered.

Execution Priority (Dispatching Priority)

Once a job is scheduled and initiated, its various steps are executed. The DISPATCHING PRIORITY (DPR) is used to control the amount of CPU time a particular step can obtain relative to other steps currently competing with it for CPU time. The DPR is represented by n, where $0 \leqslant n \leqslant 9$ (0 = highest priority) and can be defined by the XPRTY option of $STEP, or the operator MDPR command. If the priority option is not present, the default value is determined by the job class (see table 1-1).

NOTE : The dispatching priority of a step can be modified by using the operator MJ command. The MDPR command is used to control what specific CPU allocation is available to steps which are executing with a certain dispatching priority, see below.

Parameters available to the operator in the MDPR command enable further optimization of overall throughput to be achieved by «controlling» the dispatching priority. These parameters :

– enable equal amounts of CPU time to be allocated to steps of equal DPR concurrently executing (MDPR dpr SLICE command).
– limit the amount of CPU time that steps in a certain range of dispatching priorities can use, (MPDR dpr : dpr n)
– declare as optimized two consecutive DPRs to adjust dynamically the relative priorities in order that I/O-bounded steps always have a higher priority over CPU-bounded steps.

Refer to the System Operator Guide for further details.

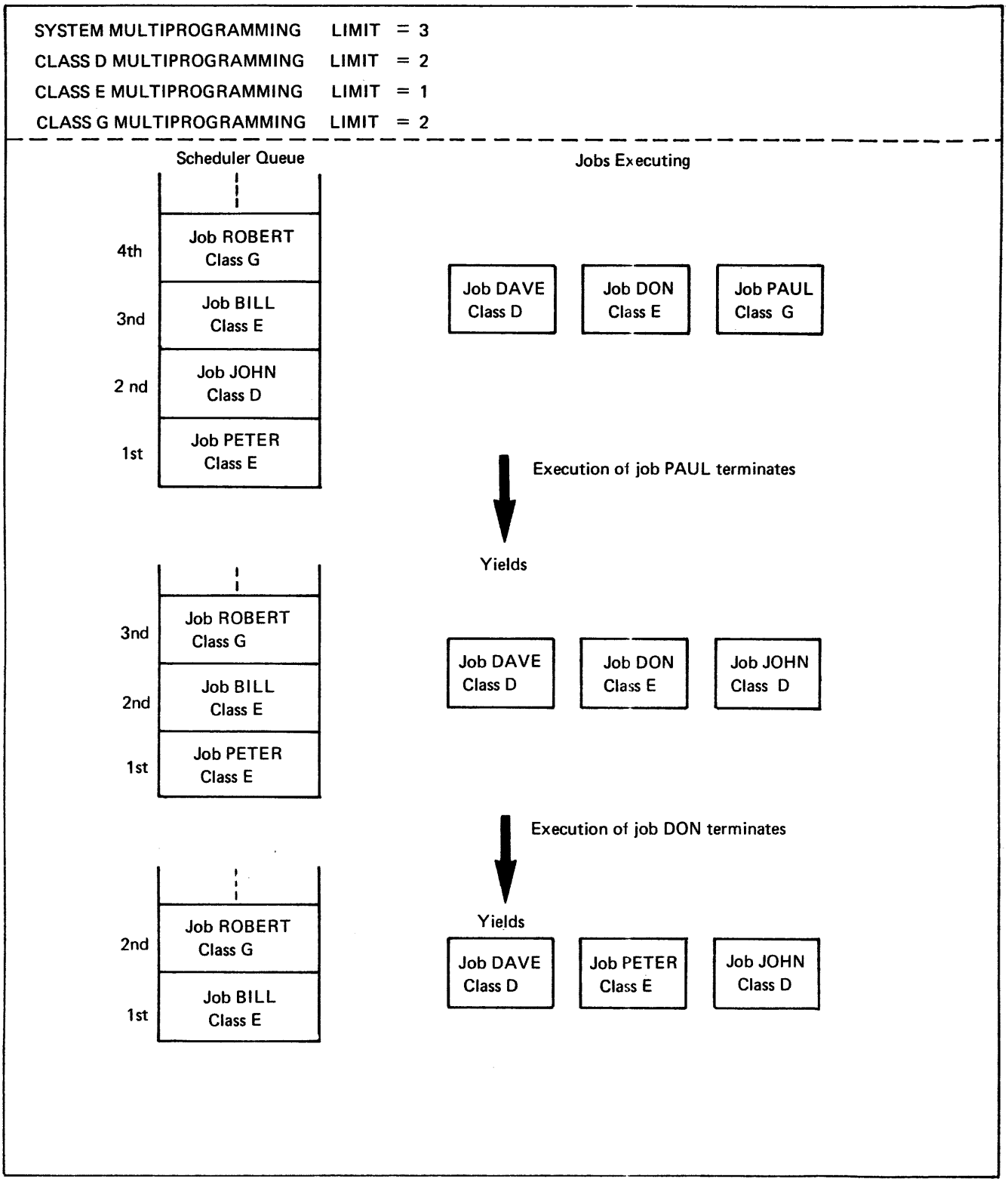


Figure 1-4. Selection of Jobs for Execution

COMMUNICATIONS JCL

The communications session applies to the execution of Message Control System (MCS) COBOL application programs or any of the Virtual Communications Access Method (VCAM) subsystems.
The three types of VCAM subsystems are,
- TDS (transaction driven system)
- IOF (interactive operation facility

The JCL for these software modules is the same as for any other job but with the following additions, namely,

- $QASSIGN
- NDL commands (network description language)
- QMAINT commands (queue maintenance).

Specific to MCS COBOL or MAN (Message Access Method) application programs is the $QASSIGN statement for relating physical queues declared in the network to their symbolic queues referred to in the programs. Queues are maintained using the QMAINT utility which is driven using specific commands.

The network to be generated for the communications session is defined by NDL commands within the $INPUT enclosure for the CNC utility (communications network configurator).

Overview IOF

The IOF facility enables a user to be connected to GCOS through a terminal. From the terminal the user is able, using Library Maintenance and thetext Editor, to build and modify job descriptions and data files ; and then ask GCOS to execute such jobs. These jobs are introduced to GCOS as if they had been read by the card reader, using the same commands as a job entered on cards for instance. An IOF user who requires work to be done by GCOS, specifies his requirements at the terminal using a subset of JCL, see the IOF Manual for full details.

TDS Overview

The user provides COBOL programs in the form of Transaction Processing Routines (TPR). Each TPR can receive a message to process data and generate a response.

At TDS generation, the user links his TPRs with the TDS executive in order to tailor a specific version of TDS for his requirements. The version so generated is executed as a job step with BTNS and the network generated by the CNC utility.
For debugging purposes, the TDS version can be executed as a job step with a simulated network of batch entry programs functioning as terminals.

For further details on TDS, see the appropriate manuals listed,
- TDS Programmer Reference Manual
- TDS User Guide
- TDS Site Manual.

2 • Unit record Input Output

UNIT RECORD INPUT/OUTPUT

In a Level 64 environment, much use will be made of a card reader for input purposes (of data, source programs, job descriptions), and of a line printer or card punch for output purposes (of reports, invoices, etc.). These devices, known as Unit Record Devices, operate at slow speeds compared with the speed of data transfer between main and secondary storage, and the speed at which central processor operations are performed. To separate the input/output function from that of job translation and execution, and thus prevent the build-up of large queues for these devices, GCOS employs a spooling system. This facility makes use of intermediate storage on disk or tape files, and operates independently from the execution of user jobs. Note that it is possible for the programmer to bypass the intermediate storage and make the program access a device directly.

HANDLING CARD INPUT

There are three ways in which card input can be handled :

- by storing it in an intermediate system file ; that is, spooling it onto the file and thus relieving the user of the responsibility for device assignment. This facility is known as «standard SYSIN».
- by storing it in a sequential input file using the utility $CREATE, or by storing it as a member (subfile) of a source library by using the utility $LIBMAINT. This facility is known as «permanent SYSIN»
- by assigning the card reader directly to the program.

Each of these methods is describe below, and shown in figure 2-1.

Input Enclosures - $INPUT and $ENDINPUT

For standard and permanent SYSIN, the card data is contained in an input enclosure. An input enclosure is defined by the Basic JCL statements $INPUT and $ENDINPUT. The $sign is mandatory with these statements.

The ENDCHAR and CONTCHAR parameters to the $INPUT statement allow the user to select characters from cards, and to concatenate records together.

ENDCHAR

When the ENDCHAR parameter is used, consecutive input cards are concatenated in the same SYSIN record until the character specified in the ENDCHAR parameter appears as the last non-blank character on a card.
For example,

ENDCHAR = /

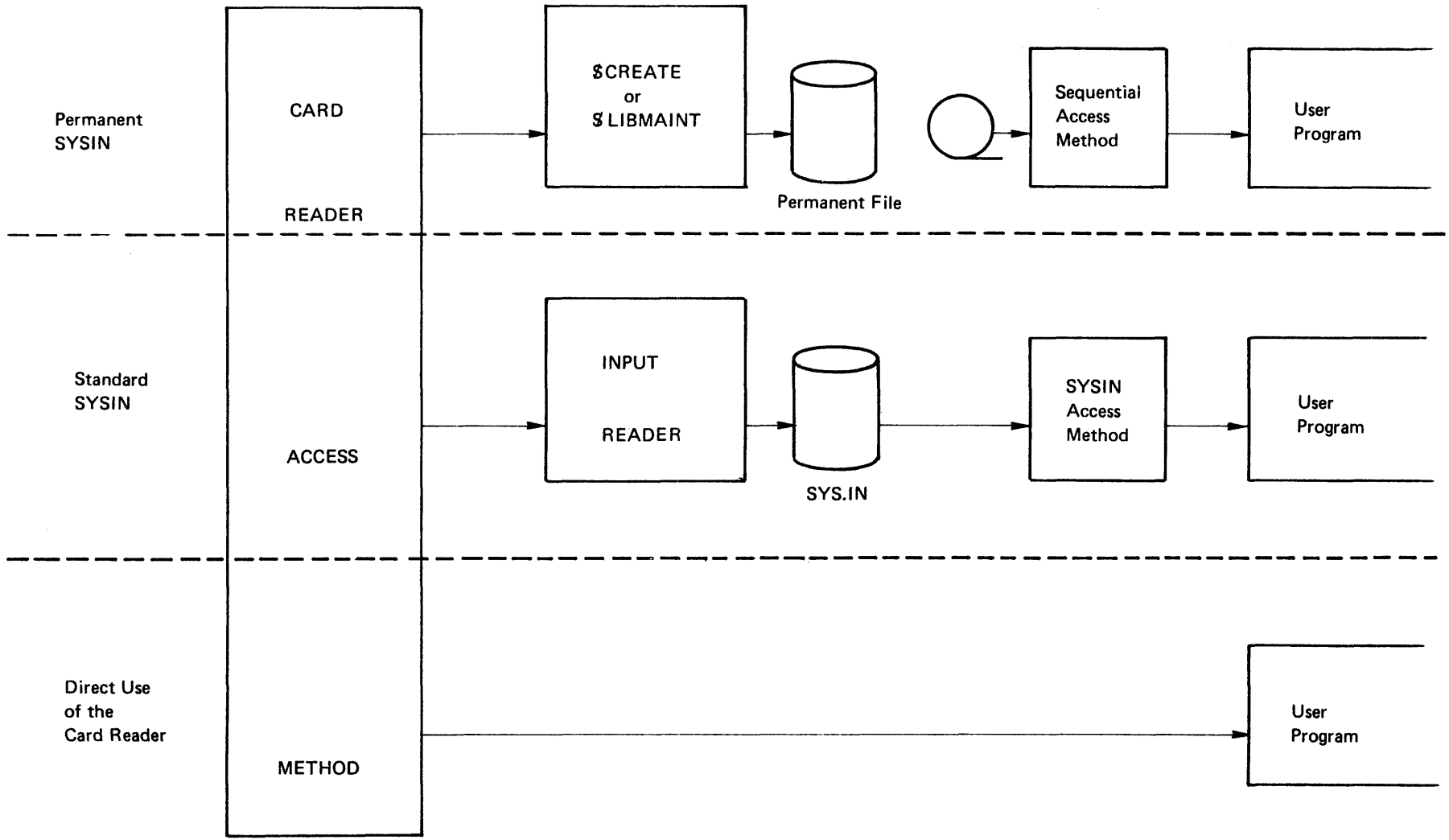


Figure 2-1. Methods of Handling Card Input

```
input cards :  ABC/DE  . . . .  80      80 characters
               FGHIJ                    80 characters

resulting
SYSIN record : ABC/DE  . . . . FGHIJ    85 characters
```

CONTCHAR

When the character specified in the CONTCHAR parameter appears as the last non-blank character of a card :

- only the characters preceeding it in the current card are copied in the SYSIN record ;
- the next card will be concatenated to the current one in the same input record.

For example,

CONTCHAR = +

```
input cards    :  ABC/DE +. . . . .  80    80 characters
                  FGHIJ                    80 characters

resulting SYSIN
record :          ABC/DEFGHIJ . . .        86 characters
```

ENDCHAR AND CONTCHAR USED TOGETHER

The ENDCHAR and CONTCHAR parameters can be used together. For example,

ENDCHAR = / CONTCHAR = +

```
input cards :  ABCDE + . . . . . .  80    80 characters
               FGHIKJL/ . . . . . .       80 characters

resulting SYSIN
record      :  ABCDEFGHIJKL               12 characters
```

If the card deck to be read does not have the $ character in column one of any of the cards, the $ENDINPUT statement need not be written, and the END parameter of the $INPUT statement can be used. For example,

```
$INPUT INDECK, END = DOLLAR ;

card-deck

$STEP . . . ;   NOTE : The $ sign is mandatory in this case.
```

If the card deck to be read contains a $ENDINPUT statement, the match facility can be used. For example,

```
$INPUT INDECK, END = MATCH ;
$INPUT CARD ;
other cards to be read          } DATA
$ENDINPUT CARD ;
$ENDINPUT INDECK ;
```

Because the Input Reader always considers a $JOB command to be the start of a new job enclosure, a $JOB command cannot be read as data in an input enclosure.

NOTE : If the ENDCHAR is the blank character, all the characters on the input card up to the last non-blank character are transferred to the sysin file. This can be used to save space on the SYS.IN system library file for standard SYSIN. In this case, the last column of the card must always be blank, otherwise this card will be concatenated to the next one in the same input record.

An error in the $INPUT statement will cause the job to abort after JCL translation, and this will also happen if two input enclosures in the same job stream have the same name.

The Use of Standard SYSIN

With the standard SYSIN, the data from the input enclosure is stored as a subfile of an intermediate system file known as SYS.IN. An input enclosure is associated with its processing program by means of a $ASSIGN statement of the form :

$ASSIGN internal-file-name, *input-enclosure-name ;

Other parameters to $ASSIGN are ignored, but if present, each one causes a warning message to be produced on the JOR.
If the specified input-enclosure-name does not exist in the job stream, the job will abort at JCL translation time.

Within the COBOL program, a file-description must be supplied for each standard SYSIN used, and this file-description must specify that the file is a standard SYSIN file. This is done in the ASSIGN clause :

ASSIGN TO internal-file-name-SYSIN

For example, the following program reads a card deck :

JCL

```
$JOB USER =JONES,PROJECT=P1 ;
STEP PROGA . . . ;
ASSIGN CARDFILE, *INDECK ;
ENDSTEP ;
$INPUT INDECK ;
        card deck to be read
$ENDINPUT ;
$ENDJOB ;
```

COBOL Program :

```
SELECT CARD.
ASSIGN TO CARDFILE-SYSIN.
FD CARD.
BLOCK CONTAINS 6 RECORDS.
RECORD CONTAINS 80 CHARACTERS.
LABEL RECORD IS STANDARD.
. . .
OPEN INPUT CARD.
READ CARD RECORD INTO ident1.
. . .
CLOSE CARD.
```

Permanent SYSIN

With permanent SYSIN, the intermediate file which contains the card input may be :

- a permanent BFAS or UFAS sequential disk or tape file filled by using the $CREATE utility
- a subfile of a source library, filled by using the $LIBMAINT utility.

If the permanent SYSIN file is a un cataloged tape file, the file characteristics must be specified in the OUTDEF parameter of $CREATE. If it in cataloged, this information may be in ithe catalog entry. The recommended values are :

RECFORM = FB, RECSIZE = 80

$CREATE can be used in two ways :

1) when the input deck is an input enclosure. For example, to create a sequential tape file :

```
CREATE INFILE = *INDECK, OUTFILE = (CARD,DEVCLASS=MT/T9,
                                    MEDIA =TAPE03),
        OUTDEF = (RECFORM= FB, RECSIZE = 80, BLKSIZE = 2400) ;
$INPUT INDECK ;
   cards to be read
$ENDINPUT;
```

2) when the input deck is read directly from the card reader. For example, to fill an already preallocated sequential disk file :

```
CREATE INFILE = (CARD,DEVCLASS = CD/R/C80, MEDIA = INDECK),
        OUTFILE = (MY.FILE, DEVCLASS = MS/M400, MEDIA = DISK02) ;
```

When the step is initiated, the operator will be requested to mount the input deck INDECK onto the card reader.

The use of the card reader in direct mode with $CREATE is very convenient when a very large input deck is to be read.

When $CREATE is used with the card reader in direct, it is possible to read a card deck containing $JOB and $CKP cards. When $CKP cards are present and an incident occurs(an unrecoverable I/O error a system crash,) while the cards are being read, the processing can resume at the last checkpoint. Only the cards following the last checkpoint have to be reloaded onto the card reader.

$CREATE using the card reader in direct also allows you to read 51 column cards and marked cards. 51 column cards are 80 column cards truncated on the right, and marked cards are cards where marker strokes replace the punches. For marked cards, the DEVCLASS parameter must be

DEVCLASS = CD/R/MI/C80 for IBM mode marked cards, or

DEVCLASS = CD/R/MB/C80 for HIS mode marked cards.

HIS mode cards have the marks placed in odd columns, and IBM mode cards have the marks placed in even columns. Note that marked cards can also be punched (punches may occur in all the columns), but if the marks and punches occurring the same column only the punches will be read, and the marks will be ignored.

To read 51 column cards, the DEVCLASS parameter to $CREATE must be :

DEVCLASS = CD/R/C51 for punched cards, or
DEVCLASS = CD/R/MI/C51 for IBM mode marked cards, or
DEVCLASS = CD/R/MB/C51 for HIS mode marked cards.

The MOVE command of $LIBMAINT stores an input deck as a subfile (member) of a source library. For example, if the input deck has to be stored in subfile MEMB1 of the cataloged source library SOURCE1.SLLIB, the program would be:

```
LIBMAINT SL,
        LIB = SOURCE1. SLLIB,
        COMFILE = *CD ;
$INPUT CD ;
        MOVE COMFILE : MEMB1, TYPE = DATASSF ;

            input deck

// EOD
        $ENDINPUT ;
```

Using a permanent SYSIN File

A COBOL program uses a permanent SYSIN file in the same way as it uses any sequential file. A $ASSIGN statement must be present in the JCL to make the correspondence between the internal-file-name and the external-file-name.

An example of the use of a permanent SYSIN file is :

JCL :

```
STEP PROGA . . . ;
ASSIGN CARDFILE, CARD, DEVCLASS =MT/T9,
    MEDIA =TAPE03 ;
ENDSTEP ;
```

COBOL Program :

```
SELECT IN1.
ASSIGN TO CARDFILE-SYSIN.
FD IN1.
RECORD CONTAINS 80 CHARACTERS.
LABEL RECORD IS STANDARD.
```

NOTES : At OPEN time, COBOL checks the record size found in the file label. If it is greater than the record size declared in the program, a warning message is produced on the JOR, but processing continues.

At READ time, if the record being read is greater than the receiving area, the read data is right truncated. The processing can continue if the program specified a USE procedure for this file.

Direct Input

The direct use of the card reader is similar to processing a non-standard file.

At program level, a file-description for the card reader has to be defined.
At JCL level, a $ASSIGN statement has to make the correspondence between the internal-file-name used in the program and the card deck to be read.
The form of this statement is :

$ASSIGN internal-file-name, DEVCLASS = CD/R/C80, MEDIA = INDECK ;

The cards to be read are not part of the job stream, but are a separate deck.
The system will ask the operator to mount the deck at the initiation of the corresponding step. The last card of the deck must be a $EOS card, which is the end-of-file mark. At least one blank card has to be added after the $ EOS card so that a RECOVER does not occur at $ EOS reading.

The example below shows a COBOL program, PROGA, which reads a card deck, INDECK, with the card reader in direct.

JCL :

```
$JOB . . . ;
STEP PROGA . . . ;
ASSIGN CARDFILE, DEVCLASS = CD/R/C80, MEDIA = INDECK ;
ENDSTEP ;
$ENDJOB ;
```

COBOL program :

```
SELECT CARD.
ASSIGN TO CARDFILE-CARD-READER
ORGANIZATION IS LEVEL 64 SEQUENTIAL.
FD CARD.
BLOCK CONTAINS 1 RECORD.
RECORDS CONTAIN 80 CHARACTERS.
LABEL RECORD IS OMITTED ...
OPEN INPUT CARD.
. . .
READ
. . .
CLOSE CARD.
```

At PROGA step initiation, the operator will receive the message :

hh. mm CD 01 MOUNT INDECK FORXn

Input Data Types

With a permanent SYSIN file, data type SSF can be asked for regardless of the type of card to be read (BIN, HOL, punched or marked).

With $CREATE, the SSF header (8 bytes) must be taken into account when the file is allocated.

With an input enclosure, to get SSF all that has to be specified is TYPE = DATASSF in the $INPUT statement. For example,

```
CREATE INFILE = *INDECK,
        OUTFILE = (CARDFILE, DEVCLASS = MT/T9,
                        MEDIA = TAPE 03) ;
$INPUT INDECK, TYPE = DATASSF ;

    data cards

$ENDINPUT ;
```

Note that the format of a permanent SYSIN file is exactly the same as that of the standard SYSIN subfile.

With $LIBMAINT, the TYPE = SSF parameter to the MOVE command is mandatory. $LIBMAINT will always create a Library member in SSF. For further details of SSF and SARF, see the COBOL User Guide.

When the card reader is used in direct with $CREATE, the input file will always be in SARF format, regardless of what is specified in the OUTDEF parameter group.

Reading SSF Input

Standard or Permanent SYSIN files and subfiles allow the COBOL programmer to process the SSF header. When nothing is specified in the SELECT statement for the SYSIN file, the READ statement only delivers the data part of the record, regardless of the format of the file (SARF or SSF). If the input file is in SSF, COBOL skips the control records and suppresses the SSF header.

When WITH SSF is specified in the SELECT statement, COBOL suppresses the eight byte SSF header and skips the control records without checking what format the file is in. Because of this, the user must ensure that the file is actually in SSF, otherwise the first eight bytes of data will be lost, and any records which look like control records (bit 0 of byte 1 is equal to 0) will be skipped.

When WITH SARF is specified in the SELECT statement, the READ statement delivers the entire record to the COBOL program, including the SSF header, if it exists. Thus if WITH SARF is specified and the SYSIN file is in SSF, the programmer must include the eight byte header in the record-description.

The WITH SARF option is used to access the SSF header. For example,

```
JCL :                          COBOL Program :

STEP PROGA. . . ;              SELECT CARD
ASSIGN CARDFILE,*INDECK;       ASSIGN TO CARDFILE SYSIN.
ENDSTEP ;
$INPUT INDECK,                 FD CARD
   TYPE = DATASSF ;            RECORD CONTAINS 88 CHARACTERS
                               01 INREC.
                                  02 SSFHEAD PICX(8).
$ENDINPUT ;                       02 USERDATA PIC X (8 0).
```

When the card reader is in direct with $CREATE, it is not possible to access the SSF header. The READ statement always delivers the card image.

Additional Notes on Card Input

The ACCEPT COBOL verb inputs data from any input sequential file (input enclosures, permanent SYSIN files and the card reader in direct mode) which has to be assigned by a $ASSIGN statement to the COBOL reserved internal file H_RD. For more details, see the COBOL User Guide.

All the considerations on each input have assumed that the cards are punched in H 36 mode. There are facilities for reading cards punched in binary and H14 mode. For further details, see the JCL Reference Manual and the Unit Record Devices User Guide.

The $MAP control card can be used with input decks to select fields from the cards. The selected fields will be transferred to the COBOL program if the card reader is being used directly, or to the SYSIN file or subfile otherwise. For further details, see the JCL Reference Manual and the Unit Record Devices User Guide.

HANDLING OF PRINTED AND PUNCHED OUTPUT

The GCOS Output Facilities

In general, there is only a small number of printers and/or card punches associated with a machine. However, in a multiprogramming environment, all jobs will produce some output to be sent to a printer. Although this may only be the Job Occurrence Report (JOR), the majority of jobs will produce additional reports and some jobs will produce several reports concurrently. A user program can access a printer or punch directly ; however, to avoid the situation of having to assign a printer and/or a punch to each job while it is in execution, a spooling technique is available. Each report is assigned to a file on disk or tape and later printed or punched from that file by the system component known as the Output Writer.

When considering the spooling of output within GCOS, the user should distinguish between two separate stages :

1. when a file to be output is being created (i.e. written) by means of a file access method ;
2. when the contents of the file are being printed or punched by means of the Output Writer

A special access method, known as the SYSOUT Mechanism, is available for the writing of files to be output. The SYSOUT mechanism incorporates the editing requirements into the file as it is built ; its use is generally recommended as it saves time when the file is output later by the Output Writer (see SYSOUT Mechanism below).

Each output «file», if any, produced during the execution of each step is normally written by the system to a standard system output subfile, known as a standard SYSOUT subfile, which is generally printed at the end of the job ; the user may modify the standard output parameters (print belt, number of copies, etc.) by using a $OUTVAL statement within the job enclosure or a $SYSOUT statement in the corresponding step enclosure. Alternatively, the user can write output to a permanent file that is output by a $SYSOUT statement in the same step or by a $WRITER statement in the same or a later job. This facility has the advantage that the file, known as a permanent SYSOUT file, is not deleted after printing, as is the case with a standard SYSOUT subfile. The program can also access the printer or card punch directly by an assignment (using $ASSIGN) of the internal file name to the device itself. In general, this procedure is not recommended.

NOTE : The use of standard SYSOUT subfiles is the simplest and most common method of producing printed output. As will be shown below, the user need not be concerned with file assignment nor with special instructions to the system ; all that is required is either a JCL statement ($SYSOUT) in the relevant step enclosure or an indication in the (COBOL) program (–SYSOUT).

The main differences between standard and permanent SYSOUT files are as follows :

– A standard SYSOUT subfile is a member of a system file, called SYS.OUT, that is automatically assigned to the step. Once the information to be output is printed or punched, the relevant SYSOUT subfile is deleted from SYS.OUT. The SYSOUT mechanism is always used for the creation of a standard SYSOUT subfile and any editing requirements are incorporated into the file as it is written.

– A permanent SYSOUT file is a permanent sequential file on disk or tape (or a permanent source library member) and the assignment of the file is the responsibility of the user. In general, the contents of a permanent SYSOUT file are preserved after the execution of the job. The file can be created under UFAS, BFAS or HFAS and may be in SSF or SARF format ; any editing requirements in these circumstances will be incorporated when the file contents are printed or punched. However, the user can choose to use the SYSOUT mechanism (see below) to edit the file when it is being created.

SUMMARY OF FACILITIES

The following output facilities are available under GCOS :

1. use of the SYSOUT Mechanism and the Output Writer :
   a. for temporary subfiles (standard SYSOUT subfiles)
   b. for permanent files (edited permanent SYSOUT files) ;

2. use of the Output Writer for permanent files created under UFAS, BFAS or HFAS (unedited permanent SYSOUT files) ;

3. direct use of the output device (no intermediate file).

Refer to Figure 2-3 at the end of this Section for an illustration of the functions of the various output facilities.

SYSOUT MECHANISM

DESCRIPTION

As described above, when output spooling is used, program output to a unit record device takes place in two stages :

1. the creation of a SYSOUT file ;
2. the printing (or punching) of the file.

The use of the SYSOUT Mechanism at stage 1 to produce an «edited SYSOUT file» saves the Output Writer time at stage 2 and, in general, gives a net increase in throughput over both stages.

The SYSOUT Mechanism edits a SYSOUT file as it is created as follows :

– suppresses trailing blanks from each record produced by the program (e.g. by a COBOL WRITE verb) ;

– writes each record in the file in a format suitable for the output device ;

– formats the output page according to the user's requirements.

An edited SYSOUT file is said to be in «SYSOUT format».

USE

To use the SYSOUT Mechanism for the creation of a SYSOUT file or subfile ; the user can specify one of the following options :

– a $SYSOUT statement in the relevant step enclosure ;
– the suffix–SYSOUT in a SELECT clause for the appropriate. internal-file-name in a COBOL program ;
– the parameter SYSOUT in a $DEFINE statement in the relevant step enclosure.

The $SYSOUT Statement

The use of the $SYSOUT statement is the simplest way of requesting output from a program. If there is no assignment of the internal-file-name of the file to be printed, the SYSOUT Mechanism will create a standard SYSOUT subfile whose contents will be printed by the Output Writer and the file will be deleted. If the internal-file-name of the file to be printed is assigned to a permanent file, the SYSOUT Mechanism will create a permanent SYSOUT file and the Output Writer will print the file contents. file contents.

Examples :

```
1. STEP TSTA, . . . ;
      ASSIGN INP, *CRDS ;
      ASSIGN REF, MY.PFILE ;
      SYSOUT RESULTS ;
   ENDSTEP ;
```

In the above example, the data associated within the program with the internal-file-name RESULTS will be printed after the end of the job ; the default installation parameters (e.g. standard stationery, standard print density) will be used for the printing of the listing ; no permanent copy of the data will be kept.

```
2. STEP TSTB, . . . . ;
      ASSIGN INP, *CRDS ;
      ASSIGN REF, MY.PFILE ;
      ASSIGN RESULTS, MY.DATA ;
      SYSOUT RESULTS ;
   ENDSTEP ;
```

Assuming that the program writes to RESULTS, assigned to external-file-name MY.DATA, the SYSOUT mechanism will be used to create MY.DATA ; the required editing parameters, in this case the default installation parameters, will be incorporated into the file as it is created (subject to the Restriction on Record Size described below) and the edited file will be printed after the end of the current job. The data written to RESULTS will be preserved in the edited permanent SYSOUT file MY.DATA.

The –SYSOUT Suffix

If a COBOL program contains a statement of the form :

```
   SELECT file-name ASSIGN TO ifn-SYSOUT
```

the SYSOUT Mechanism will be used for the writing of data to the file with

the specified internal-file-name (corresponding to «ifn»). If the relevant step enclosure does not assign the internal-file-name, the SYSOUT Mechanism will create a standard SYSOUT subfile whose contents will be printed by the Output Writer and the subfile will be deleted ; no $SYSOUT statement is necessary in this case. If the internal-file-name is assigned to a permanent file, the SYSOUT Mechanism will create a permanent SYSOUT file, incorporating the required editing parameters into the file (subject to the Restriction on Record Size below). The contents of the permanent SYSOUT file will not be printed unless an appropriate $SYSOUT statement appears in the corresponding step enclosure or an appropriate $WRITER statement appears in the current job (see «Use of Output Writer Facilities» below).

Here is an example of part of a COBOL program which uses the –SYSOUT facility :

```
    SELECT OUT
    ASSIGN TO OUTFILE-SYSOUT
    ORGANIZATION IS LEVEL 64 SEQUENTIAL
FD OUT
    RECORD CONTAINS 100 CHARACTERS
    LABEL RECORD IS STANDARD
    OPEN OUTPUT OUT
    WRITE record name
    CLOSE OUT
```

The $DEFINE parameter SYSOUT

The use of the SYSOUT parameter in a $DEFINE statement in the relevant step enclosure has an identical effect to that of the suffix –SYSOUT in a user program. If there is no assignment of a file to be output, the parameter SYSOUT will force the creation and printing of a standard SYSOUT subfile. For a permanent file assigned and created in the current step, it will force the use of the SYSOUT Mechanism and edit the file as appropriate subject to the restriction on Record size given below. The contents of the permanent SYSOUT file will not be printed unless a corresponding $SYSOUT or $WRITER statement is specified.

NOTES :– The $DEFINE statement is normally used for the purpose of overriding, for a particular step execution, certain options that have been specified in a production program. For the use of the SYSOUT Mechanism users are advised to specify either –SYSOUT in the program (for standard program output) or $SYSOUT (useful for specifying particular output handling parameters) or both options together, instead of specifying the SYSOUT parameter in $DEFINE.

– If there is no –SYSOUT in a user's program but the user wishes to force the use of the SYSOUT Mechanism for the creation of a permanent SYSOUT file without printing the file contents, the following alternative to the use of a $DEFINE statement is available :

specify WHEN=DEFER in a $SYSOUT statement in the corresponding step enclosure (see Output Handling Parameters, below).

Restriction on Record Size

In order that the SYSOUT Mechanism can edit a file at creation time (and thus increase the performance of the job), the file must have a record size of at least 600 bytes. This is no problem for standard SYSOUT subfiles since the record size of all members of SYS.OUT is above this minimum value. However, in the case of permanent SYSOUT files, the user must allocate the file with an appropriately large record size. If this is not done, the record size given in the program will be used and

the SYSOUT **Mechanism** will not edit the file at file creation time. The recommended procedure is to specify a $PREALLOC statement before the step that creates the permanent SYSOUT file. The record advised is VB.

Example :

```
PREALLOC R2D2, DEVCLASS=MS/M300,
          GLOBAL=(MEDIA=VADAR, SIZE=10), FILESTAT=UNCAT,
          BFAS=(SEQ=(BLKSIZE=5000, RECSIZE=1000,
                                   RECFORM=VB) ) ;
```

If a $ALLOCATE statement is used to allocate a permanent SYSOUT file, a $DEFINE statement that specifies the appropriate record size must appear in the same step. This is useful for an uncatalogued tape file, since $PREALLOC cannot be used in this case.

Example (for disk) :

```
STEP   PROGA, . . . ;
  ASSIGN    OUTFILE, R2D2, DEVCLASS MS/M300, MEDIA=VADAR ;
  ALLOCATE OUTFILE, SIZE=10, UNIT=CYL ;
  DEFINE OUTFILE, RECSIZE=1000, BLKSIZE=5000, RECFORM=VB ;
ENDSTEP ;
```

Example (for tape) :

```
STEP PROGA . . . ;
     ASSIGN OUTFILE, TAPEFILE, DEVCLASS=MT/T9, MEDIA=TAPE01 ;
     DEFINE OUTFILE, RECSIZE=1000, BLKSIZE=5000, RECFORM=VB ;
ENDSTEP ;
```

NOTE : For a permanent SYSOUT file, the record size specification in the program is not affected by the RECSIZE value in $PREALLOC or $DEFINE. For example, the following file-processing statements might appear in a COBOL program that is associated with the above examples of $PREALLOC and $ALLOCATE. :

```
SELECT OUT
ASSIGN TO OUTFILE-SYSOUT
ORGANIZATION IS LEVEL 64 SEQUENTIAL
FLR
FD OUT
  RECORD CONTAINS 108 CHARACTERS
  LABEL RECORD IS STANDARD
  OPEN OUTPUT OUT
  WRITE record name
  CLOSE OUT
```

As far as the user is concerned, the program is still writing records with a length of 108 characters ; apart from the fact that trailing blanks are suppressed on a SYSOUT file, the record structure superimposed by the SYSOUT Mechanism is of no importance to the programmer.

EFFECT OF THE VARIOUS SYSOUT OPTIONS

Table 2-1 illustrates the effect of the main options concerned with the use of the SYSOUT Mechanism. If read horizontally, it shows what effect the presence (indicated by YES) or absence (indicated by NO) of certain situations in the program, the JCL, or the file label (in particular, the record size) have on a file that is created by a user program. Each result indicated on the right of the table gives the type of file created (e.g. standard SYSOUT subfile, edited permanent SYSOUT file) and whether or not the contents are printer or punched (column headed Output).

NOTES :

- The case where there is no –SYSOUT in the program and $SYSOUT does not appear in the JCL is not considered to be relevant to this table (produces either an unedited file or an error condition, depending on whether or not the internal-file-name has been assigned within the step enclosure).
- A hyphen (–) in the column headed –SYSOUT indicates that either YES or NO can apply (the case where the column headed $SYSOUT contains YES).
- A hyphen in the column headed RECSIZE indicates that the entry is not applicable (the case where there is no permanent file).
- If a $SYSOUT statement contains the parameter WHEN=DEFER, the entries marked with an asterisk in the column headed Output will not produce output in connection with the current step.
- The presence in the step enclosure of a $DEFINE statement with the parameter SYSOUT has an identical effect to that of –SYSOUT in the program.

TABLE 2-1. SYSOUT Options

Program	JCL		Permanent File	Result	
SYSOUT	$SYSOUT	$ASSIGN	RECSIZE	File	Output
YES	NO	NO	-	standard SYSOUT	YES
-	YES	NO	-	standard SYSOUT	YES
YES	NO	YES	⩾ 600	permanent SYSOUT edited	NO
YES	NO	YES	< 600	permanent not edited	NO
-	YES	YES	⩾ 600	permanent SYSOUT edited	YES
-	YES	YES	< 600	permanent SYSOUT not edited	YES

AVOIDING THE USE OF THE SYSOUT MECHANISM FOR OUTPUT EDITING

There are certain cases where the use of the SYSOUT Mechanism is unsuitable for the editing of permanent files at creation time ; for example :

- where the contents of a permanent SYSOUT file are to be reused by another system component (e.g. as input to $LIBMAINT) ;

- where a file is used in conjunction with the report selection facility of the COBOL Report Writer (which means that the REPORT option of the $WRITER statement cannot be used with files in SYSOUT format) ; this rule also implies that the COBOL Report Writer selection facility cannot use standard SYSOUT subfiles since they are by definition in SYSOUT format.

In addition, there are a few cases where the editing of a file at creation time might degrade the job performance ; for example where the contents of a file are only to be punched - which requires re-editing by the Output Writer (see «Notes on Punched Card Output» below), or where a permanent SYSOUT file is to be created but in general the contents will not be printed.

In the above situations, the user can override the presence of the –SYSOUT suffix in the program in one of the following ways :

- by preallocating the file with a record size of less than 600 bytes (normally the record size given in the program) ; if in this case a $SYSOUT statement is specified for the file, the SYSOUT Mechanism will take account of the enqueuing requested (i.e. the value of the WHEN parameter) but the file will not be edited.

  NOTE : This method is not possible for uncataloged tape files since $PREALLOC cannot be used for them.

- by specifying the parameter NSYSOUT in a $DEFINE statement ; this is the usual way to avoid the use of the SYSOUT Mechanism for an uncataloged tape file (otherwise, of –SYSOUT appears in the program, the system forces a record size equal to that of SYS.OUT, i.e. at least 600 bytes) ; the record size in the program will apply, the SYSOUT Mechanism will not be used to write the file, and the file will not be edited.

  NOTE : If the SYSOUT Mechanism is not used the file will be written as a SSF, SARF or ASA format, see COBOL user Guide.

OVERRIDING RULES FOR THE SYSOUT MECHANISM

Figure 2-2 illustrates in the form of a low diagram the overriding rules that decide whether or not a SYSOUT file will be edited.

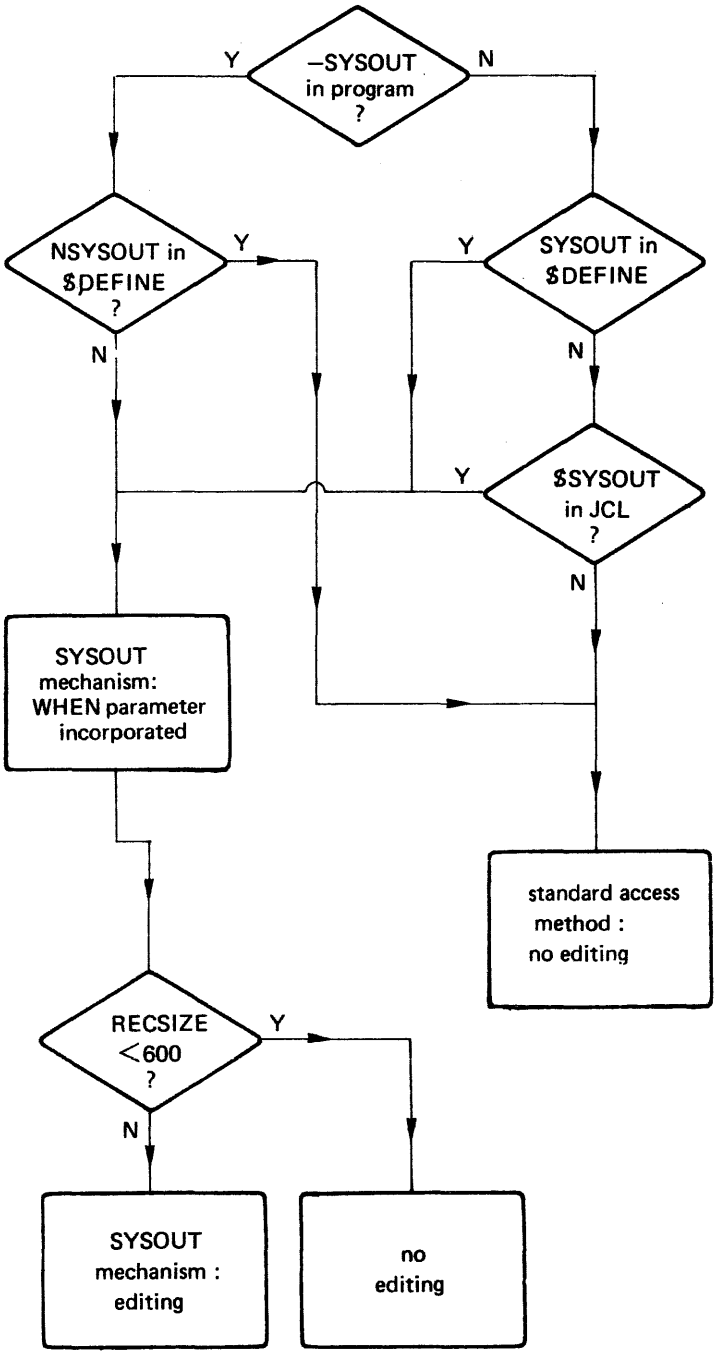


Figure 2-2. Overriding Rules for SYSOUT Mechanism

Use of Output Writer

STANDARD SYSOUT SUBFILES

The most frequent use of the Output Writer will be for the printing of reports on standard stationery with no requirement to maintain a copy of a report on disk or tape. A neat way of doing this is to use the $SYSOUT statement, for example :

```
SYSOUT F1 ;
```

(where F1 is an internal-file-name). There is no need for a $ASSIGN statement ; in fact, as explained previously, a $ASSIGN statement is only relevant for permanent SYSOUT files. To produce punched card output, specify the appropriate device class, for example :

```
SYSOUT F2, DEVCLASS = CD/P/C80, MEDIA = DECK ;
```

If the user wants the Output Writer to be notified at a time other than the end of the job (the default value of the WHEN parameter), or requires nonstandard stationery, the appropriate parameters can appear on the $SYSOUT statement ; for nonstandard editing (e.g. margin setting, form control) a $DEFINE statement is necessary. Both types of parameter are discussed later.

A statement of the form :

```
SELECT PRFILE ASSIGN TO F1-SYSOUT
```

is an alternative means of requesting the printing of a standard SYSOUT subfile (provided no $ASSIGN statement is given for the internal-file-name F1). However, if nonstandard options are required, a $SYSOUT statement (or a previous $OUTVAL statement) with the appropriate parameters must also appear.

One cannot open a standard SYSOUT subfile several times within a step ; at each open, the system will create a new member in SYS.OUT. For example, two standard SYSOUT subfiles will be created if a COBOL program contains the following statements :

```
SELECT OUTFILE
ASSIGN TO OUT-SYSOUT
        .
        .
OPEN OUTPUT OUTFILE
        .
        .
CLOSE OUTFILE
        .
OPEN EXTEND OUTFILE
        .
        .
CLOSE OUTFILE
```

The above example shows that, for a standard SYSOUT subfile, EXTEND processing mode is treated as OUTPUT processing mode.

Use of Several $SYSOUT Statements for One Subfile

Several $SYSOUT statements can appear for the same standard SYSOUT subfile if the user requires several copies of a listing and/or card deck, each copy having different characteristics.

For example, if a printed copy of a card deck is required :

```
$JOB CRDOUT, . . . . ;
    STEP   S1, . . . . ;
      SYSOUT CARD, DEVCLASS=CD/P/C80, MEDIA=DECK ;
      SYSOUT CARD ;
  ENDSTEP ;
$ ENDJOB ;
```

PERMANENT SYSOUT FILES

In some cases, a permanent copy of a report or punched card deck may be required on disk or tape. Furthermore, since there is a limitation on the size of the file SYS.OUT, a standard SYSOUT subfile should not be used when a job is likely to produce an excessive amount of output. In these cases, instead of using SYS.OUT, the user should make an assignment to the file in which the copy is to be held, for example :

```
ASSIGN F1, OUT.PRFL1 ;
```

or, for output to tape :

```
ASSIGN F1, XYZ.TAPE, EXPDATE= 3,
        DEVCLASS =MT/T9/D1600, MEDIA = Q45 ;
```

To request the printing or punching of a report created in the current step, inform the Output Writer by means of a $SYSOUT statement, for example :

```
SYSOUT F1 ;
```

If at a later stage (in the current job, or in another job) the user requires a printed or punched copy of the file, a $WRITER statement will be necessary ; for example, for a printer listing :

```
WRITER OUT.PRFL1 ;
```

or for punched cards :

```
WRITER OUT.PRFL1, DEVCLASS=CD/P/C80, MEDIA=X99 ;
```

Both output handling and editing parameters can appear in the $WRITER statement. These are discussed later in this Section.

NOTE : The $ASSIGN statement for a permanent SYSOUT file must not contain the parameter MEDIA = WORK nor the DVIDLIST parameter.

The WRITER (efn DEVCLASS =.....MEDIA= .....) are residency parameters and

WRITER (DEVCLASS=....MEDIA=...) are editing parameters.

Filling of Permanent SYSOUT Files in Several Steps.

The user can fill a permanent SYSOUT file in several steps and request the Output Writer to print or punch the complete file, as follows :

1. Open the file in OUTPUT processing mode in the first step that writes to it ; open the file in EXTEND processing mode in the other steps that write to it.

2. If a $SYSOUT statement appears in one or more of the steps, insert the WHEN = DEFER parameter to prevent a request to the Output Writer.

3. Ensure that the output records are processed in an identical manner in each step (i.e. either all of the records are edited by the SYSOUT Mechanism, or none of them are).

4. Use a $WRITER statement to make the request to the Output Writer.

NOTE : To ensure that the latest contents of the file are printed or punched even if one of the job steps aborts, do one (or both) of the following :

- Make use of the $JUMP statement (see Section VI).
- Put the $WRITER statement immediately after the $JOB statement.

Partial Output of Files

The PART parameter of the $WRITER statement allows the user to output specified parts of a permanent SYSOUT file.

Example :

```
WRITER    (PFILE, DEVCLASS=MS/M350, MEDIA=C018),
          PART = (40:60, 90:$) ;
```

The above example prints pages 40 to 60 and pages 90 to the end, for an uncataloged disk file.

The SUBFILES parameter of the $WRITER statement allows the user to output specified members of a permanent SYSOUT library file.

Example :

```
WRITER MY.LIB, SUBFILES = (TOM, DICK, HARRY) ;
```

The above example prints members TOM, DICK and HARRY of the cataloged library file MY.LIB.

Deallocation of a Permanent SYSOUT File

Since the activity of the Output Writer is independent of the execution of the job that has requested its use, the user must take care before attempting to deallocate a permanent SYSOUT file after an output request has been made for it. The user is advised never to use the $DEALLOC statement for a file in the same jobs as it is output, nor even in a later job unless it is certain the file has already been printed or punched. Otherwise, there is a danger that the file will be deallocated before it has been output. In order to overcome the problem, the user can do one of the following :

- use a standard SYSOUT subfile instead of a permanent SYSOUT file ;
- put the $DEALLOC statement in a separate job whose $JOB statement contains the HOLD parameter ;
- use a member of a permanent library file as the SYSOUT file and include the DELETE parameter in the $SYSOUT or $WRITER (or $OUTVAL) statement.

Editing and Handling of Output

It is important to distinguish between output parameters that are concerned with the editing of the data to be output and those that deal with the handling of listings and decks.

OUTPUT EDITING

The parameters that are used to specify editing characteristics are as follows :

- for the printer :
  - the MEDIA parameter of $SYSOUT and $OUTVAL and the PRINTER parameter group of $DEFINE and $WRITER ;
  - the DEVCLASS parameter of $SYSOUT and $OUTVAL (type of printer and number of hammers)
- for the card punch : the PUNCH parameter group of $DEFINE and $WRITER.

For a printer, the value of the MEDIA parameter (or the default value if MEDIA is not specified) determines which default system parameters for output editing will apply. If the SYSOUT Mechanism od is used to edit the file at creation time, a $DEFINE statement in the current step will override and/or complement these parameter values. Thus, for an «edited» SYSOUT file, all editing parameters are incorporated into the file at creation time. Provided the requirements for the output do not change, there is no need for editing parameters to be respecified each time an edited permanent SYSOUT file is to be printed. The above rules apply for punched card output except that there is a unique set of default system parameters for output editing in this case.

If a step creates an unedited permanent SYSOUT but a $SYSOUT in the STEP requires an edited output :
- the SYSOUT is output according to the specified edition parameters
- if the same editing parameters are required in the future then they must be restated in full.

OUTPUT HANDLING

The parameters that are used to direct the handling of output are those that appear in the $OUTVAL statement (except MEDIA for a printer). All those parameters can be specified also within the $SYSOUT and $WRITER statements. With the exception of the WHEN parameter (see below) they are obeyed at output time by the Output Writer. The required handling parameters must be specified each time a permanent SYSOUT file is to be printed or punched.

NOTE : Although the SLEW and NSLEW appear both in the $OUTVAL statement and in the PRINTER group of $DEFINE and $WRITER, they are treated as output handling parameters and apply only to a current request for the Output Writer.

LINES AND CARDS LIMITS

To limit the amount of output produced by a program, for example to anticipate the occurrence of an infinite loop, the user can specify in $STEP a maximum number of lines printed (LINES parameter) and cards punched (CARDS parameter). When a limit is reached, the program is abnormally terminated, with the return code ERLMOV.

Example : Suppose a program works satisfactorily with test data and produces less than 100 lines of SYSOUT output; if, for a particular production run, the user wants to print and punch the SYSOUT report, $STEP statement of the form given below should appear :

STEP STEP1, . . . , LINES = 100, CARDS = 100, . . . ;

NOTES : The limit controlled is the number of WRITEs that the program produces. In general, this value will be identical to the number of lines printed (or cards punched) ; however, if several copies of a SYSOUT file are made (COPIES parameter), the additional lines or cards produced by the extra copies are not included. The lines printed in the Job Occurrence Report are also independent of the LINES and CARDS limits.

OUTPUT EDITING PARAMETERS

Effect on Different SYSOUT File Types

The following paragraphs summarize the way in which editing is done for the different types of SYSOUT files.

STANDARD SYSOUT SUBFILES : Editing is done as the subfile is filled. Any given editing parameters of $OUTVAL and/or $SYSOUT and/or $DEFINE override the default system values.

EDITED PERMANENT SYSOUT FILES : Editing is done as the file is filled. If any editing parameters appear for a previous $OUTVAL statement or for a $SYSOUT and/or $DEFINE statement in the step in which the file is created, their values override the standard system parameter values. If an edited permanent SYSOUT file is output at a later stage by $WRITER, the Output Writer will do so according to the editing done at file creation time. The user can supply alternative editing parameter values in the $WRITER statement, but these will be ignored unless the FPARAM parameter also appears. Note, however, that this use of FPARAM to force new editing parameters will mean that extra processing time will be required in order to re-edit the file.

Example : Suppose an uncataloged permanent SYSOUT file is edited at creation time with a nonstandard form height setting and a nonstandard print density ; if the user wishes to print the file later with standard characteristics, the following $WRITER statement can be used :

```
WRITER (MYFILE, DEVCLASS=MS/M400, MEDIA=V1), FPARAM ;
```

Similarly, if a standard form height and print density are required but the user wants a different margin setting :

```
WRITER (MYFILE, DEVCLASS=MS/M400, MEDIA=V1),
        FPARAM, PRINTER = (MARGIN=10) ;
```

UNEDITED PERMANENT SYSOUT FILES : Since the record size is less than 600 bytes, the SYSOUT Mechanism does not edit the file at file creation time. If the file is to be output later with any nonstandard editing parameters, these parameters must appear in the $WRITER statement, even if they appeared in a $SYSOUT and/or a $DEFINE statement when the file was written. If the data records are not in SSF format, the user must specify the format in the DATAFORM parameter.

ORDINARY PERMANENT FILES : Since the SYSOUT Mechanism has not been requested at file creation time, the file is not edited. If the file is to be output with any nonstandard editing parameters, these parameters must appear in the $WRITER statement. If the data records are not in SSF format, the user must specify the format in the DATAFORM parameter.

Example :

```
WRITER MY.FILE, DEVCLASS = CD/P/C80, DATAFORM = SARF,
                PUNCH    = (CHARSET = EH14) ;
```

The above example will punch the contents of the SARF file MY.FILE (an unedited permanent SYSOUT file or an ordinary permanent file) in the H14 character set.

Media Definition for Printer

If the user wants nonstandard printer paper or a nonstandard printer character set, the necessary information can be supplied in the MEDIA parameter. The media-name contains the following two fields :

- the first two characters identify the character set to be used ;
- the remaining characters identify the paper form.

The Unit Record Devices User Guide describes in detail printer character sets and form numbers.

Example :

```
STEP STEPA, . . . ;
   ASSIGN PRINT, MY.PRFL ;
   SYSOUT PRINT, DEVCLASS = PR, MEDIA = I20003 ;
ENDSTEP ;
```

In the above example, the character set number is I2 and 0003 is the form number.

NOTE : The standard character set is I1, which consists of the following characters :

```
0 1 2 3 4 5 6 7 8 9
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

space & – /\
¢ ! : . $ , # < * % @ ( )
–' +; >=| ¬ ? "
```

Logical Page Setting

Within a physical printer page, the user can define a «logical» page within which the output text will be confined. The relevant parameters are :

- head of form (HOF) ; this is the first line to be printed in a page ;
- full form 1 (FF1) ; this identifies the last line to be printed in a page ;
- margin (MARGIN) ; this is the number of character positions on the left of the page to be skipped before the first character of each line is printed.

The Unit Record Devices User Guide explains these concepts in detail.

NOTES : – There is also a Full Form 2 (FF2) ; if the corresponding line is reached when the printer is used directly (see below), a return code is produced.

– The following relationship must hold :
0 ⩽ HOF ⩽ FF2 ⩽ FF1 ⩽ FH
where FH is length of the page.

– The standard paper form (number 0000) has the following characteristics : HOF = 5
FF1 = 60
MARGIN = 0

This means that each page has 4 blank lines at the top and 6 blank lines at the bottom (since FORMHT = 66).

Setting Stop Levels in a Printer Page

The user can define stop Levels within a page by means of the CHi parameters. Refer to the Unit Record Devices User Guide for more details.

Setting Margins for Punched Cards

The user can specify the number of initial card columns to be skipped when the Output Writer punches cards, by means of the MARGIN parameter.

Example :

```
STEP STEPC, . . . ;
   SYSOUT CRDS, DEVCLASS = CD/P/C80, MEDIA = DECK ;
   DEFINE CRDS, PUNCH =(MARGIN=9) ;
ENDSTEP ;
```

In the above example, the data in the punched cards will begin in card column 10.

NOTE : For punched card output, the record size declared in the program must observe the following relationship with the value of MARGIN (number-card-columns) :

number-card-columns + record-size $\leqslant$ 80

Binary Punching

There are two types of binary mode punching that the user can request : L64 mode (BINARY parameter) and H200 mode (HBINARY parameter). These modes are described in the Unit Record Devices User Guide.

Example :

```
STEP STEPX, . . . ;
   SYSOUT BINOP, DEVCLASS=CD/P/C80, MEDIA=CARD ;
   DEFINE BINOP, PUNCH = (BINARY) ;
ENDSTEP ;
```

Character Set for Punched Cards

If the required code conversion for card output is different from EBCDIC to H36, the user can specify the translation table name by means of the parameter CHARSET. Refer to the Unit Record Devices User Guide.

Example :

```
WRITER (H14CRDS, DEVCLASS=MT/T7, MEDIA = TP5),
     DEVCLASS = CD/P/C80, MEDIA=H14DK, PUNCH=
                    (CHARSET = EH14) ;
```

OUTPUT HANDLING PARAMETERS

Enqueuing of Output Writer Requests

By default, a request to output a SYSOUT file is sent to the Output Writer when the current job terminates. By use of the WHEN parameter, it is possible to change the time that the Output Writer is notified. The possibilities are :

- at job termination (default value) — WHEN = JOB
- at step termination (for $SYSOUT only) ; for a job that contains several steps, this option allows the output of one step to be printed or punched concurrently with the execution of later steps (depending on the current activity of the Output Writer - see note 4 below). — WHEN = STEP
- as soon as the SYSOUT file is closed (i.e. «immediately»); — WHEN = IMMED
- no Output Writer request made at this point (for permanent SYSOUT files only, with $SYSOUT) — WHEN = DEFER

NOTES :

1. The system always takes account of the value of the WHEN parameter on a $SYSOUT statement even if the SYSOUT Mechanism is not used edit the file. This means that, if a $SYSOUT statement is specified for a permanent file with a record length of less than 600 bytes, the enqueuing request is observed even through the editing parameters. Editing parameters must be restated if a future output is required, see SYSOUT Mechanism above.

2. The WHEN = DEFER option is useful in the following situation. A program creates a permanent SYSOUT file but the user wishes to delay the output of the file contents ; however, the program does not contain the –SYSOUT suffix in the SELECT statement. In order to use the SYSOUT Mechanism to write to the file (to increase efficiency by editing at the time of file creation), the user can add to the JCL a $SYSOUT statement that contains the parameter WHEN = DEFER. Provided that the record size is large enough (see SYSOUT Mechanism above), the SYSOUT Mechanism will edit the file but the file contents will not be printed or punched. For example, if a program contains the following statement :

   SELECT OUT ASSIGN TO OUTFILE-PRINTER.
   the following step enclosure will ensure the editing of the file assigned to OUT without printing its contents.

```
STEP STEPA, . . . ;
   ASSIGN OUTFILE, MYTAPE, DEVCLASS=MT/T9, MEDIA=TAPE03 ;
   SYSOUT OUTFILE, WHEN=DEFER ;
ENDSTEP ;
```

3. The requests to output the SYSOUT files that are created by various system utilities (e.g. $CREATE, LIBMAINT are made, by default, at job termination (i.e. WHEN =JOB). Where the utility statement contains the PRTOUT parameter, the user can override this default value.

   Example :

```
LIBMAINT SL LIB = MY–LIB COMFILE = *MY-IN
                PRTOUT = (WHEN = IMMED) ;
```

4. The time between the request to the Output Writer and the start of printing or punching on the output device is dependent on the current Output Writer activity and on the order of the request within the output queue. Even if WHEN = IMMED is specified, for example, the printing or punching may still be delayed beyond the termination of the step or job. Refer also to Deallocation of a Permanent SYSOUT File, above.

Output Selection and Naming

Each output request in the output queue belongs to a given output class and also has a particular output priority. The existence of different output classes means that the operator can control the printing and/or punching of different catagories of output. For example, if several types of paper are used in an installation, all the requests for output on a particular type of paper should belong to one class and requests for output on a different type should belong to a different class, for each type of paper ; if the operator activates the Output Writer only for one output class, all the listings on the corresponding type of paper will be printed consecutively. The user can specify the class and priority for each output request by means of the CLASS and priority parameters. The operator can select a particular output class by means of the operator SO command.

NOTE : The requests to output the SYSOUT files that are created by various system utilities (e.g. CREATE, LIBMAINT), belong, by default, to output class C. The user can override this value by means of the PRTOUT parameter in the utility statement.
Example :

```
LIBMAINT SL LIB=MY-LIB COMFILE=MAY.SEQ PRTOUT= (CLASS=D);
```

The $OUTVAL statement can also influence the output class of output requests from subsequent utility statements (see $OUTVAL Statement, below).

The user can prevent the selection by the Output Writer of an output request, by means of the HOLD parameter. The Output Writer will not select a «held» request until the operator specifies a Release Output (RO) command.

NOTES : – The HOLD parameter differs fundamentally in use from the WHEN= DEFER parameter : the WHEN=DEFER parameter prevents the notification of the Output Writer, since the output request is not made and no entry is put in the output queue ; the HOLD parameter delays the printing or punching by the Output Writer, but the request is made and an entry is put in the output queue.

– If the user specifies HOLD for an output request, it is advisable also to specify the NAME parameter so that the operator can easily identify this request. Name is also useful for identification purposes if a job produces many output listings.
Example :

```
STEP ST1, . . . ;
   ASSIGN INPT, *ADECK ;
   SYSOUT PRT1, NAME= REPA;
   SYSOUT PRT2, HOLD, NAME=HLDOP ;
   SYSOUT PRT3, NAME= REPB;
ENDSTEP ;
STEP ST2, . . . . ;
   ASSIGN FL1, ABC.X14 ;
   ASSIGN FL2, ABC.Q14 ;
   SYSOUT FL2, NAME=REPC ;
ENDSTEP ;
```

- If the $JOB statement contains the HOLDOUT parameter, all output requests made within the job will be held until released by the operator ; the user can override this for a particular request by using the NHOLD parameter of the $SYSOUT or $DEFINE statement (or for several consecutive requests by using the NHOLD parameter of the $OUTVAL statement).

Production of Several Copies

The COPIES parameter allows the user to make several identical copies of a SYSOUT file.

Example :

```
STEP STEPA, . . . ;
  SYSOUT PRINT, COPIES = 3 ;
ENDSTEP ;
```

Output Banners

The user can suppress the standard output banners that appear on listings and card decks or supply alternative values for the items they contain (e.g. Run Occurrence Number, user-name). The appropriate parameters are NBANNER and BANINF respectively. Details of the format of printer listing banners are given in Section VII ; each punched deck is preceded and followed by there banner cards, as follows :

- a flag card
- a card containing the job's Run Occurrence Number and the user's name and job statement
- a card containing the Job Identification and account of the $JOB statement.

Deletion of Library Members

If a permanent SYSOUT file is a member of a permanent library file, the user can delete the member from the library after it has been printed or punched, by use of the DELETE parameter.

Example :

```
STEP PROGA, . . . ;
  ASSIGN OUT, PRTLIB, DEVCLASS=MS/M400, MEDIA=VOL13,
                          SUBFILE = TEMPFL ;
  SYSOUT OUT, DELETE ;
ENDSTEP ;
```

Suppression of Skip Function

The user can force the replacement of every skip function in the program by a skip to the following line, by means of the NSLEW parameter.

USE OF THE $OUTVAL STATEMENT

By means of the $OUTVAL statement, the user can supply output handling parameter values that override the default system parameter values. The new default values will apply to all $SYSOUT and $WRITER statements that appear after the $OUTVAL statement, up to the next $OUTVAL statement or, if there are no more $OUTVAL statements, to the end of the job. Any explicit appearance of a particular parameter value in a $SYSOUT or a $WRITER will in turn override for that statement only any new default value supplied by $OUTVAL. Note that the $OUTVAL statement takes effect at the time of the execution of the job and not at JCL translation time.

Examples :

1.
```
$JOB XYZ, . . . ;
  OUTVAL CLASS=D ;
    .
    .
    SYSOUT OP1 ;
    .
    .
    SYSOUT OP2, CLASS=E ;
    .
    .
    SYSOUT OP3 ;
    .
    .
$ENDJOB ;
```

In the above example, OP1 and OP3 will belong to class D and OP2 to class E.

2.
```
$JOB ABC, . . . ;
  OUTVAL CLASS=D ;
START :
  STEP ST1, . . . ;
  SYSOUT OP1 ;
    .
    .
  ENDSTEP ;
  OUTVAL CLASS= E ;
  STEP ST2, . . . ;
    SYSOUT OP2 ;
      .
      .
    ENDSTEP ;
    JUMP START, SW1, EQ, 1 ;
  $ENDJOB ;
```

The first time ST1 is executed, its output will belong to class D ; if, as a result of the $JUMP statement, ST1 is executed again, its output will then belong to class E.

THE JOB OCCURRENCE REPORT AND THE JOBOUT

The first $OUTVAL statement (for the printer) that appears before the first step in a job description defines the output characteristics of the Job Occurrence Report ; if no such statement is specified, or if a $OUTVAL statement without any parameters is specified, the job Occurrence Report will be printed according to the following parameter values :

CLASS=C, PRIORITY=3, WHEN=JOB, COPIES=1

The group of standard SYSOUT subfiles that have the same output characteristics as the file that contains the JOR is considered as a single file for output purposes and is known as the JOBOUT. For example, suppose a job does a compilation, a linkage and an execution of a program ; assume that no nonstandard output handling parameters are specified in the job description ; as a result, the operator will be aware of only two output listings :

JOB_REP	which contains the Job Occurrence Report
JOB_OUT	which contains the JOBOUT, i.e. the output from the compiler, the linker and the step corresponding to the user program.

The discussion in the first paragraph above about the JOR also applies to the JOBOUT.

Examples :

1.
```
$JOB QRS, . . . . ;
   OUTVAL CLASS = E ;
   STEP STPA, . . . . ;


   $ENDJOB ;
```

In the above example, the JOBOUT contains all the output listings with characteristics WHEN=JOB, CLASS=E, PRIORITY=3, COPIES=1.

2.
```
$JOB TUV, . . . . ;
   OUTVAL ;
   OUTVAL CLASS=D, PRIORITY=4, COPIES=2 ;
   STEP STAB, . . . . ;
      .
      .
$ENDJOB ;
```

In the above example, the JOR is the only listing that has the characteristics CLASS=C, PRIORITY=3, WHEN=JOB, COPIES=1 (assuming that there is no other $OUTVAL statement in the job description, and that no $SYSOUT nor $WRITER statement specifies those four values).

NOTES ON PUNCHED CARD OUTPUT

Use of SYSOUT Mechanism

When the SYSOUT Mechanism is used at file creation time, it edits the data records as if they were destined for the printer (even if the $SYSOUT statement specifies the card punch) ; in other words, for reasons of efficiency, SYSOUT format is compatible with the format required by the printer device. At the time a SYSOUT file is punched, the Output Writer will transform the edited records into the format

suitable for the card punch.

In order to avoid this re-editing, the user is advised not to use the SYSOUT Mechanism for card output unless the SYSOUT file is also to be printed ; in other words, if a SYSOUT file is only to be punched, the user should use a permanent SYSOUT file preallocated with a record size less than 600 bytes ; this prevents the SYSOUT Mechanism from editing the file at creation time.

Output of Source Programs, Compile Units and Load Modules

This Section has been principally concerned with the output of data. For the punching of the contents of source programs, compile units and load modules, the user can execute the $LIBMAINT utility. Details appear in the Library Maintenance User Guide.

Example of the Uses of $SYSOUT and $WRITER in a Job

The following example contains a variety of output request. An illustration of the different effects of these statements is shown in Figure 2-3.

```
$JOB . . . ;
      STEP STEP1, . . . . ;
         ASSIGN IN1, *INCRDS ;
         ASSIGN G1, MY.P99,END=PASS ;
COMMENT'SEND THE OUTPUT OP1 TO A PERMANENT SYSOUT FILE' ;
         ASSIGN OP1, MY.OUT1 ;
   ENDSTEP ;
   STEP STEP2, . . . ;
         ASSIGN IN2, MY.P99, END=PASS ;
COMMENT' USE A STANDARD SYSOUT SUBFILE FOR THIS STEP' ;
         SYSOUT OP2 ;
   ENDSTEP ;
JUMP ST3, SEV, GE, 3 ;
COMMENT 'NOW REQUEST COPY OF STEP1 OUTPUT' ;
   WRITER MY.OUT1 ;
ST3 : STEP STEP3, . . . . ;
      ASSIGN IN3, MY.P99 ;
COMMENT 'SEND SOME OUTPUT TO A PERMANENT SYSOUT FILE' ;
      ASSIGN OP3A, MY.OUT3 ;
COMMENT 'REQUEST TO BE MADE AT STEP TERMINATION' ;
      SYSOUT OP3A, WHEN=STEP ;
COMMENT 'USE A STANDARD SYSOUT SUBFILE FOR MORE OUTPUT ' ;
      SYSOUT OP3B ;
   ENDSTEP ;
$INPUT INCDRS ;
      .
      .
      .
$ENDINPUT ;

$ENDJOB ;
```

```
$JOB . . . ;
  STEP STEP1, . . . . ;
     ASSIGN IN1, *INCRDS ;
     ASSIGN G1, MY. P99, END=PASS ;

     ASSIGN OP1, MY.OUT1 ;
  ENDSTEP ;


  STEP STEP2, . . . ;
     ASSIGN IN2, MY.P99, END=PASS ;
     SYSOUT OP2 ;
  ENDSTEP ;
  JUMP ST3, SEV, GE, 3 ;
  WRITER MY.OUT1 ;

  ST3 : STEP
     ASSIGN IN3, MY.P99 ;
     ASSIGN OP3A, MY.OUT3 ;
     SYSOUT OP3A, WHEN = STEP ;
     SYSOUT OP3B ;
  ENDSTEP ;
```

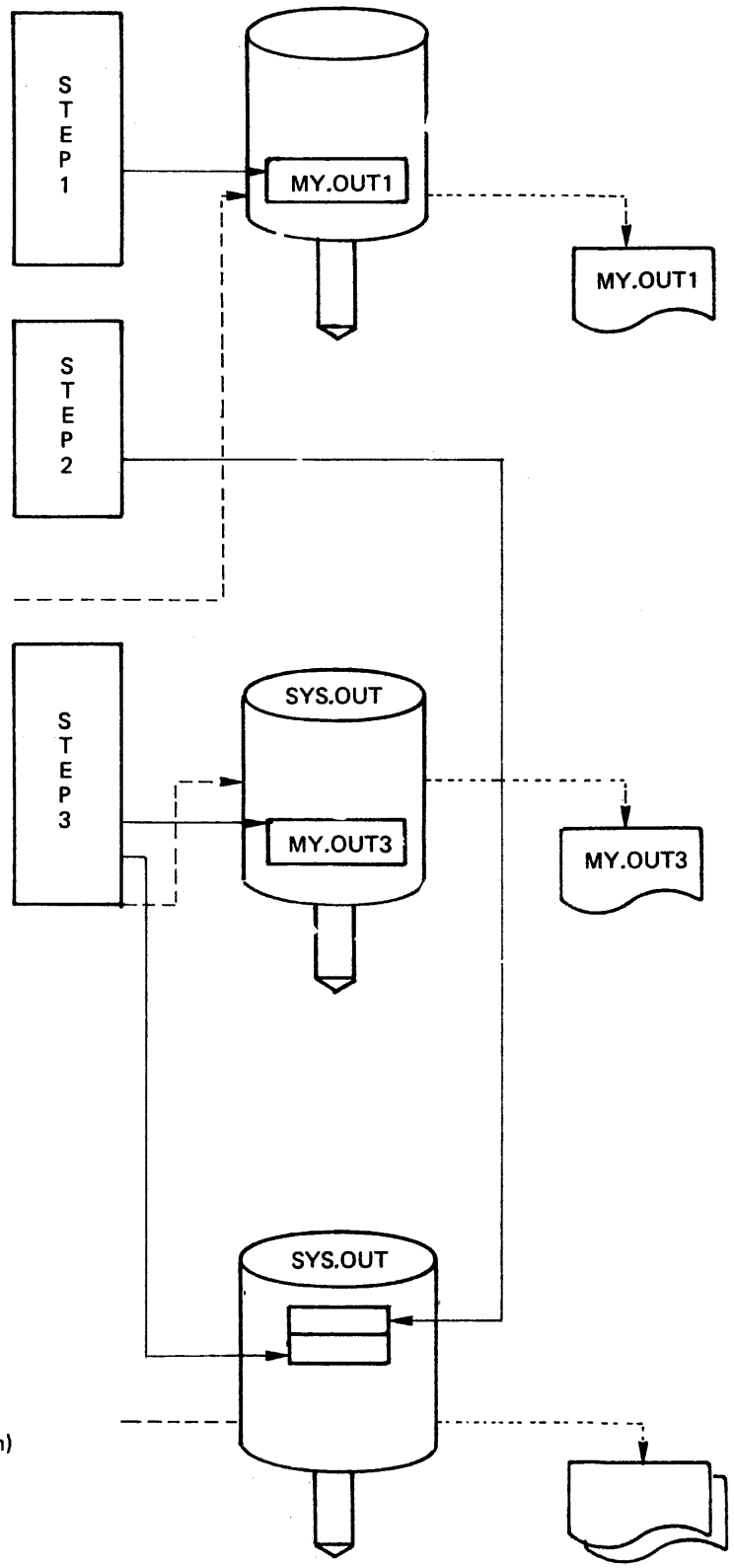


Figure 2-3. Job Output

Direct Use of the Printer and Punch

When a unit record device is used directly, the records written are sent to the device without any intermediate storage on a temporary or a permanent file. The user has exclusive use of the device until the execution of the current step has terminated or, if the COBOL program contains the WITH LOCK option, when the CLOSE statement is executed.

To use a printer or punch directly the user must specify the device in a $ASSIGN statement. In addition, the COBOL program should contain the suffix –PRINTER or –CARD–PUNCH as appropriate in the ASSIGN clause of the SELECT statement. The $ASSIGN statement can also specify the paper form for a printer (MEDIA parameter).

Examples :

1.
```
$JOB DIRECT, . . . . ;
   STEP STEPA, . . . ;
      ASSIGN PROUT, DEVCLASS=PR, MEDIA= I50000 ;
   ENDSTEP ;
$ENDJOB ;
```

2.
```
$JOB EXCARD, . . . ;
   STEP STEPA, . . . . ;
      ASSIGN CROUT, DEVCLASS =CD/P/C80,MEDIA=DIRECT ;
   ENDSTEP ;
$ENDJOB ;
```

NOTES :

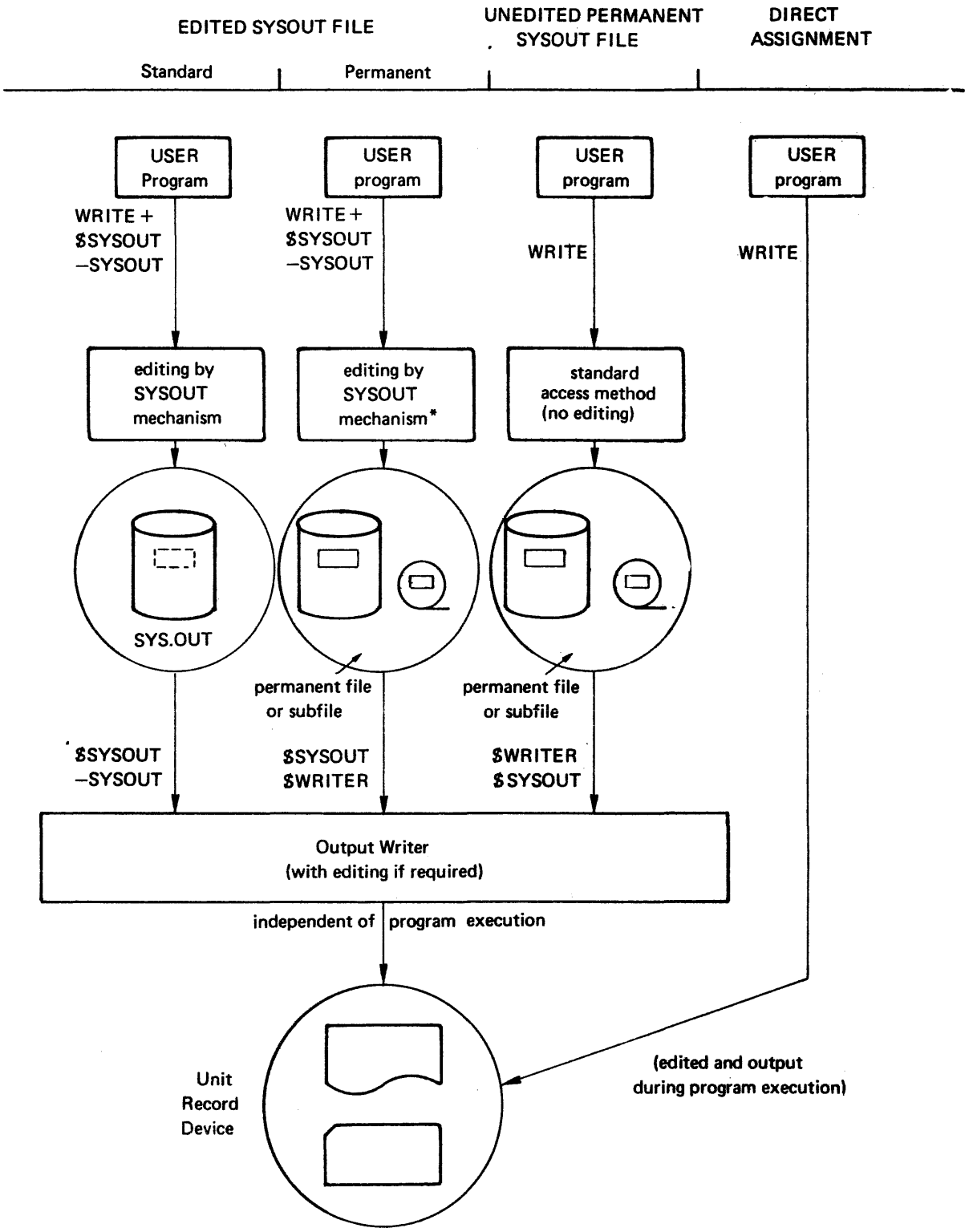
By default, cards are punched in H36 code ; the user can specify a different mode by means of the $DEFINE statement (CHARSET parameter).

If a device is used directly, no banners are provided.

The only relevant parameters of the $ASSIGN statement for direct use are the internal-file-name, DEVCLASS, MEDIA and POOL.

Diagram of Output Facilities

Figure 2-4 summarizes the action of the different output methods that are available to the user.



*no editing if record size < 600 bytes

Figure 2-4. Summary of Output Facilities

3•File assignment and allocation

Associated with each job step, and necessary to its execution, are a number of system resources such as memory space, physical files and devices. The user is able to exercise considerable control over the handling of files and devices by means of JCL statements.

This section explores the various means by which the user allocates space for and assigns files for use in a user job and the use of the catalog facility.

The allocation of file space is carried out by either the $ ALLOCATE statement or the $ PREALLOC utility, while the allocation of device and volume is performed in conjunction with the $ ASSIGN statement. The following paragraphs explain the concepts involved in resource allocation.

These major classes of files are available to the GCOS Level 64 user, namely :
- temporary files
- permanent cataloged files
- permanent uncataloged files

These three classes inducle the standard file formats (BFAS, UFAS, HFAS) ; file organisations (e.g. sequential, indexed sequential) and media types (disk, tape, cassette), see Table 3-1 below for restrictions

FILE TAPE		UFAS			BFAS			HFAS		
		S	IS	R	S	IS	R	S	IS	R
TEMPORARY	T	X			X			X		
	D	X	*I	*I	X	*I	*I	X	*I	*I
	C	X			X			X		
CATALOGED	T	X			X					
	D	X	X	X	X	X	X			
	C	X			X					
UNCATALOGED	T	X			X			X		
	D	X	X	X	X	X	X	X	X	X
	C	X			X			X		

Table 3-1 File Class, Organization and Media
I* Not normally used.
S= SEQUENTIAL, IS=INDEXED SEQUENTIAL, R=RANDOM T=TAPE, D=DISK, C=CASSETTE

A temporary file is available to the user when a work area is required in which the data is not needed after the end of the step. Such a file exits only for the duration of the step and is deleted at step termination. If the user requires such a file be retained for more than one step within a job, but feels that the creation of a permanent file is not justified, then the file may be passed (END=PASS, see File Passing), to the

next step. A temporary file is never accessible to another job. The organisations available to temporary files are the same as for permanent files, but are normally sequential. Space for a temporary file is made available using $ALLOCATE in the step in which the file is to be used. The $PREALLOC statement can also be used when the options available in $ALLOCATE are insufficient to describe the temporary file to be generated. When the user makes no explicit space allocation a default value of 1 cylinder is available for a temporary file.

Permanent files are of two types, cataloged and uncataloged. The main difference between these two classes is the catalog. A catalog contains information such as file location, file generations and usage. The catalog thus simplifies user JCL since a cataloged file may be referenced simply by its internal-file-name (see $ASSIGN). The highest level of catalog is the site catalog which contains information on users, projects, billings. Private catalogs can be created which contain information on user files. Where private catalogs are used a search path can be defined using $ATTACH, so that files, or objects can be more readily found (see Catalog Overview).

Space for a permanent file can be made available using $ALLOCATE (within a step) or in a job enclosure using the $PREALLOC utility. On all three classes of file, a program is written to access a file name described in the conventions of the language being used. When step execution is launched, access will have to be made to a physical file. The link between the file as known by the program, and the file as known by GCOS has to be established. This link is provided using the $ASSIGN statement in which the name of the file as known to the program (internal-file name, or ifn) and the name of the file as known GCOS (external-file-name, efn) is established within a step. The $ASSIGN statement also establishes the class of file, access rights and device requirements (refer to JCL Language Reference Manual). The $ASSIGN statement allows any user program to see an input enclosure as ordinary sequential file.

```
$JOB TEST, USER=TD,PROJECT=ED ;
  STEP LMI, LMUB.EXS ;
       ASSIGN CRDN1, INENC1, DEVCLASS = MS/M300,
            MEDIA = NI ;
COMMENT ' IFN CRDNI IS CONNECTED TO EFN INENC1, WHICH
       IS AN UNCATALOGED DISK FILE';
       ASSIGN DATAI, *CRDR ;
COMMENT ' INPUT ENCLOSURE DATAI IS READ INTO CRDR';
       ASSIGN STAS, CDEX.DCOM, TEMPRY ;
COMMENT ' TEMPORARY FILE STAS IS ASSIGNED TO CDEX.DCOM,
       AND SINCE NO PASS PARAMETER IS STATED THE FILE IS
       DELETED AT THE END OF THE STEP' ;
       ASSIGN IN3, TD.COM ;
COMMENT ' CATALOGED FILE OF THE SITE CATALOG' ;
       ENDSTEP ;
$ENDJOB ;
```

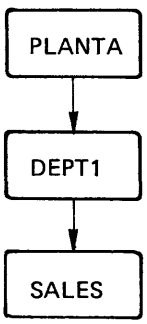
CATALOG OVERVIEW

The catalog facility of Level 64 allows you to keep a permanent record of each file. A catalog is, in itself, a permanent file consisting of a set of object records, each of which contains information on a file. The information includes the file name, the generation, sharing information and so on.

Using catalogs gives you many advantages, including ;

- All the information that is needed to locate and use a file is held in one place, and is easy to access and update.
- This information need not be supplied in most JCL statements. Normally it is retrieved automatically from the catalog. For example, the $ASSIGN Basic JCL statement for a cataloged file can normally be reduced to $ASSIGN ifn, efn ;
- By using file generations, the JCL required to update a file need not be changed for each update, and previous versions of the file can remain accessible.
- Greater control of file usage and access is possible, with project controllers being able to allocate files and other resources to projects, and oversee their usage.

Permanent-file-names consist of a combination of component names and separators (see the JCL Reference Manual for full details of file naming conventions), and each component name represents an object record in the catalog. The object records are linked together to form a tree-structure and thus, for example, the tree-structure created for a file called PLANTA.DEPT1.SALES would be :



PLANTA is known as a Master Directory because it is at the top of the tree, DEPT1 is a directory, and SALES, the lowest level, is a file. All the information on the file PLANTA.DEPT1.SALES is held in the object record SALES. The other object records associated with the file only act as pointers and unique identifiers for the file.

The catalog entry for a file is created by the $CATALOG extended JCL statement, with any additional information required to complete the entry being supplied when the file is preallocated. The information contained in the catalog entry overrides the same information if it is supplied through JCL statements.

Once a file has been cataloged, the catalog containing the file description must be available to any job which uses the file. This is done by the $ATTACH extended JCL statement, which attaches up to five catalogs to any job. If the file is cataloged in the SITE. CATALOG. there is no need to have a $ATTACH statement, since, when the SITE. CATALOG is the only one to be searched, the search is performed automatically.

Note : The maximum number of catalogs that can be attached to all jobs at one time is 10. Jobs which ask for new catalogs when this limit has been reached will be enqueued.

To illustrate how the use of catalogs simplifies JCL, consider the $ASSIGN statement.

To assign the uncataloged file PLANTA.DEPT1, the $ASSIGN statement might be :

```
$ASSIGN ifn, PLANTA.DEPT1,SHARE = ONEWRITE,
        EXPDATE = 100,DEVCLASS = MS/M400,
        MEDIA = (D1, D2) ;
```

This information would have to be given for the file every time a $ASSIGN on the file occurs. If the file was catalogued, the corresponding $ASSIGN statement would be :

```
$ASSIGN ifn,PLANTA.DEPT1 ;
```

The other information is contained in the catalog entry for the file PLANTA.DEPT1, and this entry would have been set up by statements of the form :

```
$CATALOG PLANTA.DEPT1, SHARE = ONEWRITE, RETPER=100;
```

The remainder of the entry (the media where the file resides) is supplied when the file is preallocated.

NOTE :
SEARCH RULES FOR PASSED FILES (see File Passing)

If the DEVCLASS and MEDIA parameters are missing from the $ASSIGN statement, the following occurs :

1) The system looks for a cataloged file of the given name in the currently attached catalog(s).

2) If there is no cataloged file in this name, the system looks for an uncataloged file passed from a previous step.

3) If there is no passed file of this name, the system assumes it is a RESIDENT uncataloged file.

Assignment of Cataloged Files

The extended JCL statement $ATTACH is used to specify an ordered list of catalogs which are to be searched for a particular catalog object(s), particulary when those objects are files. If the only catalog to be searched is the SITE.CATALOG, the $ATTACH statement may be omitted. If other catalogs are to be searched in addition to the SITE.CATALOG, the SITE.CATALOG must be explicitly declared in the $ATTACH statement. The $ATTACH statement is an extended JCL statement and appears outside a STEP enclosure.

```
$JOB USER =   RT,PROJECT = ZONE ;

ATTACH        CATALOG1  = .CATALOG ;
              CATALOG2  = DEPT. CATALOG,
              CATALOG3  = SITE. CATALOG,
              CATALOG4  = INV. CATALOG ;

STEP RTS, LMLIB ;

ASSIGN CT1, .TOWN.STREET ;

   .
   .
   .
   .
ENDSTEP ;
$ENDJOB ;
```

The private cataloged file ZONE.TOWN.STREET is to be accessed by step RTS. The catalogs are searched in order site catalog, Catalog 2, catalog 3, catalog 4 until the object describing the file ZONE.TOWN.STREET is found. Note that the PROJECT= ZONE is automatically prefixed to the ifn so that the full file name need not be stated in the ASSIGN statement.

NOTE : The catalog object contains all volume and device information.
The search path stated in a $ATTACH can be overridden by use of CATALOG parameter in $ASSIGN. For example :

```
$JOB .....
   $ATTACH    CATALOG1 = DEPT ;CATALOG ,

              CATALOG2 = SITE.CATALOG ,

              CATALOG3 = INV. CATALOG ;

$STEP   RTS,LM-LIB . . . ;

   $ASSIGN CT1, .TOWNSTRETT, CATALOG = 3 ;
      .
      .
   $ ASSIGN ..........;
      .
      .
$ENDSTEP ;
```

The catalog INV. CATALOG only is searched for file ZONE. (TOWN.STREET). If it is not found, the step is abnormally terminated.

FILE ALLOCATION AND PREALLOCATION

The $ALLOCATE and $PREALLOC (Extended JCL statement) can be used to allocate space for either permanent or temporary uncataloged disk or tape files. The $ALLOCATE statement must be used in conjunction with a $ASSIGN in the same $STEP enclosure. The $PREALLOC is used outside a $STEP enclosure.

NOTE : $ALLOCATE, should never be used for cataloged files (refer to Catalog Management Manual).

Temporary Disk Files

Temporary disk files can be allocated space by means of the $ALLOCATE statement associated with the $ASSIGN which defined the file status as temporary. One cylinder is available for a temporary file in the absence of $ALLOCATE. Temporary file organizations can only be BFAS direct, BFAS sequential or UFAS (all organizations). $PREALLOC can be used in cases where unsifficient options are available in $ALLOCATE to completely describe the required temporary file.

The following general points should be noted :

- A temporary disk file cannot exist after the end of execution of the step that created it (unless it is passed to a later step see Section IV). Once created, a temporary disk file cannot be given permanent status (although the contents can always be copied into a permanent file under program control).

- Space for a temporary file is only reserved when the file is opened by a processing program or utility. Consequently, if a temporary file is not opened during the execution of a particular job step, the space will not be allocated.

- By default, temporary disk files are deallocated by the system at the end of the job step in which they are created and used (i.e. privilege of access to the file, and device, are removed and the file label is destroyed). A temporary disk file can be prematurely deallocated during job step execution by closing the file with deassign - CLOSE WITH LOCK in COBOL. If a temporary disk file is required for more than one job step, it can be passed to a subsequent step under the direction of each $ASSIGN by END=PASS for example (see Section IV, File Passing).

- If an «increment size» is specified in a $ALLOCATE for a BFAS or a UFAS sequential disk file, the size of the file will be increased dynamically by the specified amount whenever a write operation in the current job step cannot be performed because the file is full.

Permanent Disk Files

Space reservation for permanent disk files can be done in one of two ways :

- As a special operation before any use of the file. This is a disk file preallocation. It is the recommended procedure for all permanent disk files. For cataloged disk files, the catalog entry must be created using $CATALOG before the file is preallocated (see Catalog Management).

- As part of the first open operation on the file (as for temporary files). This is a dynamic allocation of a disk file. The mechanism can be used only for sequential or direct BFAS files or UFAS files (any organization).

Once allocated, a permanent file will continue to exist after the execution of the job. Under normal circumstances when the file is no longer required, the space must be deallocated under explicit user control, by use of the $DEALLOC utility. Volume preparation (using the Data Management utility $VOLPREP) on the volume containing the file will also perform this function.

At the beginning of each job step in which an existing permanent disk file is assigned, system resources such as access to the file and to the device are given to the job. Unless the file is passed to the next job step (see the Section IV, File Passing), all these resources (excluding the file space itself) are freed at the end of the job step.

However, if a file is closed with deassign (CLOSE WITH LOCK in COBOL), the resources will be freed at the time of file closing and the file cannot be opened again in that job step.

EXAMPLE

```
$JOB HJEX, USER=K1, PROJECT = WASF ;

COMMENT 'THE FOLLOWING JOB STEP CREATES AND USES A TEMPORARY
DISK FILE SCR AND REFERENCES AN EXISTING PERMANENT FILE ABC.
PR' ;

STEP STO1, ABC.LDI;

   ASSIGN FILE1, SCR, TEMPRY, DEVCLASS = MS/M300,

         MEDIA = 12345;

COMMENT  'NOW ALLOCATE 10 TRACKS FOR THIS TEMPORARY FILE';

         ALLOCATE FILE1, SIZE 10, UNIT = TRACK ;

   ASSIGN FILE2, ABC.PR,DEVCLASS =MS/M300, MEDIA=X42 ;

COMMENT  'SINCE RESIDENCY PARAMETERS (DEVCLASS AND MEDIA)
ARE GIVEN THEN THE FILE CONSIDERED TO BE UNCATALOGED'
                              .
                              .
                              .
                              .
   ENDSTEP ;

COMMENT ' AS TEMPORARY FILE SCR OF STEP STOP NO LONGER EXISTS,
THE FILE NAME CAN BE USED IN ANOTHER STEP' ;

   CAN BE USED IN ANOTHER STEP' ;

   STEP STO2, ABC. L02 ;

   ASSIGN NEX, SCR, TEMPRY ;

COMMENT ' AS NO ALLOCATE IS PROVIDED FOR SCR HERE IT WILL
ALLOCATED ONE CYLINDER (ON A RESIDENT DISK )';

   ENDSTEP ;

$ENDJOB ;
```

Comment : the use of CLOSE WITH LOCK in conjunction with Check point/Restart is not advised.

PREALLOCATION OF A PERMANENT DISK FILE

For UFAS and BFAS permanent files, space may be reserved and file labels may be created using the $PREALLOC utility. This utility is described in detail in the

UFAS or BFAS User Guide as appropriate, but its major characteristics are outlined below. $HALLOC, the corresponding utility for HFAS disk files (which can only be permanent) in a Level 64 environment, is described in the HFAS User Guide and is not discussed here.

The $PREALLOC utility gives the user full control over the size and location of the disk space allocated, the file organization, the file attributes such as record size and record format, and the expiration date of the file.

Once a PREALLOC statement has been executed successfully, the file exists, even though it may not have been accessed during the job that created it. The $PREALLOC statement appears within a JOB enclosure.

EXAMPLES :

```
$JOB PREP, USER=PREPF, PROJECT = MKT ;
COMMENT ' CREATE A FILE PREPF. N01 RESERVING 50 TRACKS ON MS/M402
VOLUME C018 RECORD FORMAT FB WITH 80 BYTE RECORDS IN 6400
BYTE BLOCKS' ;
PREALLOC PREPF. N01, DEVCLASS=MS/M401 UNIT=TRACK,
   GLOBAL = (MEDIA=C018, SIZE=50), FILESTAT = UNCAT,
   BFAS = (SEQ =(BLKSIZE=6400, RECSIZE=80,RECFORM=FB) ;
COMMENT 'NO MORE PROCESSING TO BE PERFORMED IN THIS JOB' ;
$ENDJOB ;
```

The $PREALLOC INCRSIZE parameter allows a file to be dynamically extended whenever more space is required, subject to volume limitations. Unlike $ALLOCATE, the effect of INCRSIZE in $PREALLOC is global. Dynamic extension can therefore be dangerous if a program gets into a loop which causes file extension within a step.

ALLOCATION OF A PERMANENT DISK FILE

The $ALLOCATE basic JCL statement can be used in a job step to allocate space for a permanent file, instead of the $PREALLOC utility, but it must have an associated $ASSIGN statement in the same job step. The statement cannot be used for BFAS indexed sequential files (which require $PREALLOC), or for HFAS files. $ALLOCATE is generally used to create temporary disk files.

The $ALLOCATE statement requests disk space in units of track (by default) or cylinder, but without defining a location for the file. In addition, for BFAS sequential, UFAS sequential and indexed, and library files, a dynamic file extension mechanism (INCRSIZE) is available if all the space in a file is fully occupied, or likely to become so.

COMMENT : $ALLOCATE (with INCRSIZE specified) must be present in the step in which the extension is required.

The $ALLOCATE statement enables an optional check mechanism to be used to prevent the overwriting of an already created file, by erroneous allocation to a step.

The execution of a $ALLOCATE statement does not occur until the file is opened for the first time. The $ALLOCATE statement does not supply the external file name (provided instead by an associated $ASSIGN statement in the same step), or the file characteristics (taken from the file description in the processing program or

from an associated $DEFINE statement).

```
$JOB NEWPERM, USER=PREPF, PROJECT=MKT
        STEP LM1, PREPF. COBCR ;
        ASSIGN KDIS, PREPF. NO1,
                DEVCLASS=MS/M402, MEDIA=C018 ;
        ALLOCATE KDIS, SIZE=50, UNIT=TRACK, INCRSIZE=1 ;
        ENDSTEP ;
$ENDJOB ;
```

Assuming the load-module LM1 has been built from the COBOL program :

```
        SELECT MISAJ.
        ASSIGN TO KDIS.
        ORGANISATION IS LEVEL-64 SEQUENTIAL.
        FLR.
        FD KDIS.
        BLOCK CONTAINS 80 RECORDS.
        01 KDIS-REC PIC A(80).
```

a file named PREPF. N01 will be created on volume C018. Its size will be 50 tracks. It will be a BFAS sequential file with fixed blocked records, each record being 80 bytes long, with 80 records per block. One track will be dynamically allocated to this file whenever a write operation would overfill the file during this job step.

COMPARISON OF $PREALLOC AND $ALLOCATE

Table 3-2 shows the main differences between the Extended JCL Statement $PREALLOC and the basic JCL Statement $ALLOCATE.

Table 3-2. Comparison of $PREALLOC and $ALLOCATE

$PREALLOC	$ALLOCATE
Permanent files (cataloged or uncataloged) and Temporary file Job Enclosure Statement	Permanent uncataloged or temporary files Placed inside a step enclosure with associated $ASSIGN
Must be used for indexed sequential	Only allocates sequential or direct files
The number of extents and placement of space can be explicitly declared	Automatic space allocation only.
The maximum number of extents per volume may be restricted by user (MAXEXT)	Up to 16 extents per volume may be allocated, if required
The organization, block size, record size and record format are declared explicitly (BLKSIZE, RECSIZE and RECFORM)	The organization, block size, record size and record format are taken from the program which is executed (or from an associated $DEFINE statement)
Extension of file space must be performed explicitly for sequential disk files	Specifies the space extension to be made if end-of-file on output is reached by the executing step ; applies to sequential files only. CHECK feature for existing files.

Files can only be allocated on disk volumes which have been prepared using the $VOLPREP utility.

It is recommended that $PREALLOC be used for permanent files and $ALLOCATE for temporary files. $PREALLOC must be used if the files are cataloged.

```
$JOB        PJACJ, USER= TP, PROJECT= CMS ;

COMMENT     ' THE NEXT STATEMENT ALLOCATES A PERMANENT
            UNCATALOGED FILE USING THE PREALLOC UTILITY' ;

PREALLOC    CMS.PIX, EXPDATE=800, UNIT=CYL,DEVCLASS=MS/M400,

            GLOBAL=(MEDIA=(BD14, BD15), SIZE=80),

            MAXEXT=2,

            BFAS = (INDEXED=(BLKSIZE=800, RECSIZE=160,

                                RECFORM=FB, CYLOV=2, GENOV=12,

                                KEYLOC=5,KEYSIZE=12)) ;
```

```
STEP        TP47, CMS.LML ;
            ...
COMMENT     'THE NEXT STATEMENTS WILL BUILD A TEMPORARY
            FILE WHICH WILL BE USED IN THIS STEP AND THE NEXT STEP'
ASSIGN      FILA, CMS.TPIX, FILESTAT=TEMPRY, END=PASS,
            MEDIA=BD14, DEVCLASS=MS/M400 ;
ALLOCATE    FILA, SIZE=25, UNIT=TRACK ;
DEFINE      FILA, FILEFORM=BFAS, BLKSIZE=450, RECSIZE=90,
            RECFORM=FB ;
COMMENT     'THE NEXT STATEMENT REFERENCES THE FILE BUILT BY
            PREALLOC ABOVE';
ASSIGN      FILE CMS.PIX,  DEVCLASS=MS/M400,
            MEDIA=(BD14, BD15) ;
            ...
ENDSTEP ;
STEP        TP48, CMS.LML;
ASSIGN      FNP, CMS.TPIX, FILESTAT=TEMPRY ;
            ...
ENDSTEP ;
ENDJOB ;
```

Tape Files

No space allocation is required for tape files except for cataloged files, whether temporary or permanent. At the beginning of a job step system resources, such as access rights to tape drives, are assigned to the step. The file name is written when the tape file is opened in output processing mode (even if the permanent file already exists on the tape volume). Tape naming also occurs when a file which does not already exist on the volume is opened in append processing mode.

EXAMPLE

```
$JOB HJTP, USER = BR, PROJECT = TAX ;
        STEP TRIAL, TAX.RUN ;
                .
                .
                .
        COMMENT ' ASSIGN TAPE FILE TAX.TPERM WITH INTERNAL
        NAME TEST AND UNCATALOGED' ;
        ASSIGN TEST, TAX.PERM ,DEVCLASS =MT/T9, MEDIA=PR16 ;
                .
                .
                .
        ENDSTEP ;
$ENDJOB ;
```

The file label contains information from the $ASSIGN statement (external file name, internal file name, expiry date) and the file definition in the generating program, or $DEFINE, (BFAS, UFAS, HFAS, FOREIGN sequential with characteristics as for disk files).

Unless they are passed from one job step to another, temporary tape files, like temporary disk files, are known to the system only for the duration of the job step in which they are assigned (and opened). In fact temporary tape files are not destroyed automatically by the system since a new file can be created on the tape by overwriting the current one. Note that work tapes, see below, are dissimilar in this respect.

A permanent tape file, cataloged or uncataloged, exists after the job step that created the file terminates. The contents of the file are preserved until a new file is created on the named volume, or until the VOLPREP utility is used on the volume. Note however that the integrity of cataloged tape files is subject to the same security given by the catalog function, as for permanent disk files.

The destruction of a file is subject to file security rules, in particular any expiration date applying to the file (See Section IV).

WORK TAPES

A WORK tape is a tape volume that has been prepared by the data Management utility $VOLPREP (with WORK option). WORK tapes are intended to free the user from the need to indicate the exact name of the tape, particularly if temporary work space is required. When the programmer specifies MEDIA = WORK in a $ASSIGN statement, the operator at execution time is instructed to mount a WORK volume for the job.

The $ASSIGN specifies whether the file to be written is temporary (TEMPRY parameter) or permanent (see $ASSIGN for permanent cataloged and uncataloged files). If a temporary file is requested the tape volume remains a WORK in type. However, if a permanent file is requested, the tape volume loses its WORK status and becomes a normal named volume. The next time the file on tape is used, the programmer must supply the proper volume name, i.e. the volume name of the work tape (displayed in the original job occurrence report). The WORK status of a tape can also be removed using $VOLPREP utility.

EXAMPLE

```
$JOB,....;
        STEP....;
        ASSIGN SCRI, OFF.TEMP, DEVCLASS = MT/T9,
        MEDIA = WORK, TEMPRY;
        ENDSTEP;
        STEP;
        ASSIGN EXTRA, HOME.PERM, DEVCLASS = MT/T9,
            MEDIA = WORK;
        ENDSTEP;
$ENDJOB;
```

A temporary work tape is allocated for the duration of the first step. The tape used in the second step will lose its work status and become permanent.

Tape File Extension

Work tapes can be used for the extension of existing tape files. If, during a writing operation on a normal tape file, the end of the last specified tape is reached, Level 64 GCOS will try to use a work tape to extend the file, rather than abort the step. If no WORK tape is premounted, the system will ask the operator to mount one. The operator may refuse to do so, in which case the writing operation is not performed and the job step will be aborted.

If the existing tape file is a permanent file, the new work tape will lose its WORK status. If a file is passed to a later step, it will be considered as a multivolume file and treated as if the new tape had been indicated in the respective $ASSIGN statement. If a file is not passed, the new tape will not be usable in a subsequent step. The exception to this is for a cataloged file in which case file passing is not necessary.

If the (multivolume) file is used afterwards, the associated $ASSIGN statement must include the new volume name in the MEDIA list. This name will have been indicated in the original Job Occurrence Report.

USE OF MULTIVOLUME FILES

A single file may be spread across a number of volumes up to a maximum of 10 volumes. All the volumes for the file must be of exactly the same type (all disk, same disk type, or all tape, same tape type). The user must always supply volume names, in the $ASSIGN statement, in the same order as they were specified when the file was written. However, if a user requires records from a subset of the volumes of a multivolume tape file (for example in APPEND processing mode) the user may specify only the required volume name(s). This avoids the unnecessary reading of preceding volumes of the file. This is illustrated in Figure 3-1.

For tape files and for HFAS and BFAS disk sequential files, the programmer can indicate how many volumes of a multivolume file are to be mounted simultaneously. This facility, introduced by means of the MOUNT parameter in the $ASSIGN statement, can be helpful in reducing device requirements. Details are given under the paragraphs entitled Device Management.

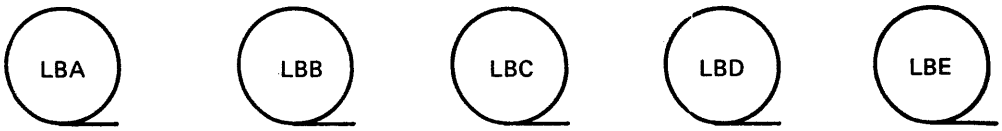
Multivolume files can be temporary or permanent.

Multivolume Work Tapes

A multivolume tape file can consist entirely of WORK tapes. If MEDIA=WORK is indicated in $ASSIGN, the system will automatically use as many WORK volumes as are required. The sequence in which they are used will be listed on the Job Occurrence Report and these names will then have to be used in references to the file in subsequent jobs.

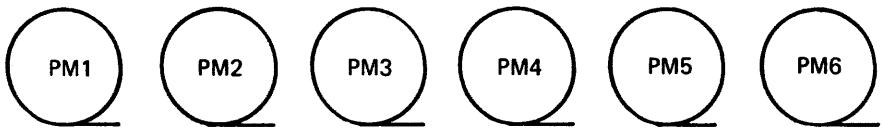
It is not possible for a file to reside partly on normal tapes and partly on work tapes. (Note that a work tape once used for normal tape extension becomes a normal tape itself).

File FNAL. A



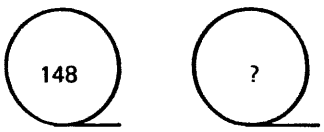
program only reads records within volumes LBC and LBD. Does not read to end-of-file so LBE not needed

File HMQC. 41



PM4 is the last volume currently used. This file will be opened in APPEND mode and the user wishes that expansion occurs on reserved volumes PM5 and, later, PM6.

File NCU.BX



File NCU.BX, opened in APPEND mode, is to grow using work volumes. Currently, only one volume, 148, accommodates the file.

```
.
.
STEP        GROFIL MY.LMLB, DEVCLASS=MS/M400, MEDIA=MSD ;
ASSIGN      FLA, FNAL.A,   DEVCLASS=MT/T9, MEDIA=(LBC,LBD) ;
ASSIGN      FLB, HMQC.41  DEVCLASS=MT/T9,
            MEDIA=(PM4, PM5, PM6) ;
ASSIGN      FLC, NCU.BX, DEVCLASS=MT/T9, MEDIA=148;
ENDSTEP;
.
.
```

Figure 3-1. Partial/extensible Multivolume Processing

Example :

```
COMMENT ' THE NEXT STATEMENT ASSIGNS A MULTIVOLUME DISK FILE' ;

ASSIGN FILA, MST.PLN, DEVCLASS=MS,M350

   MEDIA=(VOL1, VOL2, VOL3) ;

COMMENT'THE NEXT STATEMENT ASSIGNS A TAPE FILE WHICH IS TO BE
WRITTEN ON A WORK TAPE OR WORK TAPES' ;

ASSIGN FILB, N.MSTPLN, DEVCLASS=MT/T9,

   MEDIA=WORK, EXPDATE= 340 ;
```

Note that in the second $ASSIGN statement, the EXPDATE parameter ensures that the file N.MSTPLN will be retained for 340 days. Expiry settings are described in Section IV.

MULTIFILE TAPE VOLUMES

A tape volume may contain one file (a monovolume file), part of a file (multivolume file) or it may contain more than one file, in which case it is known as a multifile volume.

There are parameters of the $ASSIGN statement which are specific to the processing of multifile tape volumes. They are the END, ABEND, and FSN parameters.

With the END and ABEND keywords, there is the special value of LEAVE, which ensures that a multifile tape volume is left positioned at the start of the next file on the tape when processing of the current file is finished. If this is not specified, the tape would normally be rewound after each file is processed.

The FSN parameter must be specified for multifile tapes, and specifies the file sequence number of the file to be assigned. Sequence numbers of files start at 1. There are two special values for FSN, 254 and 255.

If FSN=255 is specified, the tape will be searched for a file of the specified name at file open time. Note that if the processing is in output mode, the existing file will be over-written. If there is no file and processing is in output mode, a new file will be created after the last file on the tape.

To avoid problems with possible overwriting of existing files of the same name when processing in output mode, the value of 254 can be given for FSN. This value causes the file to be written after the last file on the tape regardless of whether a file with the same name already exists on the tape.

Cassette Files

Cassette files can be treated in the same way as tape files. The device-class for cassettes is CS.
The format of cassette files is fully described in the BFAS and UFAS User Guides.

File Concatenation

Several BFAS sequential files, UFAS standard sequential or cassette files can be accessed as if they were a single sequential file. File concatenation, as it is called, is specified by successive $ASSIGN statements in required sequence, with the omission of the internal-file-name of all but the first $ASSIGN. For example :

```
$STEP .... ;

    ASSIGN TOTO, MY.FILE1, DEVCLASS=MT/T9, MEDIA=A1 ;

    ASSIGN ,MY.FILE2, DEVCLASS=MT/T9, MEDIA=A2 ;

    ASSIGN ,MY.FILE3, DEVCLASS=MT/T9, MEDIA= A3 ;
```

In this example the three uncataloged tape files are treated as a single file with an internal file name TOTO. The file starts at MY.FILE1 and ends with MY.FILE3. Concatenated files must have compatible record and block sizes and have the same device class and device attributes.

FILE SPACE RE-ASSIGNMENT

The release of space occupied by an outdated file, to allow a new file to be created, is achieved in different ways, depending on the type of file i.e. disk file, (cataloged or uncataloged), or tape file (cataloged or uncataloged) as follows.

Uncataloged Tape files

No particular action is required to reassign the tape.

Cataloged Tape files

A cataloged tape file must be deallocated using $DEALLOC.

Uncataloged Disk files

The space is released by means of a $DEALLOC (or $HDEALLOC for HFAS files). The volume preparation utility program (VOLPREP) makes it possible to release the space of all files on a volume.

Cataloged disk files

The space is released as for uncataloged files but the file name and description will remain in the appropriate catalog until it is systematically deleted by use of UNCAT statement (see Catalog Management).

NOTE : The above methods, where appropriate, are subject to expiration date checks.

The deallocation of space can only be achieved prior to an expiration date by a BYPASS parameter in the following manners

- BYPASS in VOLPREP, HVOLPREP in addition to operator consent for disk files.
- DEALLOC and HDEALLOC
- VOLSCRAT for UFAS or BFAS tapes.
- In the case of cataloged files which are preallocated and not known by the catalog (when a system crash occurs for instance), the $DEALLOC statement must have the FORCE parameter and DEVCLASS, MEDIA information.

$DEFINE Overview

The $DEFINE statement allows the user to supply information on a file this information can override program supplied information and/or supply information that is otherwise not available. It may not be available either because the file label does not contain it (for example, the label does not exist), or because the program does not contain it (for example, the language does not allow this information to

be given).

$DEFINE is always associated with an internal-file-name, and if the same file is assigned to different internal-file-names, there may be a $DEFINE for each assignment.

The information provided by $DEFINE sets up the file characteristics when the file is opened. This information overrides any file-description in the program, and contents of the file label will override the $DEFINE information. The exception to this is when an non-native tape file is indicated (FILEFORM=NSTD), when any file labels are ignored. The GCOS Level 64 override rules are described in this section.

The following information can appear in $DEFINE :

- block size and record size
- recording format
- file format
- number of buffers and number of blocks per buffer
- the inclusion or omission of block sequence numbers
- the occurrence or not of read after write check
- the residency of the index for Indexed Sequential files
- key position and size
- control interval and control area size (UFAS only)
- control interval and control area free space (UFAS only)
- the frequency with which checkpoints are taken
- unit record device control options
- file journalisation can be requested

For HFAS files :

- tape format (H200 form)
- EBCDIC code conversion/non-conversion
- date code
- filler character
- flag character

For non-native tape files :

- the function mask to control processing of the tape

GCOS Level 64 Override Rules

When specified, and if $DEFINE does not indicate a non-native tape file, the file label provides, refer also to figure 3-2 :

- file configuration
- record length
- block size
- for UFAS and BFAS tape files, the specification or omission of block sequence numbers
- size and location of the key

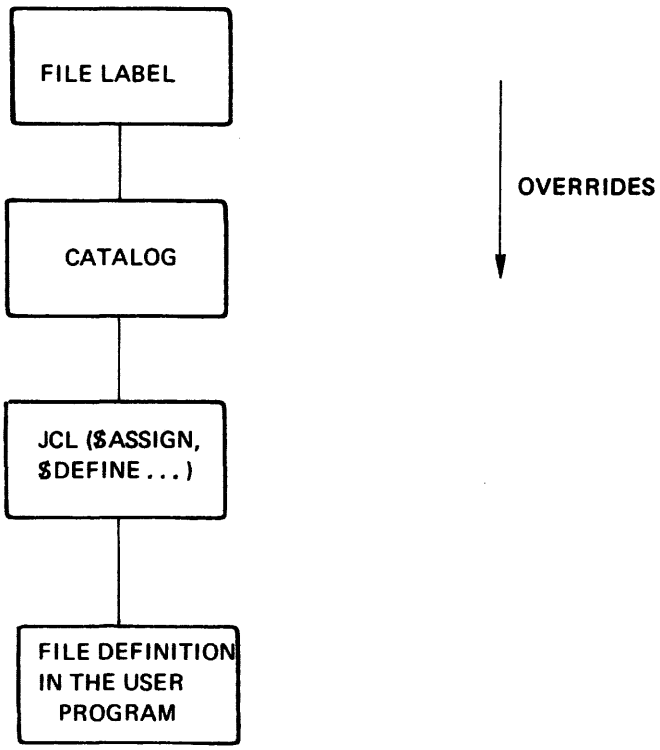


Figure 3-2. GCOS OVERRIDING RULES

- whether deleted records are to be allowed or not
- CI and CA size and available space (UFAS)
- whether the file is catalogued or not

$ASSIGN provides :

- the external file name
- the label type
- the name of the volumes and the type of device on which the file resides
- the level of sharing and access allowed to the file
- whether the file is temporary, permanent uncatalogued or permanent catalogued
- whether the file is multivolume

The file definition in the program provides the other features, namely :

- access mode
- number of buffers
- move or locate mode
- code set used for data storage
- all label information when the label is not present

The file label is considered to be missing for :

- tapes without labels (LABEL = NONE)
- files which have to be generated

GENERATION GROUP CREATION ACCESS, AND DELETION

A generation-group is a set of cataloged files that are chronologically or functionally related. Each member of the group is called a generation. Level 64 GCOS supports update generations, which are different versions of a master file. Update generations can be processed in a «closed loop» way, that is, when the newest generation is assigned, the media and the file space of the oldest generation are used for creating the newest generation.

All the generations in the group are referred to by a common name, but each generation has a specific suffix joined to the common name. This suffix uniquely defines the generation. The suffix may be :

- an absolute generation number (e.g. *G0024)
- a relative generation number (e.g. *G+00)
- a symbolic generation name (e.g. *G_MARCH)

The file label only contains the absolute generation number, which may be between 1 and 9999, and will be incremented by 1 each time a generation is created. When generation 9999 is reached, the cycle continues with 1. Conversion from relative to absolute is performed automatically by the system.

The symbolic generation name is a name given by the user to an absolute generation number, and enables you to keep track of the contents of a file. For example, DEPT1. INVENTORY*G_MARCH could refer to the state of an inventory file in March. If the absolute number of that generation was *G0456, the file label would contain this absolute number, and the catalog entry would contain the symbolic generation name.

The relative generation number identifies the position of a specific generation within the generation group, relative to the current generation. Thus,

*G+ 00 refers to the current generation

*G–01 refers to the generation preceding the current one

e.t.c

A reference to generation *G+ 01 indicates that a new generation is to be created, and the generation group is to be rotated (see $SHIFT, later in this section). After the shift of the generation group, the new generation will become the current one.

NOTE : Reference to *G+ 02, *G+ 03, etc. is not allowed, and a generation group may only contain a maximum of 32 generations.

When a relative generation number is given, the catalog finds the absolute (and the symbolic generation name if present) of the generation, according to the position of the current generation.

A generation group can be represented as follows :

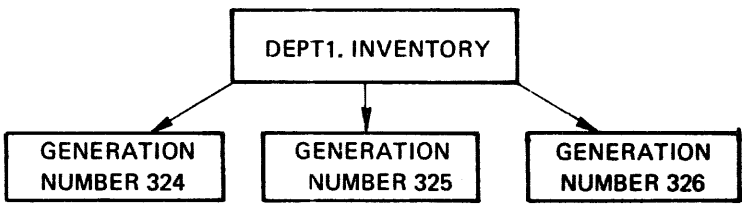


Figure 3-3. A file with Three Generations

The catalog description for each generation contains :

- The generation name (the absolute generation number and, possibly, the symbolic generation name).
- The volume list on which the generation exists.
- Security lock information
- Statistical information, including the date of preallocation, the current size of the file with its unit of allocation, and the date of last updating.

At generation group level, the catalog maintains a description which consists of :

- Sharing information ; the maximum sharing allowed for all the generations.
- The number of generations (NBGEN).
- The increment size and unit of incrementation to be used for all generations.
- The abort conditions applicable to all generations.
- The retention period as a number of days. This information is to be used when you want to give an expiry date to the generations. This expiry date is automatically stored away at each cycling of generations by applying the formula :

expiration-date = date-of-the-day + retention-period

This mechanism may be bypassed by specifying an expiration date in the $ASSIGN statement; note that if this is done, the JCL required to perform the update will have to be changed periodically, and thus one of the advantages of using file generations is lost.

- The bypass expiration-date switch. This switch is used as follows:

  when set, the creation of the newest generation on the media of the oldest will not check the expiration date; the oldest generation will be erased whether it has expired or not.

  when reset, the expiration date of the oldest generation will be checked before creation of the newest generation. If it is not expired, the open will abort with the return code NOTOBS.

- Statistical information such as the date of cataloging or the date of the last modification to the catalog description.

The following sub-sections deal with creation of generation groups, how generations are accessed, and how generations are deleted.

Generation Group Creation

Suppose that we want to create a generation group of three generations called DEPT1. INVENTORY on disk. The JCL that must be used is :

```
(1)   CATALOG DEPT1, TYPE = DIR;
(2)   CATALOG DEPT1. INVENTORY, NBGEN = 3;
(3)   PREALLOC DEPT1. INVENTORY *G1, FILESTAT = CAT
                    ,GLOBAL = (MEDIA =C161,SIZE =1), . . . ;
(4)   PREALLOC DEPT1. INVENTORY *G2,FILESTAT = CAT,
                    GLOBAL = (MEDIA = C162, SIZE = 1) . . . ;
(5)   PREALLOC DEPT1. INVENTORY*G3, FILESTAT = CAT,
      GLOBAL = (MEDIA =C163, SIZE = 1) . . . ;
(6)   CATLIST FROM = DEPT1 ;
```

In this example, statement (1) creates the master directory DEPT1. The generation group DEPT1. INVENTORY is created with three generations by step (2). The generation group description is filled with the information supplied by $CATALOG, that is :

- number of generations = 3
- maximum sharing applicable to generations = NORMAL
- no increment size
- no abort conditions
- no retention period

Statement (3) creates generation G0001 with the following attributes :

- absolute generation number is 0001
- there is no symbolic name
- the size is one cylinder

– the media for the generation is C161

Statement (4) creates generation G0002 with an identical description to G0001 except for the media, which is C162, and statement (5) creates generation G0003 identically on media C163. Statement (6) gives a list of the tree-structure from the master directory DEPT1. The above process can be represented symbolically as follows :

AFTER STATEMENTS (1) and (2)

ABSOLUTE GENERATION NUMBER	0000	0000	0000
RELATIVE GENERATION NUMBER			
MEDIA LIST			

AFTER STATEMENT (3)

ABSOLUTE GENERATION NUMBER	0001	0000	0000
RELATIVE GENERATION NUMBER	0		
MEDIA LIST	C161		

AFTER STATEMENT (4)

ABSOLUTE GENERATION NUMBER	0002	0001	0000
RELATIVE GENERATION NUMBER	0	– 1	
MEDIA LIST	C162	C161	

AFTER STATEMENT (5)

ABSOLUTE GENERATION NUMBER	0003	0002	0001
RELATIVE GENERATION NUMBER	0	– 1	– 2
MEDIA LIST	C163	C162	C161

NOTES :

1) The preallocation of the generation must be done as specified in the example. Generation 2 cannot be preallocated before generation 1. Symbolic generation names may be given at preallocation time, but if relative generation numbers are used, the *G+01 must be used.

2) If a $PREALLOC of generation *G4 is issued, an abort will occur, as the newest generation always uses the space of the oldest one which already exists.

3) It is not necessary to allocate space for all three generations at once. The space for the first one may be allocated, the generation may be processed and then the space for the following ones may be allocated.

4) The JCL is the same if the generations are on tape files, except for the device class and media.

5) Instead of $PREALLOC, you may use $REPLACE, but in this case, tape files will be duplicated, as the name of an uncataloged tape file is different to a generation name. The example might become :

```
$REPLACE INFILE = (DATA, FILESTAT =UNCAT,MOUNT =1,
                   MEDIA=(1894,1895), DEVCLASS=MT/T9/D1600),
        OUTFILE = (DEPT1. INVENTORY*G1, FILESTAT = CAT,
                   MOUNT=1, DEVCLASS=MT/T9/D1600,
                   MEDIA= (1896,1897) ) ;
```

Note that as for $PREALLOC, a $REPLACE on generation *G4 will cause an abort.

6) The first generation must be *G1, but the number can be changed later using the $MODIFY statement.

7) To make the JCL easier, generation numbers such as *G0001 can be shortened to *G1 (leading zeroes can be suppressed).

8) A generation group may be created on a private catalog or the site catalog. Figure 3-4 shows the effect of creating o three generation group on the site catalog.

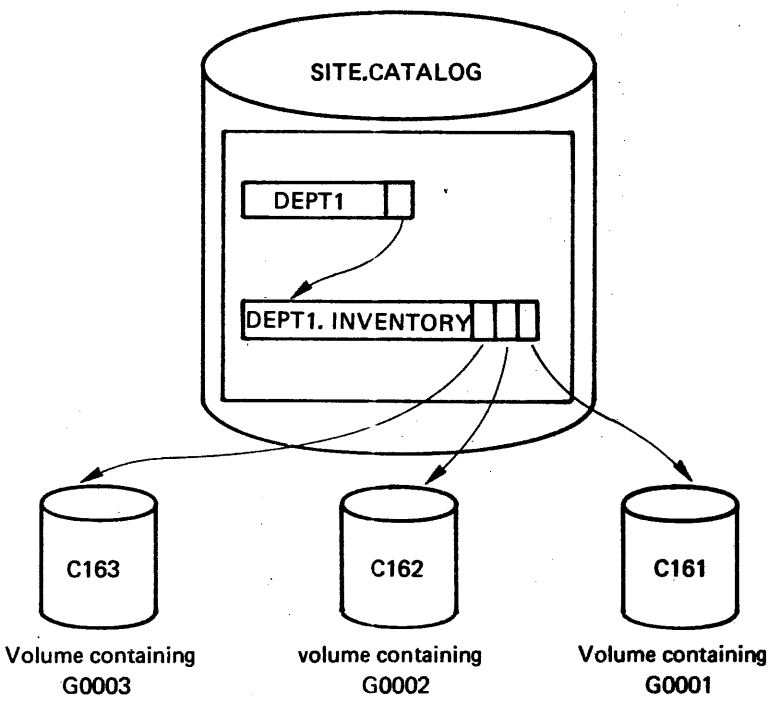


Figure 3-4. The SITE.CATALOG After Three Generations Have Been Created

Creating the next Generation

The position in the generation group created in the previous example after a $ASSIGN of DEPT1. INVENTORY *G4 is :

Absolute Generation Number	0004	0003	0002	0001
Relative Generation Number	+ 1	0	− 1	− 2
Media list	C161	C163	C162	C161

After a successful OPEN in output mode, generation *G1 is not accessible, as it is over written with generation *G4 file label. The current generation is still *G3. The act of shifting generations is entirely the responsibility of the user, that is the user must :

1) decide when *G4 is to be considered as valid

2) modify the correspondance between relative and absolute generation numbers.

In other words, the user determines when *G4, *G3, *G2 become equivalent to *G+ 00, *G-01, *G-02. The system will never perform any automatic shifting, and you must use the extended JCL statement $SHIFT (described later in this section) for this purpose. Shifting time may be decided arbitrarily or according to instructions given by the project manager.

Note that concurrent access to *G1 and *G4 is not allowed. When two jobs try to concurrently assign these two generations, enqueuing will be performed by the system. After dequeuing, the dequeued job cannot reference G1 if *G4 has been

created in the mean time. If *G4 has not been created (for example, the step with $ASSIGN *G+01 has aborted), then *G1 is still accessible.

If *G1 and *G4 are assigned within the same step, or if *G1 is passed by a preceding step and *G4 is assigned in the following step, an abort will occur with the following message appearing in the JOR :

DS09 efn OLDEST AND NEWEST GENERATIONS MAY NOT BE ASSIGNED CONCURRENTLY

For example, these two jobs will abort :

```
$JOB USER = JONES, PROJECT = PR1;
    STEP UPDATE, . . . ;
    ASSIGN ifn1, DEPT1. INVENTORY*G1;
    ASSIGN ifn2, DEPT1. INVENTORY*G4;
    ENDSTEP ;
$ENDJOB ;
$JOB USER=HUGHES,PROJECT=PR2 ;
    STEP . . . ;
    ASSIGN ifn1, DEPT1. INVENTORY*G1, END=PASS;
    . . .
    ENDSTEP ;
    STEP UPDATE, . . . ;
    ASSIGN ifn1, DEPT1. INVENTORY *G4;
    ENDSTEP ;
$ENDJOB;
```

The next example shows how $SHIFT may be used :

```
(1) $JOB UPDATE, USER = BOB ,PROJECT = DEPT1 ;
(2) DEALLOC .INVENTORY*G-2;
(3) PREALLOC .INVENTORY*G-2, FILESTAT=CAT,DEVCLASS=MS/M400,
                GLOBAL = (MEDIA =NEWVOL, SIZE = 2), . . . ;
(4) STEP . . . . ;
(5) ASSIGN ifn1, INVENTORY [*G+0];
(6) ASSIGN ifn2, .INVENTORY | *G+0 |;
(7) ENDSTEP ;
```

(8) STEP . . . ;

(9) ASSIGN ifn, .INVENTORY *G +1;

(10) ENDSTEP ;

(11) SHIFT INVENTORY ;

(12) $ENDJOB ;

All the information contained between [ ] is optional.

This job first reallocates the space for the oldest generation, *G-2, on a volume called NEWVOL. This is done by steps (1) and (2).

NOTE : These steps are not mandatory.

In step (4), the current generation, *G+ 0, is used to create the newest generation, *G+ 01, according to statements (5) and (6).

Step (8) inspects the newly created generation.

Note the uniformity of naming in statements (5) and (9) and note that the JCL in statement (5) can be applied to any generation.

Step (11) shifts the generations to make the new generation the current one.

The oldest generation is not available for access.

If a symbolic namewas required for the new generation, the $ASSIGN statement (5) could have been ASSIGN .INVENTORY *G_DEC, where DEC is a symbolic name that is unknown to the catalog. At OPEN time, the labels will be modified to contain the new generation. For tape files, the new file label is written dynamically on the volumes as they are accessed. For disk files, the labels will normally be changed at OPEN time if all the volumes are mounted.

When the newest generation is created, the OPEN must be performed in OUTPUT mode, otherwise an abnormal return code, EFNUNKN, will be issued. The OPEN will only be successful if the expiry date of *G-2 is over, or if the «ignore expiry date» option is used.

The state of the generation group before and after the example job can be shown pictorially as follows :

BEFORE $ASSIGN DEPT1. INVENTORY *G+ 1

Absolute generation number	0003	0002	0001
Relative generation number	0	– 1	– 2
Media list	C163	C162	C161

AFTER $SHIFT

Absolute generation number	0004	0003	0002
Relative generation number	0	– 1	– 2
Media list	C161	C163	C162

NOTE : Relative generation numbers must be used with some caution. Because Level 64 has a multiprogramming environment, a concurrently running job might create a new generation, which will then cause a wrong retrieval. Consider the following example :

```
        JOB 1                              JOB 2

STEP . . . ;                       STEP . . . ;

ASSIGN ifn1, .INVENTORY*G-01 ;     ASSIGN ifn2, .INVENTORY*G+01 ;
. . .                              . . .
ENDSTEP ;                          ENDSTEP ;
```

If JOB 2 causes a generation shift before JOB 1, the latter will retrieve a wrong file. For that reason, some degree of control must be maintained over the execution of job steps referring to generation with relative generation numbers. File Management cannot ensure such a synchronization. The only thing that can be done to avoid this position is to either «security lock» the *G+ 01 or to synchronize the scheduling of the two jobs.

PREFIXING

To reduce the amount of writing required in JCL statements, external-file-names can be shortened by taking advantage of prefixing. This can be done in two ways, described below.

Automatic Prefixing

If the files associated with a project are all given external-file-names that start with the project name, the first component name of the files can be omitted. This requires that the master directory for these files has the same name as the project. When this is done, the external-file-name starts with the concatenation character (.), and the system automatically supplies the first component name. For example, if a job is running under the project DEPT1, and the master directory for all files associated with this project is DEPT1, the file DEPT1. SECT2. INVENTORY could be accessed by the name .SECT2 .INVENTORY, with the system automatically providing the name DEPT1. The previous example of $SHIFT shows the use of automatic prefixing.

Automatic prefixing only allows you to omit the first component name of the external-file-name. The second method of prefixing, described below, allows you to omit as many component names as you want to.

Prefixing Using the $PREFIX statement

The $PREFIX statement (described in the JCL Reference Manual) allows you to define a prefix for all files. Each $PREFIX statement is valid until it is overridden by another $PREFIX statement. For example, if within a job there was frequent reference to files whose first two component names were DPET1 and SECT2, (DEPT1. SECT2. INVENTORY, DEPT1. SECT2. SALES, and DEPT1. SECT2. SALES, and DEPT1. SECT2.PAYROLL, for example), the first two component names can be omitted from references to the files if the statement

```
$PREFIX DEPT1. SECT2;
```

occurs before the references to the files.

When accessing the files, the only names that be given are .INVENTORY, SALES, and .PAYROLL, as the system will automatically supply the defined prefix.

NOTE : The prefix must always be taken into account when calculating file name lengths.

The $SHIFT Utility

The $SHIFT statement is used to make the latest generation of a cataloged file the «current» generation and thus to discard the oldest generation.
This utility will shift a generation group when the file is valid, that is, when :

- the newest generation contains a valid media list ;
- the newest generation name created in the catalog is identical to the one which is stored in the file label.

When the FORCE parameter is given, the consistency of the catalog information and the file label will not be checked before generation shifting, so no media mountin will be requested. After a generation shift, the oldest generation will be erased from the catalog.

Examples :

1. SHIFT DEPT1. DATABASE;

The generation group DEPT1. DATABASSE is to be shifted if it satisfies the conditions stated above.

2. SHIFT DEPT1. PAYROLL, FORCE;

The generation group DEPT1. PAYROLL is to be shifted without any checks on catalog and file label information consistency.

Deletion of a Generation Group

If you want to delete the generation group created in the previous examples, the JCL would be :

```
(1)  DEALLOC INFILE = DEPT1.INVENTORY*G+00;

(2)  DEALLOC INFILE = DEPT1.INVENTORY*G-01;

(3)  DEALLOC INFILE = DEPT1.INVENTORY*G-02;

(4)  UNCAT DEPT1. INVENTORY, TYPE = FILE;
```

Statements (1), (2) and (3) will delete the filespace occupied by the different generations in the group, and statement (4) deletes the generation group name and entry from the catalog.

NOTES :

1) The deallocation may be performed in any order, but must be performed before the uncataloging.
2) Deallocation of generation *G+ 01 is forbidden, so if this generation exists, a $SHIFT must be given before it can be deallocated. If it does exist, statement (3) will abort because the external-file-name of the generation is not found on the specified media.
3) The uncataloging (4) is not necessary if the user wants to create another generation group with the same characteristics.
4) If the expiration date of the generations is not passed, the BYPASS keyword of $DEALLOC must be used.

4. Resource management

Introduction

GCOS provides the user with JCL facilities which enable the overall system throughput to be optimized by improving the use of resources, such as memory, files and devices. This section discusses the way in which the user can influence the allocation of resources within a step enclosure and from job step to job step.

It may be useful to consider briefly the normal action taken by GCOS immediately before and during step execution.

Before a job step is initiated, the system refers to the user's JCL statements to establish the nature and extent of resources required, and attempts to reserve them. If all the necessary resources are available, they are allocated to the job for the duration of the job step. If one or more resources are not available, the step is kept waiting until they are released by a step of another job. Once all of the required resources have been allocated to the step, its load module is loaded and control is given to the first instruction.

When the step terminates, allocated resources are freed and the JCL processing continues with the next statement of the job description (i.e. the statement that follows the $ENDSTEP). Note that the multiprogramming slot occupied by a job is not released between steps, ans not released if the job is held or suspended (see Section I).

MEMORY MANAGEMENT

The virtual memory concept implemented in GCOS 64, frees the user from problems associated with program structure (e.g. segmentation, transaction sequences), since the user appears to have one large memory for his exclusive use. In a multiprogramming environment memory overload situations can occur when several jobs compete for memory resources. Memory overload causes a general degradation in overall job throughput due to increase in the number of segment transfers between backing store and main memory (see System Management Guide).

Information is made available in the Job Occurrence Report (JOR) from which the user can make a quantitative assessment of the overall processing efficiency of a given job step with respect to memory usage. This information is the SYS MISSING SEGMENT number and PROG MISSING SEGT number (see Section VII, Job Occurrence Report).

The MISSING SEGMENT number indicates the number of system segment and user program segments transfers that occurred in a given step. If the non-resident segments of a program are confined to a small amount of memory, as would occur in a multiprogramming overload situation, then the number of swapping operations would eventually seriously degrade the system throughput. The user, by means of the $SIZE parameter, with the MAXMEM option, see overleaf, is able to vary the amount of memory available to a step. A curve of the sort shown in Fig. 4-1 is generated. There exists on the curve a point such that an increase in the amount of memory available to the step does not give a big decrease in the number of missing segments ; but a decrease in the amount of (memory available produces a sharp increase in the

number of missing segments. This value corresponds to the declared. working-set value stated in $SIZE.

The user is able to state a given memory size that must be available to a job step before step execution can begin.

If the system can fulfill the stated memory value, as well as other system resources (e.g. devices, media), then step execution can be initiated. If the available memory is inadequate, the step is «WAITING FOR RESOURCES» as it would be for any other system resources.

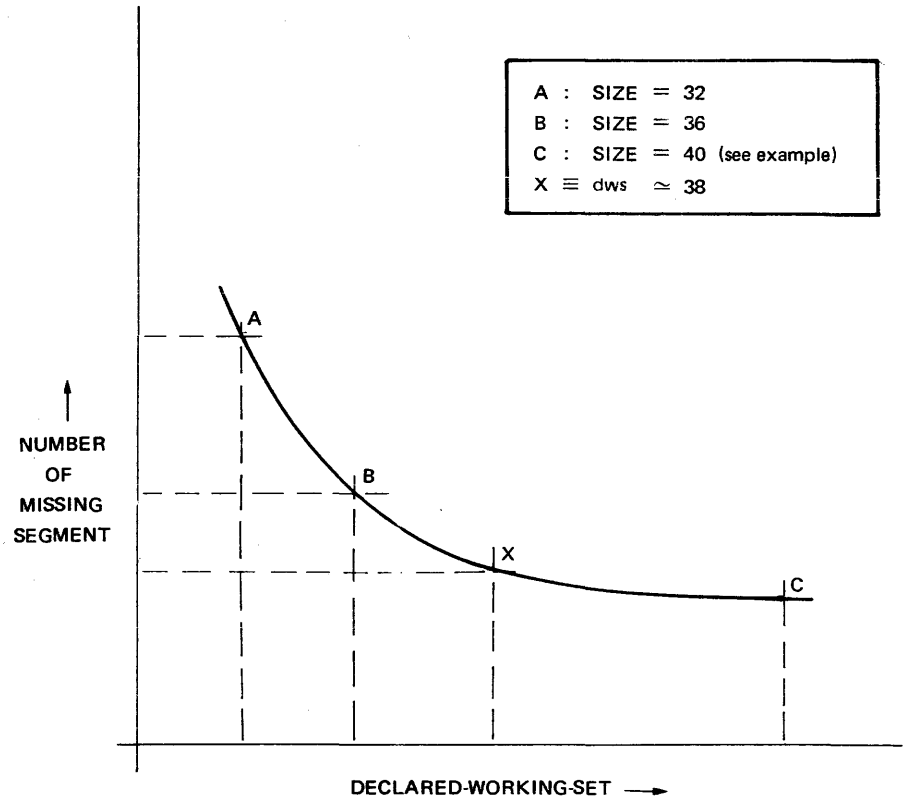
The $SIZE parameter within a step enclosure enables the user to state the memory requirements of a job step.

$SIZE Parameter

The $SIZE parameter is used to specify a declared-working-set (DWS) value for a job step. The DWS value is the total amount of memory resident, (process control structures, buffers), and non-resident (i.e. code and data segments), required for a program to execute in the most efficient manner in a given processing environment. For example:

```
STEP S03, LM.ET ;
SIZE 40 ;
ASSIGN . . . . . ;
   .
   .
   .
ENDSTEP.;
```

In this example the declared-working-set value, 40(1K bytes) of main memory must be available before step execution can be initiated. The sum total of the DWS values (35K by default) for job steps currently executing must be less than the physical memory size.



The mean working set value is the optimum value for working set and is used in $SIZE as declared-working-set.

Figure 4-1. Example of missing segment number plotted against declared-working-set

Estimation and tuning of DWS value

Information appears in the linker listing and JOR which enables an initial estimate to be made of the DWS value.

The linker listing gives the sizes of process control structures, user code and data segments, and run-time package segments (refer to COBOL User Guide).

The JOR gives the number of buffers used, channel program page size and the number of missing segments.

For further information refer to the System Management Guide.

MAXMEM and MINMEM

The MAXMEM parameter of $STEP can be used when tuning the DWS value ; its use should be discontinued when the optimal size has been obtained.

The MAXMEM facility ensures that

- the amount of memory allocated is never more than the DWS value specified in $SIZE
- the system does not attempt to execute the step if the amount of physical memory available is not greater than or equal to the DWS value, even if the step could be run on less, i.e. the step does not benefit from the gradual release of memory resources as the system load decreases.

The optimum value for DWS can be obtained simply by declaring the initial estimate in $SIZE and using MAXMEM :

```
STEP  OPTMEM,  LO.ABC,  MAXMEM ;
      SIZE  40 ;
      ASSIGN. . . . ;
      .
      .
      .
ENDSTEP ;
```

In this example the step OPTMEM executes with 40K bytes of memory reserved to it. The JOR missing segment value should be noted.

If the figure is large, the DWS value should be increased and the associated missing segment figure noted. In this manner a curve can be generated (see Fig. 4-1) and the optimum value (mean working set) estimated.

WARNING : The use of MAXMEM can degrade overall system performance.

Use of MINMEM

The MINMEM parameter of $STEP is used in conjunction with $SIZE and applies only to communications applications ; it guarantees, for the duration of the current step, memory resources equal to the DWS value, irrespective of the current system load. Use of MINMEM helps to ensure that a reasonable response time can be obtained even when the system load is high. Note that MINMEM should be used with discretion since it reserves memory for the step whether or not the step requires it.

FILE PASSING

Generally all resources are allocated at the beginning of each step and released at the end of the step. With respect to files, this situation may lead to problems when successive steps are planned to work on the same file. Between the end of one step and the beginning of the next, another step in a concurrent job could gain access to this file and modify it, jeopardizing the work performed by the first job. If the file in question is a temporary disk file, it would be deallocated at the end of the first step and the subsequent step would not be able to work on it at all.

Files may be passed from one job step to a later one in order to overcome these problems. It is managed using the END and ABEND parameters (with value PASS) in a $ASSIGN statement. END specifies file passing for normal termination of load module execution, ABEND for abnormal termination.

Consider first of all a situation where file passing is not used :

```
STEP      ST01,  ABC.LD1 ;
ASSIGN    FILE1,  ABC.SCR,  TEMPRY ;
ASSIGN    FILE2,  ABC.PR,  DEVCLASS=MS/M300,
          MEDIA = (DX143, DX127) ;
.
.
ENDSTEP ;
```

```
STEP      ST02, ABC.LD1 ;
ASSIGN    JACK, ABC.SCR, TEMPRY ;
ASSIGN    JILL, ABC.FR, DEVCLASS=MS/M300,
          MEDIA = (DX143, DX127) ;
.
.
ENDSTEP ;
```

In a multiprogramming environment there is always the danger of the permanent file ABC.PR being used by another job between the execution of program ST01 and ST02. In addition, the temporary file ABC.SCR in the first step has no particular relationship with the file of the same name in the second step.

Consider the following modification of the above example :

```
.
.
STEP      ST01, ABC.LD1 ;
ASSIGN    FILE1, ABC.SCR, TEMPRY, END=PASS ;
ASSIGN    FILE2, ABC.PR, DEVCLASS=MS/M300,
          MEDIA = (DX143, DX127), END=PASS ;
.
.
ENDSTEP ;
STEP      ST02, ABC.LD1 ;
ASSIGN    JACK, ABC.SCR, TEMPRY ;
ASSIGN    JILL, ABC.PR ;
.
.
ENDSTEP ;
```

In the modified example, the user has passed both files from ST01 to ST02. Note that in subsequent assignments of a passed permanent file it is not necessary to supply device and volume attributes. Note also that for a passed temporary file, the allocation of space is performed only in the job step within which the file is first assigned (in the above example a default size of 1 cylinder is allocated when the file is first opened in step ST01). The file is not deallocated between steps and so it can be used to pass information from one step to the next. However, the temporary file in the above example will be deallocated at the end of step ST02 as END = PASS has not been specified for the file in that step.

The following general points apply to the passing of files :

- Access to a passed file is reserved to the job, subject to the declared file sharing constraints applying to the file (see File Sharing in this Section), from job step to job step until END=DEASSIGN (or END=UNLOAD) is specified or assumed by default.

- Access to a volume which contains a passed file is not normally reserved. Therefore another job could acquire access to the same volume to use a different file (or to share the same file - see «File Sharing»).
- A file may be passed across job steps which do not refer to the file at all. In these cases, the resources are reserved throughout the job until a $ASSIGN statement that refers to the file is encountered (see the example job TRY below).

The following specific observations apply to temporary files, including temporary WORK files, which are passed :

- The file space and access to a temporary file are reserved while it is being passed, subject to file sharing rules. File passing is therefore the means of ensuring that temporary information produced or used in one job step can be accessed in a later step.
- A device used by a job for accessing a temporary file is not necessarily reserved during file passing, so the volume may have to be remounted when the file is assigned again.

For permanent tape files passed, the device used is kept reserved to the job. This ensures that no unnecessary mounting and dismounting of devices is performed. By being aware of which files are to be used again, the system can give better guidance to the operator.

Example :

```
$JOB TRY,...;
 STEP LM1,...;
 ASSIGN MAN1, SUNDAY.REP,
                DEVCLASS = MT/T7, MEDIA = CH215,
                END = PASS ;
 ENDSTEP ;
 STEP LM2,...;
 ENDSTEP ;
 STEP LM3,...;
 ASSIGN MAJ, SUNDAY.REP ;
 ENDSTEP ;
$ENDJOB ;
```

Suppose the processing program in step LM1 contains the following COBOL statements :

```
 SELECT MAN.
 ASSIGN TO MAN1.
 OPEN MAN.
 CLOSE MAN.
 OPEN MAN.
 CLOSE MAN WITH LOCK.
```

and that step LM3 contains :

```
SELECT AGT
ASSIGN TO MAJ
OPEN AGT
CLOSE AGT
    .
    .
    .
```

If the execution of LM1 terminates normally, the file SUNDAY.REP will be passed on to LM3 without interference from the execution of LM2. No other job will have access to SUNDAY.REP and LM3 is guaranteed access to it (although possibly on another tape drive if in the meantime the drive has been assigned to another job step).

If the execution of LM1 aborts, SUNDAY.REP will not be passed (because ABEND=PASS has not been specified for this file assignment in LM1) even if by use of the $JUMP statement (see Section VI) the job itself is not aborted. Note that in these circumstances the CLOSE. . .WITH LOCK in LM1 prevents any further opening of the file during the execution of this load module but does not prevent the file from being passed.

Deadlock Situation

When files are being passed in a multiprogramming environment, care must be taken to avoid a deadlock situation. This can occur when two programs are waiting on each other to release files. An example of this situation is illustrated in Figure 4-2.

JOB STREAM A	JOB STREAM B
STEP LM5, YY.MA ;	STEP LM3, YY.MB ;
ASSIGN I1, M.I,END=PASS . . ;	ASSIGN J1, M.J,END=PASS . . ;
.	.
.	.
ENDSTEP ;	ENDSTEP ;
STEP LM6, YY.MA ;	STEP LM4, YY.MB ;
ASSIGN I2, M.I . . . ;	ASSIGN J2, M.J . . . ;
ASSIGN 12J, M.J . . . ;	ASSIGN J21, M.I. . . . ;
.	.
.	.
ENDSTEP ;	ENDSTEP ;
.	.

Figure 4-2. File Passing with Deadlock

In the above example, step LM5 in multiprogramming stream A and step LM3 in stream B can run in parallel. However, step LM6, which «owns» files M.I., cannot start execution until it can access file M.J, but will not release M.I (to step LM4) until it has completed execution. Correspondingly, step LM4 cannot start execution until step LM6 has released M.I. Thus each stream is waiting on the other. The only way to resolve this situation is for the operator to terminate one of the two jobs.

FILE SHARING

There are many situations where it is desirable to allow access to one particular physical file (as specified by its external file name) by several jobs running concurrently or within the same step by means of two or more independent internal file names. On the other hand, it is useful in most circumstances to be able to control the simultaneous access to a file in order to prevent, for example, two jobs from modifying a file at the same time.

Access and sharing policy can be controlled in GCOS using the SHARE and ACCESS parameters of $ASSIGN statement, in the case of uncataloged and temporary files. For cataloged files the SHARE parameter of $CATALOG is used to set sharing information and is held permanently in the catalog (see Catalog Management), only the ACCESS parameter of $ASSIGN must be used. Figure 4-3 illustrates the simultaneous access of the same file by two concurrent jobs.

Multiple assignment of a single external file name to different internal file names is shown in Fig. 4-4.

$JOB R1.....	$JOB R2.....
STEP.....; COMMENT 'REQUEST READ ACCESS TO MJ.OMN AN UNCATALOGED FILE, ALLOWING OTHERS READ ACCESS ONLY'; ASSIGN COMP, HJ.OMN, DEVCLASS=MS/M300, MEDIA = (DX143, DX127), ACCESS = READ, SHARE = NORMAL....; . . . ENDSTEP; $ENDJOB;	STEP.....; ASSIGN COMM, HJ.OMN, DEVCLASS=MS/M300, MEDIA = (DX143, DX127), ACCESS = READ, SHARE = NORMAL....; ENDSTEP; $ENDJOB;

Figure 4-3. Interjob File Sharing

```
$JOB UNIV,...;
    .
    .
  STEP     MULTR,...;
  COMM     'THREE REQUESTS FOR READ ACCESS TO CENT.REF ALLOWING
           READ ACCESS (ONLY) TO OTHERS';
  ASSIGN   INC1, CENT.REF, ACCESS=READ;
  ASSIGN   INC2, CENT.REF, ACCESS=READ;
  ASSIGN   INC3, CENT.REF, ACCESS=READ;
    .
    .
  ENDSTEP;
    .
    .
$ENDJOB;

$JOB POLY.....;
  STEP     MANR,.....;
  COMMENT'CATALOG CONTAINS SHARE=NORMAL IN FILE DESCRIPTORS, ONLY ACCESS IS
           THEREFORE STARTED';
  ASSIGN   IST1, CENT.REF, ACCESS=READ;
  ASSIGN   IST2, CENT.REF, ACCESS=READ;
  ASSIGN   IST3, CENT.REF, ACCESS=READ;
    .
    .
    .
    .
  ENDSTEP;
    .
    .
    .
$ENDJOB;
```

Figure 4-4. File Sharing within a Job Step for Cataloged Files

When share and access requests are made on a file, the system will decide whether or not to grant the request, depending on the type of sharing currently active on the file. If the requested access is given, the system updates the current sharing mode accordingly (in preparation for further requests). If an inter-job file sharing request cannot be granted, the requesting job will be queued to wait until the request can be satisfied (when an appropriate job, or jobs, releases the file). If an access request from the step already using the file is unable to be granted, the job is aborted.

The ACCESS parameter can have one of four values, WRITE, READ, SPWRITE, SPREAD. Of the possible SHARE values, the ones recommended for standard use are NORMAL and ONEWRITE. Figure 4-5 shows the possible combinations of the SHARE and ACCESS parameter values and their corresponding meanings in terms of type of sharing requested. Note that SHARE can have the value FREE, but its use is not normally recommended since the system has no control over the access of files when FREE is specified. If SHARE and ACCESS are not specified, the step has exclusive use (read or write) of the file, via a single internal file name.

Keyword Values		Type of Sharing Requested
ACCESS=	SHARE=	
WRITE	NORMAL	Exclusive Use (default)
SPWRITE	NORMAL	Exclusive Use
READ	NORMAL	Read while any job reads
SPREAD	NORMAL	Read while same step reads
READ	ONEWRITE	Read while any job reads and one job writes
SPREAD	ONEWRITE	Read while same step reads and writes
WRITE	ONEWRITE	Write while any job reads
SPWRITE	ONEWRITE	Write while same step reads

Figure 4-5. File Sharing Requests

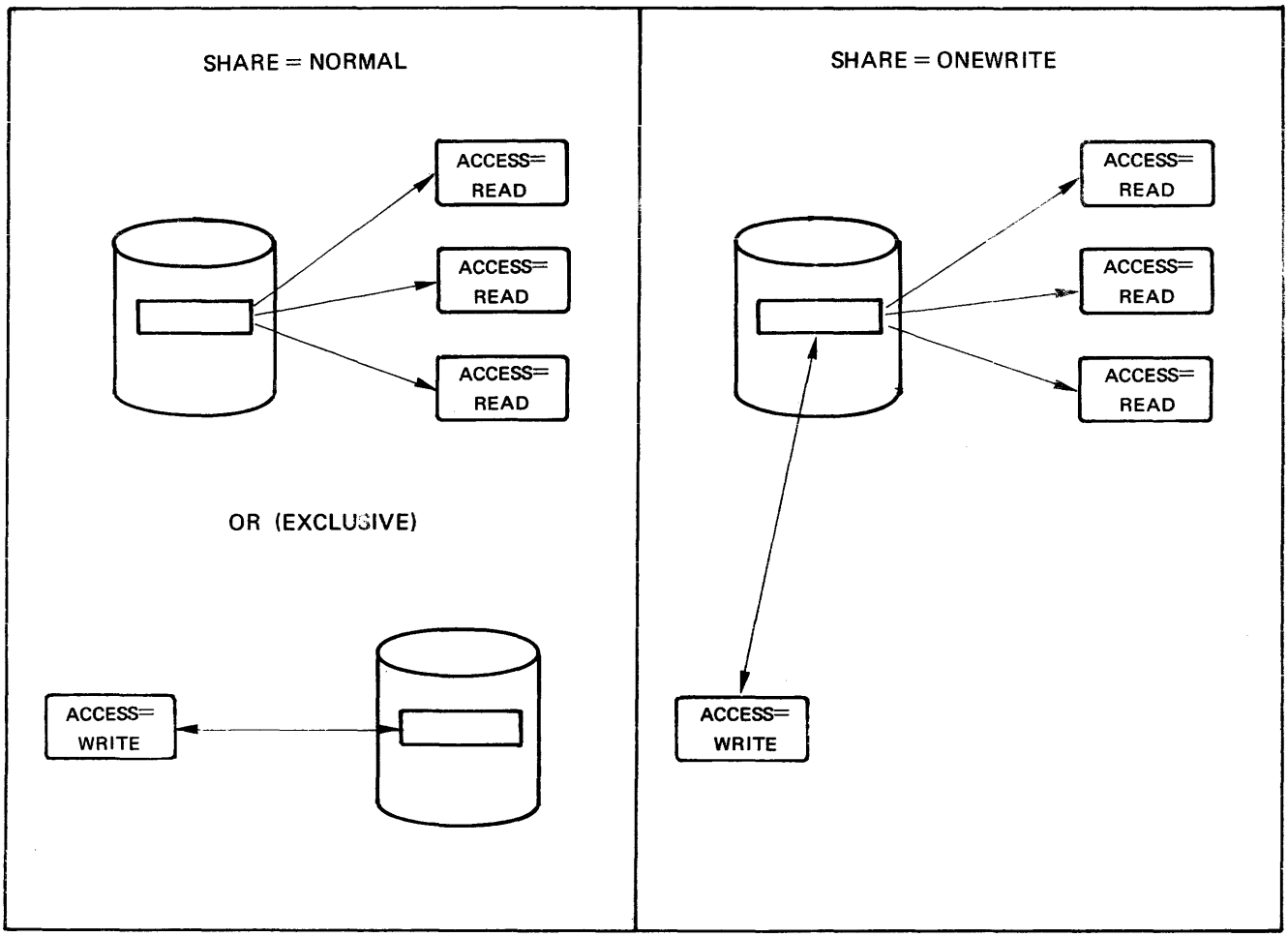


Figure 4-6. Shared Access to File

Figure 4-6 illustrates the rules that the system follows when it tests if access can be granted or not. It shows, for example, that a file assigned with values ACCESS=WRITE, SHARE=ONEWRITE may be shared with other jobs that specify ACCESS=READ, SHARE=ONEWRITE but not with any jobs that specify SHARE=NORMAL nor with another job that specifies ACCESS=WRITE, SHARE=ONEWRITE ; if, however, in this situation the original job (ACCESS=WRITE, SHARE=ONEWRITE) releases the file, any job with ACCESS=READ irrespective value of SHARE can access the file.

Note that the rules of ACCESS=SPREAD, ACCESS=SPWRITE correspond to those for ACCESS=READ and ACCESS=WRITE respectively expect that other accesses to the file are restricted to the same step (i.e. multiple assignments).

Figure 4-7. illustrates the effect of multiple assignments within a step.

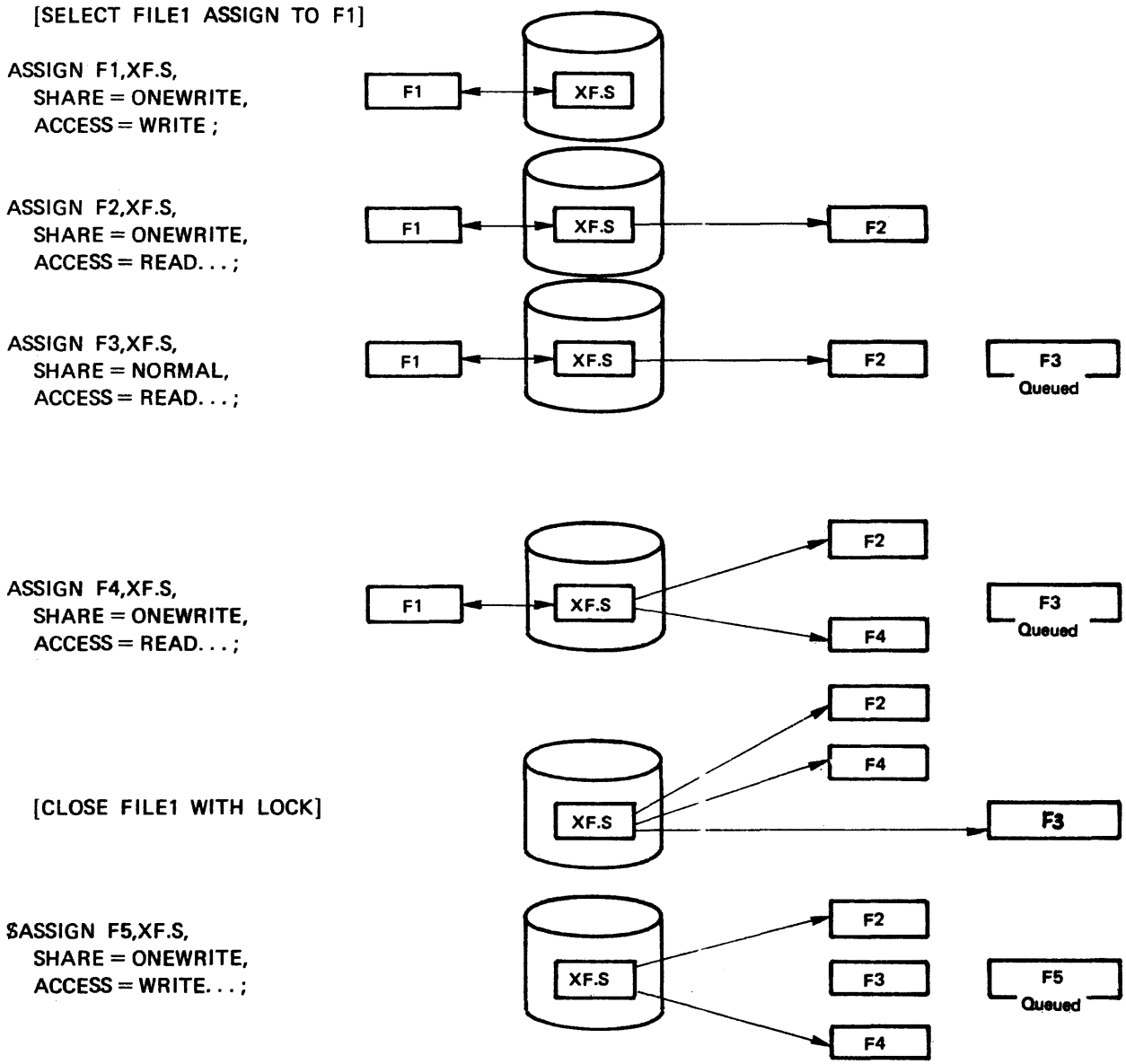


Figure 4-7. File Sharing Example

Notes :

- Sharing is possible on disk files only. A tape user always has exclusive read access or exclusive write access.
- If space is being allocated for a file within a step ($ALLOCATE), the file can only by accessed from within the step. In other words, the values WRITE and READ for the ACCESS parameter are treated in this case as SPWRITE and SPREAD respectively. The $ALLOCATE must refer to the first $ASSIGN statement in the step.
- When a file is passed (END = PASS), the sharing mode of the file remains until the next $ASSIGN for the file. This $ASSIGN must declare a new (or the same) sharing mode.

- The system does not check that the sharing mode requested is supported by the file organization. The user should make sure that the two are compatible.
- The catalog contains SHARE information. If SHARE is specified outside of the catalog and it is different to that stored in the catalog ; then the catalog sharing information is used but the step has exclusive use of the file (equivalent to SP). If the information is the same then no problem arises.
- The special value RECOVERY for the ACCESS parameter is reserved for recovery purposes of a cataloged file in the state ABORT = LOCKED ; the step has exclusive access to such a file (see Catalog Management Manual).

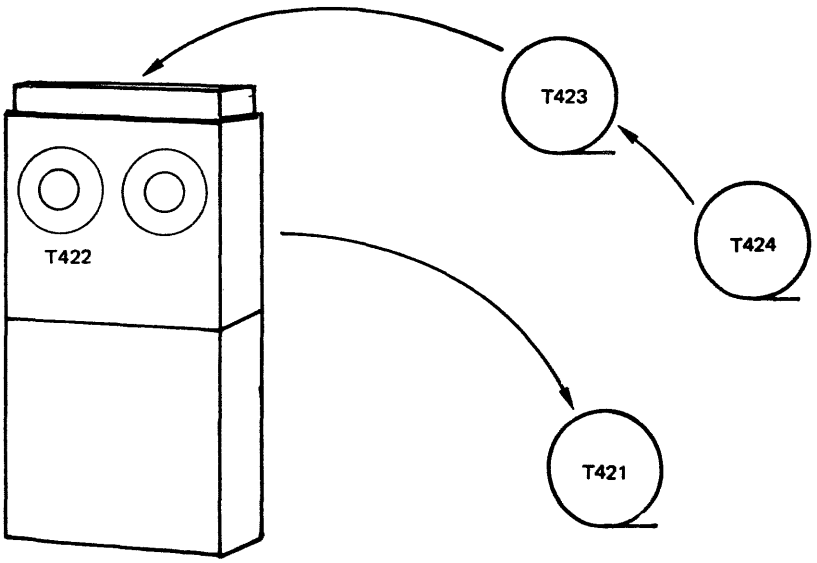
In theory, the number of devices needed to run a step will be equal to the number of different volumes which are specified in all the $ASSIGN statements present in the step description. In practice, there are two areas of application where device utilization can be minimized. These concern the mounting of multivolume files and the situation where different files using the same (type of) device are processed one after another rather than simultaneously.

If nothing is specified to the contrary, GCOS will reserve a device for each volume that is to be used in a job step. The only exception to this rule occurs for multivolume work tapes (see below).

The most obvious case for reducing this number of devices is in the case of a multivolume tape file. Since only one tape is actually required at any moment an assignment of the following type can be made :

```
ASSIGN IF1, EF1, DEVCLASS=MT, MEDIA=(M4T1, M4T2,M4T3,M4T4),
MOUNT=1,. . . ;
```

so that only one drive is requested instead of four, in this case, if MOUNT had not been specified. The MOUNT parameter in the above $ASSIGN statement indicates the number of volumes to be mounted simultaneously. The use of MOUNT is obviously essential when there are not sufficient devices available for all the volumes of a file. Figure 4-8 illustrates the function of the MOUNT parameter.




```
ASSIGN FI, MY.TF43, MEDIA=
(T421, T422, T423, T424), MOUNT = I, . . . ;
```

Figure 4-8. Use of MOUNT

The most useful values of MOUNT for multivolume tape files are MOUNT = 1 and MOUNT = 2.

If MOUNT = 1 is specified then only 1 tape drive will be used for the file. At the end of each volume used, the volume will be replaced by the next volume in sequence. Although minimizing device usage this technique does cause the program to be halted while the operator changes volumes.

With MOUNT = 2 only 2 devices are used for the file. However in this case the operator can mount in advance each volume and volume switching is not delayed by operator intervention, see Figure 4-9.

```
$COMM    'NO MOUNT PARAMETERS';
$ASSIGN  GLBE,REL.X,FILESTAT=UNCAT,DEVCLASS=MT/T9,
         MEDIA=(MA1,MA2,MA3,MA4) ;
```

4 drives used

```
$COMM    'MINIMUM DEVICE USAGE';
$ASSIGN  GLBE,REL.X,FILESTAT=UNCAT,DEVCLASS=MT/T9,
         MEDIA=(MA1,MA2,MA3,MA4),MOUNT=1 ;
```

only 1 drive used

```
$COMM    'MOUNTING IN ADVANCE BY OPERATOR';
$ASSIGN  GLBE,REL.X,FILESTAT=UNCAT,DEVCLASS=MT/T9,
         MEDIA=(MA1,MA2,MA3,MA4),MOUNT=2 ;
```

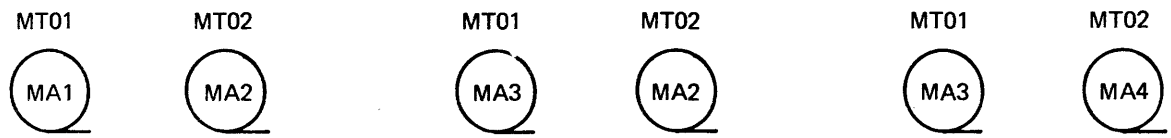


Figure 4-9. Multivolume Device Management

The use of MOUNT applies to both permanent and temporary tape files. If MOUNT is not supplied for a multivolume work tape file (MEDIA=WORK), GCOS operates as if MOUNT=1 is specified.

The MOUNT parameter continues to have effect when a normal tape file overflow onto a WORK volume (see the earlier paragraphs on Tape File Extension).

The MOUNT parameter can also be applied to the use of $ASSIGN for multivolume disk files or organization BFAS sequential or HFAS sequential and indexed sequential. MOUNT cannot be used on a file which has multiple assignments (i.e. more than one $ASSIGN in a job step, each having the same external file name but a different internal file name). Nor is MOUNT allowed when a multivolume disk file is being allocated (for example, in conjunction with a $ALLOCATE statement).

The MOUNT parameter may be used with a device pool (see below). If a $ASSIGN statement specifies POOL, FIRST then the number of devices indicated by MOUNT will be required at assign time. If POOL, NEXT is indicated, the specified number of devices will be required only at open time.

Use of Device Pools

Normally the access to a particular device is granted exclusively to a file for the duration of a job step. Suppose, however, an executing COBOL program contains the following statements :

```
SELECT FILE1 ASSIGN TO F1.

SELECT FILE2 ASSIGN TO F2.

OPEN INPUT FILE1.

CLOSE FILE1 WITH LOCK.

OPEN INPUT FILE2.
```

This means that the file FILE1 is completely processed before processing begins on file FILE2.

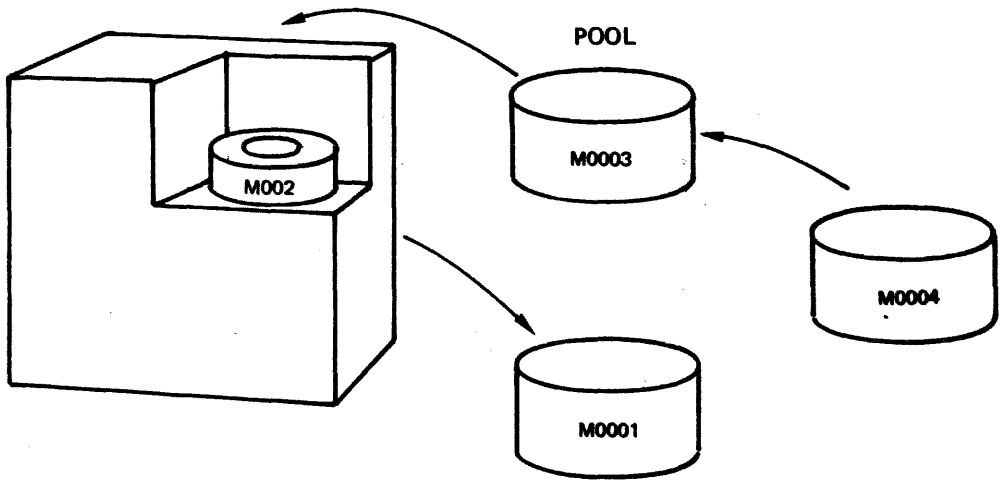
In the above case it would be possible to use the same device for F1 and F2. The user can inform GCOS that this is possible by using the $POOL statement in conjunction with the POOL parameter in $ASSIGN :

```
POOL I*MS/M402 ;

ASSIGN F1, MAX.Z, DEVCLASS=MS/M402, MEDIA=VOL1, POOL, FIRST, . . . ;

ASSIGN F2, BMY.I, DEVCLASS=MS/M402, MEDIA=VOL2, POOL, NEXT, . . . ;
```

Thus only one MSU0402 will be reserved for the use of the pooled files.
Figure 4-10. illustrates the concept of device pools.




```
POOL     I*MS/M400 ;
ASSIGN   F1,MS.4D,  MEDIA = M0001,  POOL, FIRST . . . ;
ASSIGN   F2,MS.5E,  MEDIA = M0002,  POOL, NEXT . . . ;
ASSIGN   F3,MS.6F,  MEDIA = M0003,  POOL, NEXT . . . ;
ASSIGN   F4,MS.7G,  MEDIA = M0004,  POOL, NEXT . . . ;
```

Figure 4-10. Device Pool Usage

As shown above, a device pool can be specified by the use of a $POOL statement together with, $ASSIGN statements (one for each file accessing the «pool»). The device pool technique depends on the logic of the processing program. When the program has completed the processing of a file it must signal to GCOS that the file can be deassigned,

thus freeing the devices assigned to the file. In COBOL this is done as indicated in the example by the inclusion of WITH LOCK in the CLOSE verb. A further example of device pool usage follows.

Suppose a COBOL program contains the following statements :

```
SELECT FC1 ASSIGN TO IFN1.
SELECT FC2 ASSIGN TO IFN2.
SELECT FC2 ASSIGN TO IFN3.
       .
       .
       .
OPEN   FC1.
       .
       .
       .
CLOSE  FC1 WITH LOCK.
OPEN   FC2.
       .
       .
       .
CLOSE  FC2 WITH LOCK.
       .
OPEN   FC3.
       .
CLOSE  FC3.
```

Since the files are used sequentially only one device is necessary. The job description could be :

```
$JOB....;
  STEP...;
  POOL    MT/T9/D1600;
  ASSIGN  IFN1, FIRSTFILE, DEVCLASS = MT/T9,
          MEDIA = TRUC,
          POOL, FIRST;
  ASSIGN  IFN2, SEC.FILE, DEVCLASS = MT/T9,
          MEDIA = CHOSE,
          POOL, NEXT;
  ASSIGN  IFN3, THIRD-FILE, DEVCLASS = MT/T9,
          MEDIA = MACHIN,
          POOL, NEXT;
  ENDSTEP;
$ENDJOB;
```

The mounting of volume TRUC will be requested before the load module execution is initiated. When the OPEN FC2 is executed, volume CHOSE will be mounted and when OPEN FC3 is executed, volume MACHIN. This job is able to run with only one device rather than three.

In the examples so far only one device has been pooled. In general a device pool may contain more than one device. If in the first example the disk files MAX.Z and/or BMY.I were each on two volumes, the pool statement would be :

```
POOL 2*MS/M400
```

The device pool is constructed as follows :

- The $POOL statement defines and reserves the number of devices of a particular device type to be placed in the pool.
- The POOL parameter of the $ASSIGN statement indicates that the device to be used for the current file must be selected in the pool for that device.
- The DEVCLASS or DVIDLIST parameter specifies what type of device is to be selected. The recommended practice is to use the $POOL statement with the device class parameter (DEVCLASS) so that the required number of devices is free but no explicit declaration is made concerning which physical devices are members of the pool.
- The FIRST parameter indicates that the named volume should be mounted at assign time. The sum of all the volumes of the files for which FIRST is specified (for multivolume files that includes the value of MOUNT where FIRST is specified) must not exceed the minimum number of devices specified in the $POOL statement.
- The NEXT parameter indicates that the volume mounting will be requested only at open time. However GCOS checks at the time of assignment that the file is not already being used, and the step will be kept waiting if the volume or the file is not accessible.

As shown already, a device pool can be specified for disk and tape or cassette device types. The files may be temporary or permanent. There is no restriction on the use of MOUNT in conjunction with device pools.

Example :

```
STEP. . . ;
POOL    2*MS/M400 ;
POOL    1*MT/T9 ;
ASSIGN  F1, AX.BM, DEVCLASS=MS/M400, MEDIA=4D2S, POOL, FIRST,. . . ;
ASSIGN  F2, AX.BP, DEVCLASS=MS/M400, MEDIA=4D20, POOL, NEXT,. . . ;
ASSIGN  F3, AX.BT, DEVCLASS=MS/M400, MEDIA=4D2E, POOL, FIRST,. . . ;
ASSIGN  F4, AX.BZ, DEVCLASS=MS/M400, MEDIA=4D2P, POOL, NEXT,. . . ;
ASSIGN  F5, AX.CD, DEVCLASS=MS/M400, MEDIA=4D2Z, . . . ;
ASSIGN  F6, P4.D1,  DEVCLASS=MT/T9, MEDIA=(T41, T42, T43), POOL,
        FIRST, MOUNT=1, . . . ;
ASSIGN  F7, P4.D2,  DEVCLASS=MT/T9, MEDIA=T63, POOL, NEXT, . . . ;
ASSIGN  F8, P4.D3,  DEVCLASS=MT/T9, MEDIA=T71, . . . ;
ENDSTEP ;
```

In the above example, if POOL and MOUNT had not been used we would require 5 MSU0400 disk drives and 5 tape drives. With POOL and MOUNT we require 3 disk drives (2 for files F1, F2, F3, F4 : 1 for F5) and 2 tape drives (1 for F6, F7 : 1 for F8). Note that the disk file AX.CD and the tape file P4.D3 do not access pooled devices.

Notes :

- The user should ensure that the number of volumes in a pool that are simultaneously required is always less than or equal to the number of corresponding devices in the pool.
- All $ASSIGN statements for a particular device pool may specify POOL, NEXT. This is useful in particular when the order in which the opening of the files will take place is not necessarily known in advance.

FILE PROTECTION

A measure of protection must be given to files in any processing system to ensure that spurious accesses are not possible which modify or otherwise compromise the integrity of a given file. Unrequested modifications or involuntary destruction of files due to errors made by the user or the operator, or due to the malfunctioning of hardware or software must be provided for.

GCOS provides this function in two main ways :

- By assignment, files are protected against non-requested access, since no file can be processed unless its name and supporting volume identification are supplied by the user.
- By a file logging method and checkpoint/restart mechanism, in which files are protected against unexpected events (refer to the section on Error Processing).

The protection afforded by the file assignment method consists of the following mechanisms :

- Expiry date, to protect a file against destruction prior to a given date.
- File Sharing, to allow only authorized sharing modes of a given file/s (see File Sharing).

A set of Utility Programs is provided to anticipate possible incidents .

- Saving and restoring files (FILSAVE, FILREST).
- Saving of files during RESTART (see Error Processing).

SETTING OF EXPIRATION DATES

Uncataloged Files

The EXPDATE parameter, for example, (in the $ASSIGN statement) provides security against the accidental destruction of a disk or tape file. The setting of expiration dates applies to permanent files only.

DISK FILE EXPIRATION DATES

The presence of an (unexpired) expiration date prevents a disk file from being deallocated by the normal use of the utility $DEALLOC ($HDEALLOC for HFAS files). Under normal circumstances it will also prevent the volume preparation of the volume containing the file.

The expiry date is stored in a disk file at file allocation time (as a parameter to r to $ASSIGN in conjunction with $ALLOCATE in a step in which the file is used, or as a parameter to $PREALLOC or $HALLOC).

Example :

```
$JOB . . . ;
   STEP . . . ;
   ASSIGN DK1, PRIV.ACCS, EXPDATE=30,
                  DEVCLASS=MS/M402, MEDIA=XXA2 ;
   ALLOCATE DK1, SIZE=50, UNIT=CYL ;
   ENDSTEP ;
$ENDJOB ;
```

The example given above allocates the file PRIV.ACCS and sets an expiration period of 30 days. Note that if, for instance, the above job was run on the 14th February 1977, making the expiration date the 16th March 1977, the parameter specification:

EXPDATE = 77/3/16 or EXPDATE = 77/75

would have stored the same expiration date setting in the file.

The file can be deallocated on or after the 16th March 1977.

Notes :

- The existence of an expiration date on a disk file does not prevent a program which refers to the same file name from modifying the contents of the file (i.e. in output or update processing mode); it is the file name itself (and the reservation of file space) that is protected.
- An expiration date setting on a disk file can be overridden if a $DEALLOC (or $HDEALLOC) statement contains the BYPASS parameter. This also applies to the $VOLPREP utility with BYPASS, but in this case the operator will have to give permission.
- A disk file can be renamed using the Data Management utility $FILRENAM even if an expiration date applies. However, after the file is renamed, the original expiry date setting will still be present.
- For uncataloged disk, files, the EXPDATE parameter of $ASSIGN only applies if file space is allocated in the current step (with $ALLOCATE).
- For cataloged files (disk and tape), the EXPDATE only has meaning when a new generation is created.
- For uncataloged tape files the retention period of a file can be specified each time the file is written, that is, in output mode. If no date is provided in this circumstance, then the current date is taken as the expiry date.

5 • Maintenance of stored JCL

When a set of JCL statements is to be used more than once, it is convenient to store it in a permanent disk or tape file from which it may be accessed as required. The set of statements may be part of a job description (referred to as a JCL sequence) or it may consist of one or more complete job descriptions (referred to as a job stream).

By means of the $RUN statement, a user can submit to the Stream Reader a stored job stream for translation and execution ; alternatively, the user can insert a stored JCL sequence into a current job at translation time ($INVOKE) or at execution time ($EXECUTE). Details of these statements are given in the JCL Reference Manual.

MAINTENANCE OF STORED JCL

Job streams and JCL sequences can be stored in members of permanent library files ; the user can create and maintain each library member by means of the $LIBMAINT utility, which has its own comprehensive text editing facility. Refer to the Library Maintenance User Guide for full details on the maintenance of library files.

In addition, the system provides a «mini» editing facility (see Appendix A) and a parameter substitution facility (see the JCL Reference Manual), so that the user can modify stored JCL sequences and job streams dynamically to suit the requirements of a particular job run.

Figure 5-1 illustrates a method, using $LIBMAINT, of storing in a library member an input enclosure that contains a JCL sequence ; Figure 5-2 gives an example of the corresponding job description. Figure 5-3 shows the comparison between job submission by cards and job initiation via the $RUN statement. Note that since only one job is involved here, a $INVOKE or a $EXECUTE statement could be used (provided the $JOB and $ENDJOB statements are removed from the stored file and there are no input enclosures).

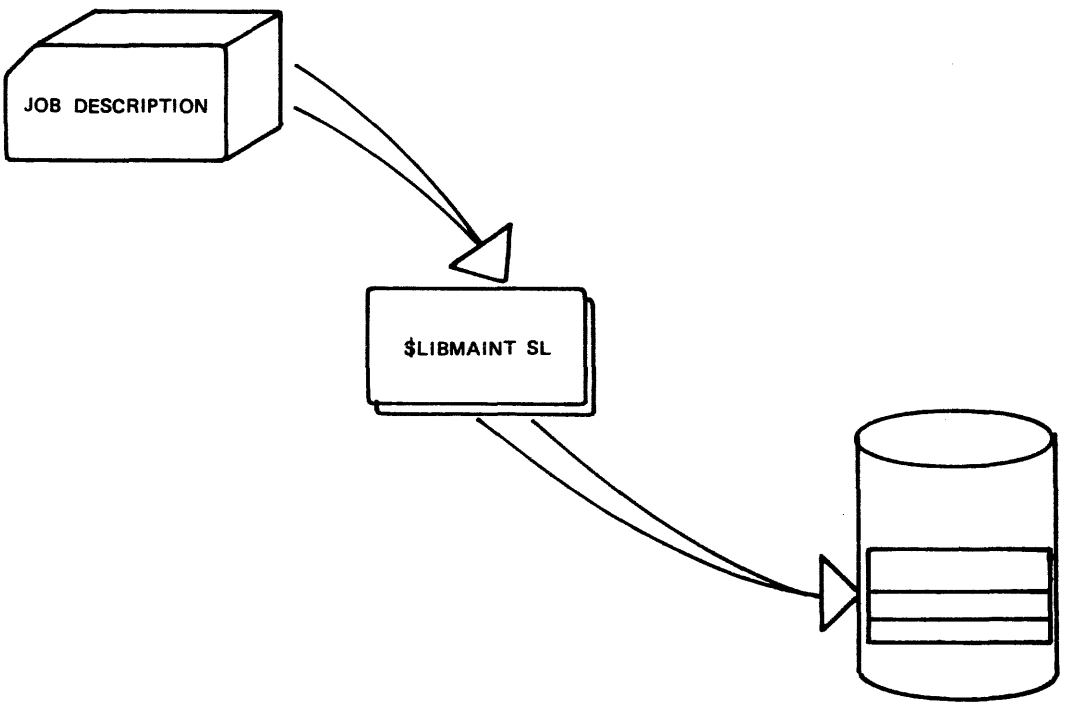


Figure 5-1. Stored JCL

```
$JOB . . . ;
     COMMENT'IT IS ASSUMED THAT THE LIBRARY GEN.JCL
              HAS ALREADY BEEN ALLOCATED ';
     COMMENT'STORE JCL STATEMENTS ON SUBFILE NEWJCL
              OF LIBRARY FILE GEN.JCL';
          LIBMAINT SL,
              LIB= GEN.JCL
              COMFILE=*INCARDS ;
$INPUT INCARDS ;
          MOVE COMFILE : NEWJCL , TYPE=JCL, NUMBER ;
                    .
                    .
        < JCL statements >
                    .
                    .
$ENDINPUT ;
$ENDJOB ;
```

Figure 5-2. Example of Storing JCL in a Library Member

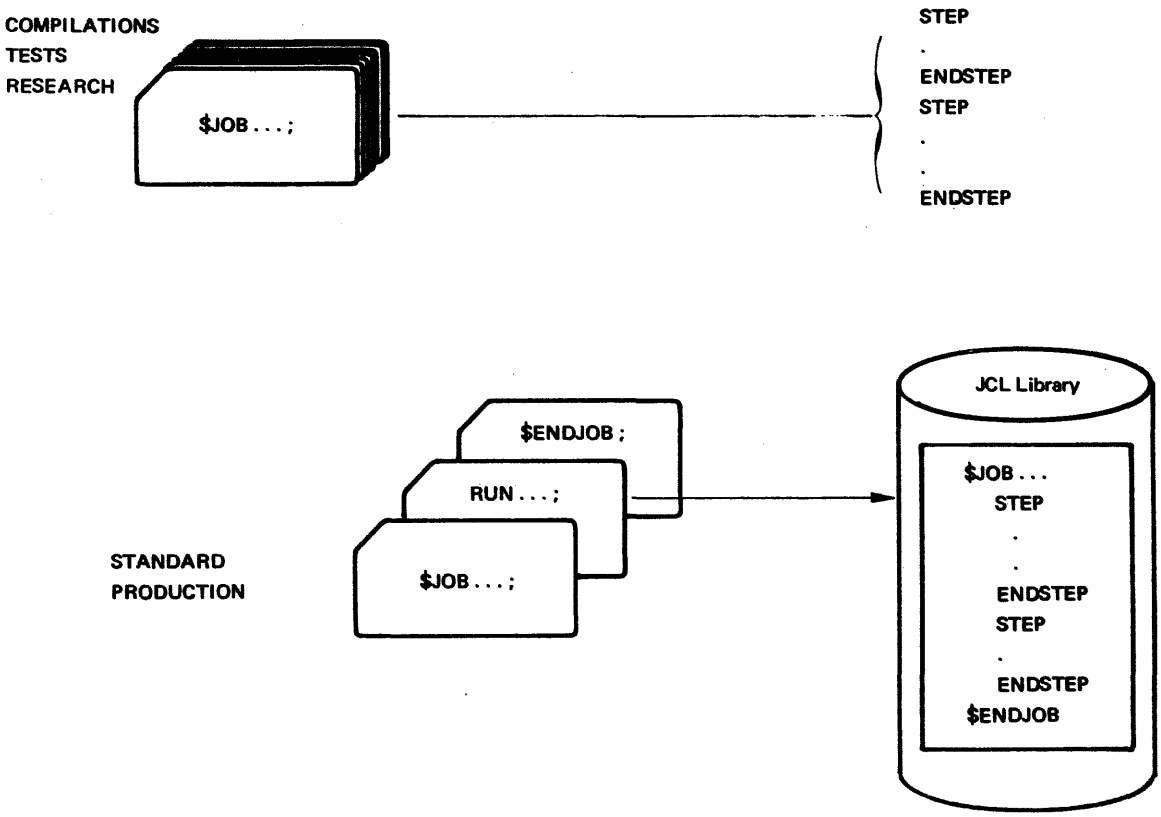


Figure 5-3. JCL Submission

$RUN, $INVOKE AND $EXECUTE

An important difference between the use of $RUN and $INVOKE (or $EXECUTE) concerns the processing of the job description. $RUN requests the scheduling for execution of a job stream (the «spawned» jobs) that is independent of the job issuing the $RUN statement (the «spawning» job). The spawning operation is activated when the $RUN statement is encountered during the execution of the original job. Suppose, for example, a spawned job contains statements which request the updating of a data file. Since the execution of the spawned job depends on its scheduling parameters and on the current job environment, the spawning job can make no assumption concerning the updating of the file. Therefore it would normally be bad practice if, subsequent to the $RUN statement, the spawning job contained a statement which accessed the same data file.

The use of $INVOKE causes a sequence of JCL statements to be inserted directly into the current job description in place of the $INVOKE statement itself. This operation is performed at JCL statement translation time (any $INVOKE statements encountered within this sequence are also replaced at JCL translation time). These statements are then executed, in order, as part of the current job.

The statement following the original $INVOKE will not be executed until all the replacing statements have been executed, subject to any $JUMP statements present (see the JCL Reference Manual).

Unlike the $INVOKE statement, $EXECUTE is activated at execution time ; no replacement is made at JCL statement translation time. $INVOKE is static in the sense that a sequence of JCL statements is inserted into a job description at translation time and thus becomes part of the job description ; $EXECUTE is dynamic since the sequence to be executed is not identified and translated until execution time and, once the sequence has been executed, it has no further significance to the job description that contains the $EXECUTE statement.

The operator SJ and SI commands perform the same function as $RUN but are entered from the operator's console. They provide the same scheduling parameters as $RUN. The operator can always exercise subsequent control over any job using other operator commands (in particular, the MJ command).

Notes :

- Although there can be a marked difference in the effect of $INVOKE and $EXECUTE, most of the rules for the use of the statements are identical. At the end of this Section there is a comparison between the two statements («Differences between $INVOKE and $EXECUTE») ; elsewhere in the Section, where there is a general explanation or example that applies to both statements, only the $INVOKE statement is used, in order to simplify the discussion.
- In this Section, the terms '$INVOKEd' and '$EXECUTEd' will be used to identify stored JCL sequences that are accessed by $INVOKE and $EXECUTE respectively.

Throughout this discussion the information given about the $RUN statement is also applicable to the operator SJ and SI commands.

Parameters of the $JOB statements of a spawned job or jobs are computed from :

- parameters of the stored $JOB statement (if any)
- parameters of the $RUN statement
- parameters of the $JOB statement of the spawning job.

The parameters of the $JOB statement for each stored job override those of the $RUN statement and those of the $JOB statement. If, however, one or more parameters are missing from the stored $JOB statement, the following rules apply :

job-name	=	job-name in stored $JOB statement.
user-name	=	user-name in stored $JOB statement.
project-name	=	project-name in stored $JOB statement ; if none, spawning job's project.
billing-name	=	billing-name in stored job ; if none, spawning job's billing.
job-class	=	parameter of $RUN statement ; if none, parameter of stored $JOB statement ; if none, class P.
scheduling-priority	=	parameter of $RUN statement ; if none, parameter of stored $JOB statement ; if none, default value associated with spawned job class.

HOLD = parameter of $RUN statement or parameter of stored $JOB statement ;
HOLD is considered as present if it is present in either statement (or both).

HOLDOUT = parameter of $RUN statement or parameter of stored $JOB statement. HOLDOUT is considered present if it is present in one statement (or both).

REPEAT = parameter of stored $JOB statement.

Note :

If the job stream referred to by $RUN contains more than one job, each job must be delimited by $JOB and $ENDJOB statements. However, if the job stream contains only one job it is possible to omit the $JOB and $ENDJOB statements. In such a case the job-name is assumed to be the same as the member-name specified in the $RUN statement which in this case must not contain more than 8 characters, the user-name is assumed to be the same as that of the $RUN statement, and the REPEAT parameter cannot be specified.

A summary of these relationships is shown in Table 5-1.

The above rules can be expressed also by the fact that identification is given by the stored job description in preference to the spawning job, while, on the other hand processing information comes from the spawning job in preference to the stored job description.

PARAMETER	$STEP	$RUN SJ/SI	STORED JOB	SPAWNING JOB	DEFAULT
Job-id	–	2	1	–	–
User-id	–	–	1	2	–
Project	–	–	1	2	catalog
Billing	–	–	1	2	catalog
Job-class	–	1	2	–	P
Sch-pr	–	1	2	–	class default
HOLD	–	1	1	–	not held
HOLDOUT	–	1	1	–	not held
REPEAT	–	–	1	–	no repeat

Table 5-1. Priority Order for Parameters of Spawned Job

The spawning job can pass information to spawned jobs through the use of parameter substitution (see JCL Reference Manual, Appendix E) and through the use of switches (see the JCL Reference Manual, $RUN statement).

The following examples illustrate the handling of $JOB parameters in spawned jobs.

Example 1 :

Assume that a job stream containing a single job has been stored in member TUES32 of library JOBS. LIB. The spawning job could be :

```
$JOB LONDON, USER = X123, PROJECT = INVHQ, CLASS = L ;
   RUN TUES32, JOBS.LIB ;
$ENDJOB ;
```

The spawned $JOB statement would be :

```
$JOB T32, USER = X123, PROJECT = INVHQ, CLASS = P,
stored $JOB  spawning $JOB  spawning $JOB          default
```

Now, if instead the $RUN statement was :

```
RUN  TUES32,  JOBS.LIB,  CLASS = K,  HOLD,  PRIORITY = 2 ;
```

The spawned job would have a $JOB statement of the form :

```
$JOB T32,  USER = X123,  PROJECT = INVHQ,  CLASS = K,
stored $JOB  spawning $JOB  spawning  $JOB            $RUN

HOLD,  PRIORITY = 2 ;
$RUN             $RUN
```

Example 2 :

Assume that a job stream containing two jobs has been stored in member WED32 of library JOBS.LIB. The spawning job could be :

```
$JOB  LONDON,  USER = X123,  PROJECT = INVHQ,  CLASS = L ;
   RUN  WED32,  JOBS.LIB ;
$ENDJOB ;
```

Assume also that the $JOB statements of the stored jobs were :

```
$JOB  MONDAY,  USER = AQ47,  PROJECT = BRW,  CLASS = M;
$JOB  TUESDAY,  USER = AQ48 ;
```

The spawned $JOB statements would be :

```
$JOB  MONDAY,  USER = AQ47,  PROJECT = BRW,  CLASS = M ;
stored  $JOB      stored  $JOB      stored  $JOB      stored  $JOB

$JOB  TUESDAY,  USER = AQ48,  PROJECT = INVHQ,  CLASS = P ;
stored  $JOB      stored  $JOB      spawning  $JOB          default
```

Note :

The use of $RUN for the control of interdependent jobs (i.e. jobs whose processing is consequent to the execution of other jobs) is shown in Section 6.

USE OF $INVOKE AND $EXECUTE

$INVOKE and $EXECUTE statements refer to stored JCL sequences. The $INVOKE and $EXECUTE statements are replaced by the referenced JCL sequences as described above. Note that the stored JCL sequence must not contain JCL statements which are handled by the Stream Reader ($JOB, $ENDJOB, $INPUT or $ENDINPUT).

Suppose that member PREA of library JOBS.ILIB contains the following statements :

```
PREALLOC MYFILE, INV,
    DEVCLASS = MS/M400, FILESTAT = CAT,
    GLOBAL = (MEDIA = C112, SIZE = 5),
    UFAS = (SEQ=(CISIZE=1024, RECSIZE=100)) ;
```

Then the job :

```
$JOB NEW, USER = PETER, PROJECT = MARY ;
    INVOKE PREA, JOBS.ILIB ;
    STEP LM1. . . ;
    ENDSTEP ;
$ENDJOB ;
```

will be equivalent to the job :

```
$JOB NEW, USER = PETER, PROJECT = MARY ;
  PREALLOC MYFILE.INV,
    DEVCLASS = MS/M400, FILESTAT = CAT,
    GLOBAL = (MEDIA = C112, SIZE = 5),
    UFAS = (SEQ=(CISIZE=1024, RECSIZE=100)) ;
  STEP LM1. . . ;
  ENDSTEP ;
$ENDJOB;
```

INPUT ENCLOSURES IN STORED JCL

As $INPUT and $ENDINPUT cannot be used in a stored JCL sequence with $INVOKE and $EXECUTE, an input enclosure cannot be contained in such a JCL sequence. However, a job containing $INVOKE or $EXECUTE can contain an input enclosure, as shown in the following example. Consider a job of the form :

```
$JOB. . . ;
  SORT INFILE = (. . .), OUTFILE = (. . .),
    COMFILE = *ORDER ;
$INPUT ORDER ;
  .
  .
  sort commands >
  .
$ENDINPUT ;
$ENDJOB ;
```

A JCL sequence of the form outlined below can be stored under the name PETER in library MY.JCLLIB :

```
SORT  INFILE = (. . .), OUTFILE = (. . .) ;
      COMFILE = *ORDER ;
```

A job of the following form can invoke the above sequence :

```
$JOB. . . ;
   INVOKE PETER, MY.JCLLIB ;
$INPUT ORDER ;
      .
      .
   sort commands
      .
      .
$ENDINPUT ;
$ENDJOB ;
```

obtaining the same result as the original job.

INDEPENDENCE OF $INVOKEd OR $EXECUTEd JCL SEQUENCES

A label which is defined inside a $INVOKEd or $EXECUTEd JCL sequence cannot be referenced outside the sequence. In addition, it is not possible to «jump» outside the sequence. Therefore the user can define the same label name both inside and outside the sequence with no subsequent ambiguity.

For example, using the following stored JCL sequence named TRUC :

```
   STEP LM1. . . ;
      .
   ENDSTEP ;
JUMP A, STATUS, NE, O ;
   STEP LM2. . . ;
      .
   ENDSTEP ;
A :
```

the following job description :

```
$JOB. . .;
   A :  STEP S1, . . . :
           .
        ENDSTEP ;
   JUMP A, STATUS, EQ, 12 ;
   INVOKE TRUC ;
$ENDJOB ;
```

will effectively become :

```
$JOB...
   A : STEP S1,...;
       ENDSTEP ;
JUMP A, STATUS, EQ, 12 ;
   STEP LM1...;
   ENDSTEP ;
JUMP A', STATUS, NE, 0 ;
   STEP LM2...;
   ENDSTEP ;
   A' :
$ENDJOB ;
```

Note :

A' is not a legal label, but is used in the above example to show the distinction between label A in the job containing the $INVOKE and label A in the $INVOKEd enclosure.

Independence of $INVOKEd or $EXECUTEd sequences also applies to parameter value substitution by means of the $VALUES statement (see the JCL Reference Manual), and to the scope of $LIB statements (see the Library Maintenance User Guide). If any of these statements appears in a $INVOKEd or $EXECUTEd sequence, it applies only to that JCL sequence. However, a $LIB statement in the job (or JCL sequence) that contains the $INVOKE will apply within a sequence that does not itself contain a $LIB statement. In other words, if there is a $LIBMAINT, but no preceding $LIB, in the sequence, the search path declared in the job or sequence that contained the $INVOKE will apply to the $LIBMAINT statement.

NESTED $INVOKE AND $EXECUTE STATEMENTS

A stored sequence may itself contain $INVOKE and/or $EXECUTE statements. These statements, in turn, can refer to stored sequences that contain $INVOKE and/or $EXECUTE statements, and so on. However, whereas there is no control over the number of levels of «nesting» for $EXECUTE, $INVOKE statements can only be nested up to a depth of nine levels. Any $INVOKE statement in a stored sequence that is referred to by a $EXECUTE statement will not be translated and replaced until the $EXECUTE itself is executed. The above rules concerning the independence of stored sequences apply to each level of nesting.

INVOKING OR EXECUTING INPUT ENCLOSURES

Sequences of JCL statements can be $INVOKEd or $EXECUTEd from input enclosures, rather than from stored files. The user can take advantage of this facility for testing and debugging purposes before storing a JCL sequence in a library.

For example :

```
$JOB. . . ;
   INVOKE *TEST ;
$INPUT TEST ;
   STEP LM1. . . ;
      :
   ENDSTEP ;
$ENDINPUT ;
$ENDJOB ;
```

will become after translation

```
$JOB. . . ;
   STEP LM1. . . ;
      :
   ENDSTEP ;
$ENDJOB ;
```

THE UPDATE PARAMETER OF $INVOKE

The UPDATE parameter of the $INVOKE statement specifies that the «Mini-Editor» commands stored in the named input enclosure must be applied to the invoked JCL sequence before translation. The use of the Mini-Editor is described in Appendix A.

DIFFERENCES BETWEEN $INVOKE AND $EXECUTE

The JCL Translator replaces $INVOKE statements at job translation time (unless such statements are contained in a stored sequence subject to a $EXECUTE statement). This implies the following :

- $INVOKE cannot refer to a JCL sequence which is created in a previous step in the same job ; nor can it take account dynamically of any file updates which may be made during job execution.
- The JCL translator does not act as a user step, in the following sense : if the $INVOKE statement references a library on a volume which is not RESIDENT and not known to the system at JCL translation time, then it will not ask the operator to mount the volume, nor will it wait for the volume to be mounted ; the job is aborted at JCL translation time ; this is done so that the translation of other jobs will not be delayed. Therefore, a user who references a non RESIDENT library via an $INVOKE statement, must ensure that the volume is mounted before the job is input to the system.

The above restrictions can be avoided by using the $EXECUTE statement as described below.

USE OF INVOKE WITH NON-RESIDENT LIBRARIES

As mentioned above a $EXECUTEd JCL sequence can contain $INVOKE statements. Such statements, because they are translated at job execution time, are not subject to the above restrictions. To force the mounting of the required volume(s) in advance of the $INVOKE statement, the following steps are required :

1. Put the appropriate $INVOKE statement in an input enclosure, or store it in a library member.

2. Specify in a $LIB statement the library that contains the stored JCL sequence that is to be $INVOKEd.

3. Below the $LIB statement, add a $EXECUTE statement that refers to the input enclosure or library member containing the $INVOKE.

For example :

```
LIB  SL,   INLIB1 = (TOOLS.SLLIB, DEVCLASS=MS/M400, MEDIA=TEAM-15)
           INLIB2 = (TEST.SLLIB, DEVCLASS=MS/M400, MEDIA=EXP42) ;
             :
EXECUTE *INVOKE-ENC ;
```

At step-initiation the operator is requested to mount the volumes TEAM-15 and EXP42. Then, if the input enclosure INVOKE-ENC is as follows :

```
$INPUT INVOKE-ENC ;

INVOKE MY-WORK, (TOOLS.SLLIB. DEVCLASS=MS/M400, MEDIA=TEAM-15) ;

INVOKE TEST-WORK, (TEST.SLLIB, DEVCLASS=MS/M400, MEDIA=EXP42) ;

$ENDINPUT ;
```

these statements will be translated successfully, because the volumes TEAM-15 and EXP42 are already mounted.

A useful feature of the $EXECUTE statement is that if a particular $EXECUTE statement is executed several times in the same job (for example, by means of a $JUMP statement), it is possible for a different version of the sequence to be created each time. For example, the file that contains the sequence may be modified using $LIBMAINT between each execution of the $EXECUTE statement. See Section VI for a discussion of this technique.

To summarize, the choice between using $INVOKE and $EXECUTE depends upon :

a) whether the stored JCL statements are on a resident disk, or a non-resident disk which is not mounted ;

b) whether the insertion of JCL can be static or must be done dynamically ;

c) whether the user wishes to have all JCL errors detected before any step is started, or at job execution-time.

PARAMETER SUBSTITUTION IN STORED JCL

The VALUES parameter in the $RUN statement allows a spawning job to supply parameter values for the execution of a stored job stream. Suppose the stored job stream TEXT in library JOBS.L1 is of the form :

```
$JOB. . . ;

FILSAVE INFILE = (&1),

              OUTFILE = (&2,DEVCLASS = MT/T9, MEDIA = &3) ;

$ENDJOB ;
```

Then the execution of :

```
RUN TEXT, JOBS.L1, VALUES = (MY.GOOD.FILE,MY.SFILE, T11 7) ;
```

will create a job of the form :

```
$JOB. . . ;
FILSAVE INFILE = (MY.GOOD.FILE),
        OUTFILE =(MY.SFILE,DEVCLASS = MT/T9,MEDIA = T11 7) ;
$ENDJOB ;
```

causing the disk file MY.GOOD.FILE (which is on a resident disk or cataloged) to be saved on tape T11 7 in file MY.SFILE.

The values indicated in the $RUN statement have no effect on the spawning job. Nevertheless, these values are ordinary parameters in the spawning job description and may themselves be variable parameters to be replaced at spawning job translation time by values supplied inside the spawning job.

For example, the spawning job could be :

```
$JOB... ;
VALUES ABC, DEF, MY.BAD.FILE ;
RUN TEXT, JOBS.L1, VALUES = (&3, MY.SFILE, T11 7) ;
$ENDJOB ;
```

This $RUN statement is equivalent to

```
RUN TEXT, JOBS.L1, VALUES = (MY.BAD.FILE, MY.SFILE, T11 7) ;
```

Using the same stored job stream as before, this will create the job :

```
$JOB. . . ;
FILSAVE INFILE = (MY.BAD.FILE) ,
        OUTFILE = (MY.SFILE, DEVCLASS = MT/T9, MEDIA = T11 7) ;
$ENDJOB ;
```

but the $VALUES statement in the spawning job will be unaffected by the $RUN statement.

It is also possible to include $VALUES statements in the stored job stream. These values will be used as default values for variable parameters when the $RUN statement does not provide any. If the stored job stream TEXT in library JOBS.L2 is :

```
VALUES F.J, F.SFILE, T117, MT/T9 ;
FILESAVE INFILE = (&1),
        OUTFILE = (&2, DEVCLASS = &4, MEDIA = &3) ;
```

The execution of :

```
RUN TEXT, JOBS.L2, VALUES = (MY.FILE) ;
```

will create a job (using default values of 2nd, 3rd and 4th variable parameters) :

```
$JOB. . . ;
   FILSAVE INFILE = (MY.FILE) ,
            OUTFILE = (F.SFILE, DEVCLASS = MT/T9, MEDIA = T11 7) ;
$ENDJOB ;
```

while :

```
   RUN TEXT, JOBS.L2, VALUES = (YOUR.FILE,,W142) ;
```

will create a job (using default values of 2nd and 4th variable parameters) :

```
$JOB. . . ;
   FILESAVE INFILE = (YOUR.FILE) ,
             OUTFILE = (F.SFILE, DEVCLASS = MT/T9, MEDIA = W142) ;
$ENDJOB ;
```

Moreover, it is possible to force from the $RUN statement a default value that is defined in the stored job stream to be considered as absent. This is done by giving the value NIL in the $RUN statement. Taking the same example as above :

```
RUN TEXT, JOBSL2, VALUES = (A.FILE,, NIL, NIL) ;
```

will create a job equivalent to :

```
$JOB. . . ;
   FILSAVE INFILE = (A.FILE),
            OUTFILE = (F.SFILE) ;
$ENDJOB ;
```

in which default values for the 3rd and 4th parameters are ignored.

Note :

The SJ or SI operator commands provide the same set of functions as the $RUN statement.

Values for variable parameters can also be provided in the $INVOKE and $EXECUTE statement. The stored JCL sequences can include $VALUES statements which are used as default values for variable parameters when $INVOKE or $EXECUTE do not provide any. NIL values can also be used in $INVOKE and $EXECUTE statements.

These mechanisms follow exactly the same rules as for the $RUN statement.

If the stored JCL sequence was :

```
VALUES D.F, C112, 10, 1024, 100 ;
PREALLOC &1, DEVCLASS = MS/M400,
      GLOBAL = (MEDIA = &2, SIZE = &3),
      UFAS = (SEQ=(CISIZE=&4, RECSIZE=&5)),
      FILESTAT=CAT ;
```

a statement :

```
INVOKE PREA, INV.JOB, VALUES = (OTHER.FILE) ;
```

will be replaced (using default value of 2nd, 3rd and 4th variable parameters) by

```
PREALLOC OTHER.FILE, DEVCLASS = MS/M400,
    GLOBAL = (MEDIA = C112, SIZE = 10),
    UFAS = (SEQ=(CISIZE=1024, RECSIZE=100),
    FILESTAT = CAT ;
```

while a statement :

```
INVOKE PREA, INV.JOB, VALUES (HIS.FILE,, 2,,200) ;
```

will be replaced (using default values of 2nd and 4th variable parameters) by :

```
PREALLOC HIS.FILE, DEVCLASS = MS/M400,
    GLOBAL = (MEDIA = C112, SIZE = 2),
    UFAS = (SEQ=(CISIZE=1024, RECSIZE=200)),
    FILESTAT = CAT ;
```

Figure 5-4 demonstrates repeated substitution of parameter values by means of $RUN and $INVOKE statements.

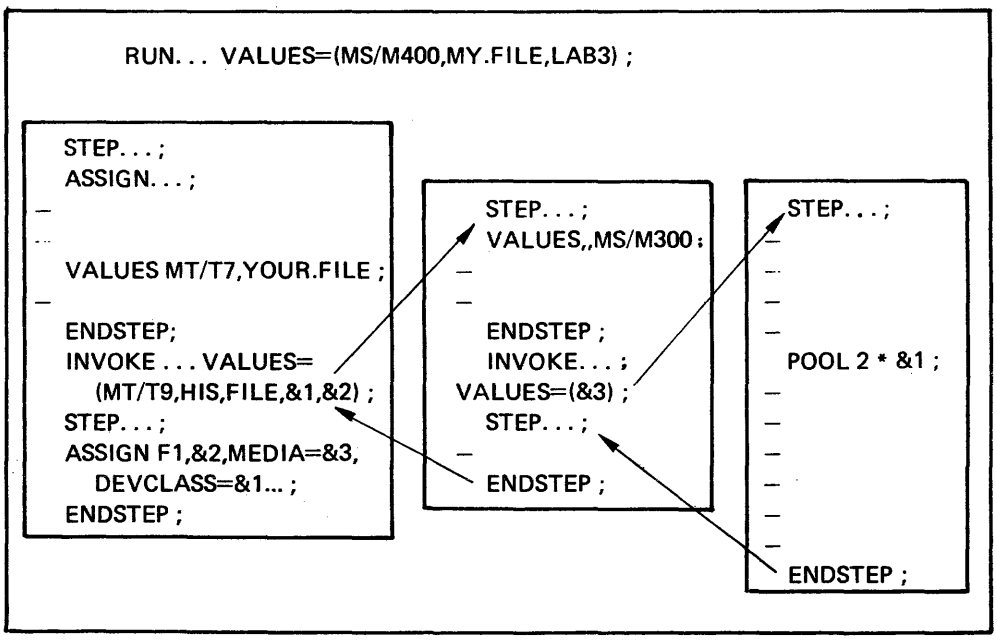


Figure 5-4. Variable Parameters in Stored JCL

JOB STREAM CREATION AND MAINTENANCE

A job stream is a sequence of job descriptions that the Stream Reader can read. The Stream Reader is able to read a job stream from

- a card deck

– a sequential disk or tape file

– a source library member

Furthermore, in the last two cases the operator can ask the Stream Reader to select only certain jobs of the job stream.

A job stream can be stored in a sequential disk or tape file using the $CREATE statement as in the following example :

```
$JOB. . . ;
   CREATE INFILE = (STREAM, MEDIA=KARDS, DEVCLASS=CD/R/C80),
                OUTFILE = (STREAM 1, MEDIA=DD345, DEVCLASS=MT/T7) ;
$ENDJOB ;
```

The same function is also provided via $LIBMAINT :

```
$JOB. . . ;
   LIBMAINT SL,
                INFILE = (STREAM, MEDIA=KARDS, DEVCLASS=CD/R/C80),
                OUTFILE = (STREAM1, MEDIA=DD345, DEVCLASS=MT/T7),
                COMMAND = 'MOVE INFILE:DECK, INFORM=SARF, TYPE=JCL,
                          OUTFORM-SARF ' ;
$ENDJOB ;
```

Execution of any of the two preceding jobs will request the operator to mount a deck of cards which is the job stream to be stored. The system will issue the message .

```
hh.ss CD0i  MOUNT  KARDS  FOR  Xj.
```

where CD0i is the device on which the deck of cards is to be mounted and KARDS is taken from the value given to the MEDIA parameter.

The job stream will be stored in file STREAM1 on tape DD345. Later, the operator will be able to issue the SI command. For example :

```
SI STREAM1 : D345 : MT/T7
```

On this command the Stream Reader will process the job stream.

The advantage of using the card reader directly, as in the above examples, is that $JOB statements can be used in the input job stream. This is not possible when the job stream is input via the Stream Reader (e.g. in an input enclosure). This is because when the Stream Reader encounters a $JOB statement it considers that the current job description is complete and that a new one is starting. The Stream Reader does this in order to project jobs from each other (otherwise a job description which did not end with $ENDJOB would be merged with the next job description).

The disadvantage of using the card reader directly is that the card reader must be assigned exclusively to the job and cannot be used during this time for normal job stream input. In certain installations the card reader might be in constant use and exclusive assignment will not be permitted.

In order to avoid direct use of the card reader the following can be done :

- Remove the $ sign from each $JOB statement in the job stream to be input, and
- Put the job stream in an input enclosure and move the input enclosure to a library member using the MOVE command of $LIBMAINT.

Note :

The above technique will result in a library member containing $JOB statements without the $ sign. This situation is the only exception to the rule that the $JOB statement must contain the $ sign.

For example, assume that the following job stream is to be stored in member STREAM2 of the source library SOURCES on a resident disk :

```
$JOB        A. . . ;
  .
  .
$ENDJOB ;
$JOB        B. . . ;
  .
  .
$ENDJOB ;
  .
  .
$JOB        LAST. . . ;
  .
  .
$ENDJOB ;
```

The $ signs should be removed and the job stream should be preceded by a MOVE command and followed by an //EOD command :

```
MOVE COMFILE : STREAM2, REPLACE, TYPE = JCL ;
JOB A . . . ;
:
$ENDJOB ;
JOB B . . . ;
:
$ENDJOB ;
JOB LAST . . . ;
:
$ENDJOB ;
//EOD
```

Because the jobs of the job stream may have their own input-inclosures, it is necessary to use the END=MATCH facility of the $INPUT statement ; otherwise, the Stream Reader would consider that the deck of cards ends at the first $ENDINPUT it encounters. The input-enclosure becomes :

```
$INPUT JOBSTREAM, END=MATCH ;
MOVE COMFILE : STREAM2, REPLACE, TYPE=JCL ;
JOB A . . . ;
.
.
$ENDJOB ;
```

```
JOB B . . . ;
   .
   .
$ENDJOB ;
JOB LAST . . . ;
   .
   .
$ENDJOB ;
//EOD
$ENDINPUT JOBSTREAM ;
```

and the corresponding $LIBMAINT step is :

```
LIBMAINT SL,
        LIB = SOURCES,
        COMFILE = JOBSTREAM ;
```

So the complete job becomes :

```
$JOB RECORD . . . ;

LIBMAINT SL,
        LIB = SOURCES,
        COMFILE = *JOBSTREAM ;
$INPUT JOBSTREAM END=MATCH ;
MOVE COMFILE : STREAM2, REPLACE, TYPE=JCL ;
JOB A . . . ;
   .
   .
$ENDJOB ;
JOB B . . . ;
   .
   .
$ENDJOB ;
JOB LAST . . . ;
   .
   .
$ENDJOB ;
//EOD
$ENDINPUT JOBSTREAM ;
$ENDJOB ;
```

At the completion of this job the stored job stream can be started using the $RUN statement or the operator commands SJ or SI. The job stream can also be updated using $LIBMAINT.

6 • Sequence modification and error proccessing

Introduction

Level 4 GCOS contains system components and utilities which are designed to minimize the effects of serious errors which can occur for diverse reasons within the system.

For this purpose JCL can be included in a job description to alter the execution sequence in the event of a step abort. Thus the abort of a single step can be prevented from causing the entire job from being terminated. Step recovery aids include the periodic storage of executing process group structures and file journalizing, so that a step can be restarted from a known point prior to where an execution abort occurred. This facility is known as Checkpoint/Restart and File Journal.

Errors in application programs can be traced simply by means of a system component called the Program Checkout Facility (PCF). Each of these facilities is discussed briefly in this section and references are given to manuals containing detailed descriptions.

Error messages and return codes are generated by the system when an abnormal incident occurs in the execution of a job. These incidents are recorded on the JOR (Job Occurrence Report).

ERROR MESSAGES AND RETURN CODES

When the system detects a malfunction during the execution of a job, an error message is entered in the Job Occurrence Report. The malfunction may be due to a user error or a system error.

Error messages can be report messages with no qualification, or may be qualified as WARNING, FATAL or SYSTEM messages. SYSTEM messages refer to some malfunction of the system itself (either hardware or software) and are normally indicated by only a message code and a message number.

Return codes are also printed on the Job Occurrence Report. These codes normally indicate that some abnormal incident has occurred within the system.

A complete list of return code mnemonics appears in the Error Messages and Return Codes Manual along with probable causes and remedial action where appropriate.

Labelling a JCL Statement

A label can be associated with any JCL statement simply by preceding the statement with the label name and a colon.

For example :

```
$JOB . . . ;
   STIL : STEP LM4 . . . ;
             .
             .
          ENDSTEP ;
   .
   .
   .
   .
   .
$ENDJOB ;
```

The step LM4 can now be referenced in a $JUMP statement (by use of the name STIL).

THE $JUMP STATEMENT

The $JUMP statement allows the modification of the order in which JCL statements are handled.

The following rules apply :

- $JUMP statements outside step enclosures can only refer to labels outside step enclosures. A $JUMP used in this way changes the order in which steps are executed within a job. Jumping can be forward or backward.
- A $JUMP statement inside a step enclosure can only refer to a label inside the same step enclosure. Furthermore, jumping can then only be forward. Such a jump can be used, for example, to select various resources associated with the step.

The first parameter of a $JUMP statement is a label. The other parameters, which are optional, correspond to condition tests and will be described later on.

The following example illustrates the previous rules :

```
$JOB. . . ;
   $JUMP  A. . . ; (forward)
Step 1 ┌   $STEP LM1. . . ;
       │        $ASSIGN. . . ;
       │     $JUMP B. . . ; (forward)
       │        $ASSIGN. . . ;
       │     B :  $ASSIGN. . . ;
       └   $ENDSTEP ;
Step 2 ┌ A :  $STEP LM. . . ;
       └   $ENDSTEP ;
   $JUMP A. . . ; (backward)
$ENDJOB ;
```

Labels in an invoked JCL sequences are considered as local. In other words, a $JUMP statement in the job (or sequence) that contains the $INVOKE cannot reference a label

defined in the invoked sequence ; conversely a JUMP statement in the invoked sequence cannot reference a label defined in the job (or sequence) that contains the $INVOKE. This concept is illustrated in this section. The above comments also apply to JCL sequences referred to by $EXECUTE.

A set of 32 switches are associated with each executing job. They are named SW0 through SW31. At the beginning of job execution, they are all set to 0 unless the job is spawned (via the $RUN statement) or released (via the $RELEASE statement) by another job. In both of these cases, the switches can be set initially to any value by the other job. Each one can be set to 0 or 1 by means of the $LET JCL statement or by the executing load module (for example, using SET SWITCH-i in COBOL). They are not modified otherwise. Each one can be tested by a $JUMP JCL statement or by an executing load module (testing in COBOL SWITCH-i).

The following example illustrates a simple use of switches. Assume that LM1 sets SW5 to 1 when the end of a procedure is reached and that the job description is :

```
$JOB. . . ;
      LET  SW5, 0 ;
LOOP :    STEP LM1. . . ;
               ENDSTEP ;
      JUMP  FIN, SW5, EQ, 1 ;
               STEP LM2. . . ;
               ENDSTEP ;
      JUMP LOOP ;
FIN :     STEP LM3,. . . ;
               ENDSTEP ;
$ENDJOB ;
```

The first $LET statement guarantees that SW5 is set to 0 initially. Then the load module LM1 is executed. If it leaves SW5 at 0, load module LM2 is executed and then LM1 again. This continues until LM2 or LM1 sets SW5 to 1. At this stage the jump is performed to FIN and LM3 is executed. The job terminates after the execution of LM3.

Through this mechanism, one job can influence the execution of another (that it spawns using $RUN). Suppose that a job description stored in member EX12 of library L.JOBS is of the form :

```
      JUMP NEXT, SW0, EQ, 1 ;
               STEP LM1,. . . ;
               ENDSTEP ;
NEXT .    STEP LM2,. . . ;
               ENDSTEP ;
```

A spawning job :

```
$JOB. . . ;
      RUN EX12, L.JOBS ;
$ENDJOB ;
```

causes the stored job to execute both LM1 and LM2 (SW0 having been set to 0 originally), while a spawning job :

```
$JOB. . . ;
COMMENT 'SPAWN JOB EX12 SETTING SW0 TO 1' ;
    RUN  EX12,  L.JOBS,  SW=80000000 ;
$ENDJOB ;
```

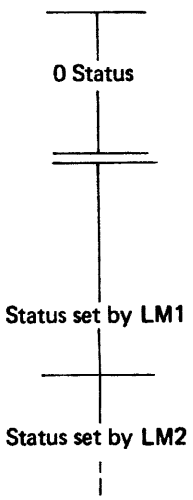
causes only LM2 to be executed (in the $RUN statement the values of all 32 switches are specified as a string of 8 hexadecimal characters). Note that the value of SW1 in the $RUN statement has no effect on the switches of the father job. The transfer of information from one job to another via switches in the $RUN statement can also be performed by means of switches in a $RELEASE statement (see $RELEASE in this section).

STATUS

The status is a decimal number which is associated with each executing job step. Its use is similar to that of switches but is more directly related to the overall results of the execution of a load module. It is set to 0 at the beginning of the execution of the load module, and its value can be modified under user control within the load module itself (for example, using a CALL H-CBL-USETST in COBOL) or by the System when it takes a decision about this execution. In the latter case, the status set by GCOS will override the value set by the user. The status can be tested by the $JUMP JCL statement, and its value remains unmodified until the next load module execution is started.

For example :

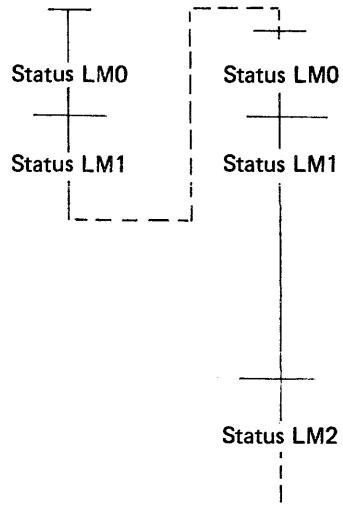
```
$JOB. . . ;
            STEP LM1 ;
                .
                .
FIRST :     ENDSTEP ;
            STEP LM2 . . . ;
                .
                .
SECOND :    ENDSTEP ;
```



The status set by LM1 may be tested between the $ENDSTEP statement labelled FIRST (which corresponds to LM1 execution) and the $ENDSTEP statement labelled SECOND (which corresponds to a new load module execution).

The scope of the status of a load module might not correspond to continuous statements if a $JUMP statement exists as in the following example :

```
REPEAT :    STEP LM0. . . ;
THIRD :     ENDSTEP ;
            STEP LM1. . . ;
FIRST :     ENDSTEP ;
            JUMP REPEAT, STATUS, NE, 200 ;
            STEP LM2. . . ;
SECOND :    ENDSTEP ;
```



If the status set by LM1 is equal to 200 it can be tested between statement FIRST and SECOND. If not the $JUMP statement will be effective and the status set by LM1 can be tested between statement FIRST and the $JUMP statement, and then between statements REPEAT and THIRD.

Thus the user can direct the flow of control of a job according to the execution of a particular load module.

The status, also referred to as the step completion code, can be set by the user to any values between 0 and 32767 ; other values are used for special cases by GCOS. Furthermore any value greater than 9999 will be interpreted by GCOS as meaning that the load module execution was incorrect and that the job execution should be aborted. Note however that it is possible by use of the JUMP statement to overcome this situation (see «Use of Status for Execution Abort» below). Status values are classified into groups. The names of these groups are SEV0 through SEV6 (for Severity 0 through 6) and can be used in $JUMP statements to test groups of values as in the following example :

```
$JOB. . . ;
            STEP LM1. . . ;
            ENDSTEP ;
      JUMP MESS, SEV, NE, 0 ;
            FILSAVE. . . ;
      JUMP CONTINUE ;
      JUMP FIN, SEV, EQ, 0 ;
MESS :      SEND 'FILE TRUC HAS NOT BEEN SAVED' ;
FIN :
$ENDJOB ;
```

if LM1 execution sets status to a value of severity 1 or more, the file save will not be attempted but the message FILE TRUC HAS NOT BEEN SAVED will be sent to the operator ; if the file save is unsuccessful (SEV=0) the same message is sent to the operator. Note the necessity for the JUMP CONTINUE in the case of an unsuccessful FILSAVE (refer to «Use of Status for Execution Abort» later in this Section). The user can set the value of SEV by means of the $LET statement.

Table 6-1 shows the relationship between a particular SEV grouping (with is corresponding status value range) and its significance to the system (for all SEV groups). Note that the interpretation of status values set under user control is defined by the user, but in all cases the system will interpret a value of 10000 or over (SEV3 or greater) as an abnormal condition. The significance of the status after the execution of a compiler is influenced by the fact that a compiler will always set the status according to the highest severity encountered during the compilation.

STATUS		Meaning	Job Occurrence Report	Consequences
Group	Value	(when produced by system)	message	
SEV0	0-99	Normal termination	TASK TERMINATED STEP TERMINATED	Execution terminated normally.
SEV1	100-999	Normal termination + WARNING	STATUS=SEV1 (or SEV2) or STATUS=numerical code	Job processing continues
SEV2	1000-9999	Normal termination +WARNING	(SEV1 or SEV2 is printed only if numerical code equals 100 or 1000 respectively)	
SEV3	10000 - 19999	Work not performed due to user error. Step is not repeatable.	STEP ABORTED STATUS=SEV3 (or SEV4) or STATUS=numerical code	REPEAT option (Checkpoint/Restart) : If operator enters YES, step repeated from last checkpoint. NOREPEAT option (no Checkpoint/Restart) : Scan JCL statements : if encounter : . $ENDJOB or step enclosure, then terminate job . $JUMP statement, then test condition ; if false, scan next statement (as above) if true, resume execution from given label.
SEV4	20000 - 32767	Work not performed due to GCOS problem or to external events (I/O Error). Step is repeatable.	(SEV3 or SEV4 is printed only if numerical code equals 10000 or 20000 respectively)	
SEV5	50000	Abort requested by the operator (Terminate Job command)	STEP ABORTED BY OPERATOR STATUS=SEV5	
SEV6	60000 61000	An exception occurred in a system procedure System Crash	STEP ABORTED BY SYSTEM STATUS=SEV6	

Table 6-1. Step Termination Conditions

Use of Status for Execution Abort

If after the execution of a step the status value is greater than 9999 (in other words the severity is greater than 2), the step is considered to be abnormally terminated. From that point on, the command interpreter scans JCL statements in sequential order, ignoring them until one of the following conditions happens :

- $ENDJOB or $STEP or any other job enclosure statement (except $JUMP CONTINUE or a $JUMP which tests the status is encountered). The job execution is abnormally terminated.
- A $JUMP statement tests the status (using STATUS or SEV) and the condition is true. In this case the $JUMP statement is executed and normal execution is resumed. If the condition is false, the job aborts.
- A $JUMP statement with the reserved label CONTINUE is encountered. This signifies that execution is to proceed in sequence.

Consider the following job :

```
$JOB. . . ;
          STEP LM1,. . . ;
          ENDSTEP ;
          STEP LM2,. . . ;
          ENDSTEP ;
     JUMP ABNORM,SEV,GE,3 ;
     SEND 'EXECUTION OK' ;
          STEP LM3,. . . ;
          ENDSTEP ;
     JUMP END ;
     ABNORM : STEP LM4. . . ;
          ENDSTEP ;
   END :
$ENDJOB ;
```

If steps LM1 and LM2 terminate normally, the message «EXECUTION OK» will appear on the operator's console and step LM3 will be started.

If LM1 execution is abnormally terminated the command interpreter will abort the job when it encounters the next step (here step LM2).

If LM1 terminates normally but LM2 execution is abnormally terminated the Command Interpreter will skip over the $SEND statement, then execute the jump to ABNORM and the job execution will resume from there (load module LM4).

Another example is :

```
$JOB. . . ;
     COBOL. . . ;
     JUMP CONTINUE ;
     COBOL. . . ;
     JUMP CONTINUE ;
     COBOL. . . ;
$ENDJOB ;
```

Each COBOL statement is an extended statement corresponding to a step requiring execution of the COBOL compiler. In this job all three steps will be executed even if the first or second one discovers a problem and sets the status to a value which would normally cause the job to be aborted.

Setting Severity Value within an Invoked JCL Sequence

The $LET statement with SEV parameter can be used to simulate an error condition within an invoked JCL sequence, thus allowing the processing sequence to be altered after the $INVOKE statement in the original job description. Suppose, for example, an invoked sequence contains three step descriptions, the first step of which is to be executed in all cases ; only one of the two remaining steps is to be executed, depending whether the first step terminates normally (severity less than 3). In other words, the success or failure of the first step determines which of the two steps will be executed. Assuming that the second step to be executed terminates normally, the severity value at the end of the JCL sequence (invoked) and therefore applying to the statement, will always be less than 3. A $JUMP statement after the $INVOKE cannot take account of the severity code of the first step in the invoked sequence. The solution to this problem is to set the severity to 3 or more at the end of the step (in the invoked sequence), which is executed in the event of an abnormal termination of the first step. This concept is illustrated in Fig. 6-1.

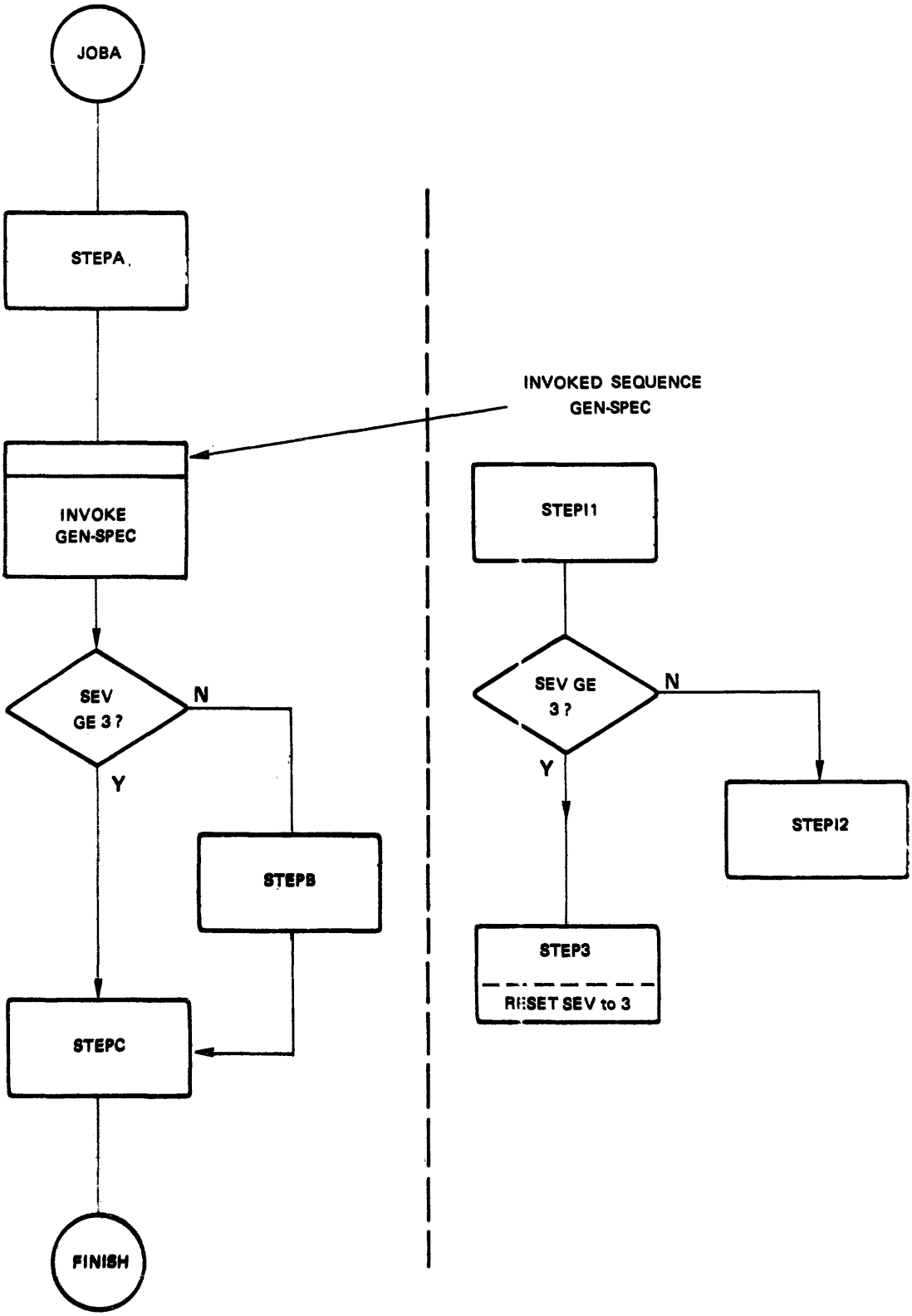


Figure 6-1. Use of $LET SEV

Figure 6-2 contains the JCL statements that correspond to the situation shown in Figure 6-1. If STEPI1 terminates normally, STEPI2 is executed and provided it also terminates normally, a severity of 0 applies to the $JUMP after the $INVOKE. In this case steps STEPB and STEPC are executed. If, however, STEPI1 aborts STEPI3 is executed ; the $LET statement ensures that a severity of 3 applies to the $JUMP statement after the $INVOKE, irrespective of the result of the execution of STEPI3, in this case only STEPC is executed next.

```
$JOB JOBA,. . . . . ;
  STEP STEPA,. . . . . ;
     .
     .
     .
  ENDSTEP ;
  INVOKE GEN.SPEC. . . . . ;
  JUMP LAST,SEV,GE,3 ;
  STEP STEPB,. . . . . ;
     .
     .
     .
     .
  ENDSTEP ;
  LAST : STEP STEPC. . . . . ;
     .
     .
        ENDSTEP ;
$ENDJOB ;
```

Contents of GEN.SPEC

```
     STEP STEPI1. . . . . ;
        .
        .
        .
        .
     ENDSTEP ;
       JUMP ERR,SEV,3 ;
     STEP STEPI2 ;
        .
        .
        .
     ENDSTEP ;
       JUMP END ;
     STEP STEPI3 ;
        .
        .
        .
        .
     ENDSTEP ;
  LET SEV,3 ;
    END : JUMP CONTINUE ;
```

Figure 6-2. Example of the Use of $LET

$RELEASE STATEMENT

The $RELEASE statement in a job enclosure can be used to release a job which has been suspended (by a JCL HOLD option in $JOB or operator HJ). The job in which the statement appears and the job to be released must have the same characteristics, namely USER and PROJECT. The statement has a single keyword, SWITCHES which has two parameters, hex-string and PASS. SWITCHES is used to initialize each of the 32 job switches associated with each job, to a required value when the job is released. The PASS parameter causes the current switch value of the releasing job to be assumed by the released job. If the SWITCH parameter is omitted from $RELEASE then all 32 switches are set to zero.

An example of the use of $RELEASE is given in the paragraphs in this section entitled Control of Interdependent Jobs.

CONTROL OF INTERDEPENDENT JOBS

The testing of switch or status values can be use to control the order in which interdependant jobs are executed. The user can do this by interspersing the JCL description with $JUMP, $RUN and $RELEASE statements. The facility is useful in cases where the execution sequence is dependent upon successful completion of other jobs in the same stream. Rather than burden the operator with the responsibility of managing the interdependencies, the system handles them by selectively spawning the JCL descriptions.

Figure 6-3 shows a flow chart illustrating interdependencies between several jobs.

Figure 6-4 shows the JCL that might be used to produce the required dependencies.

Note that the processing done in hold job JOBB3 depends on the execution of job JOBA, which releases JOBB3. The value of SW2 in JOBA at the time it releases JOBB3 determines whether JOBC30 or JOBC31 is spawned by JOBB3.

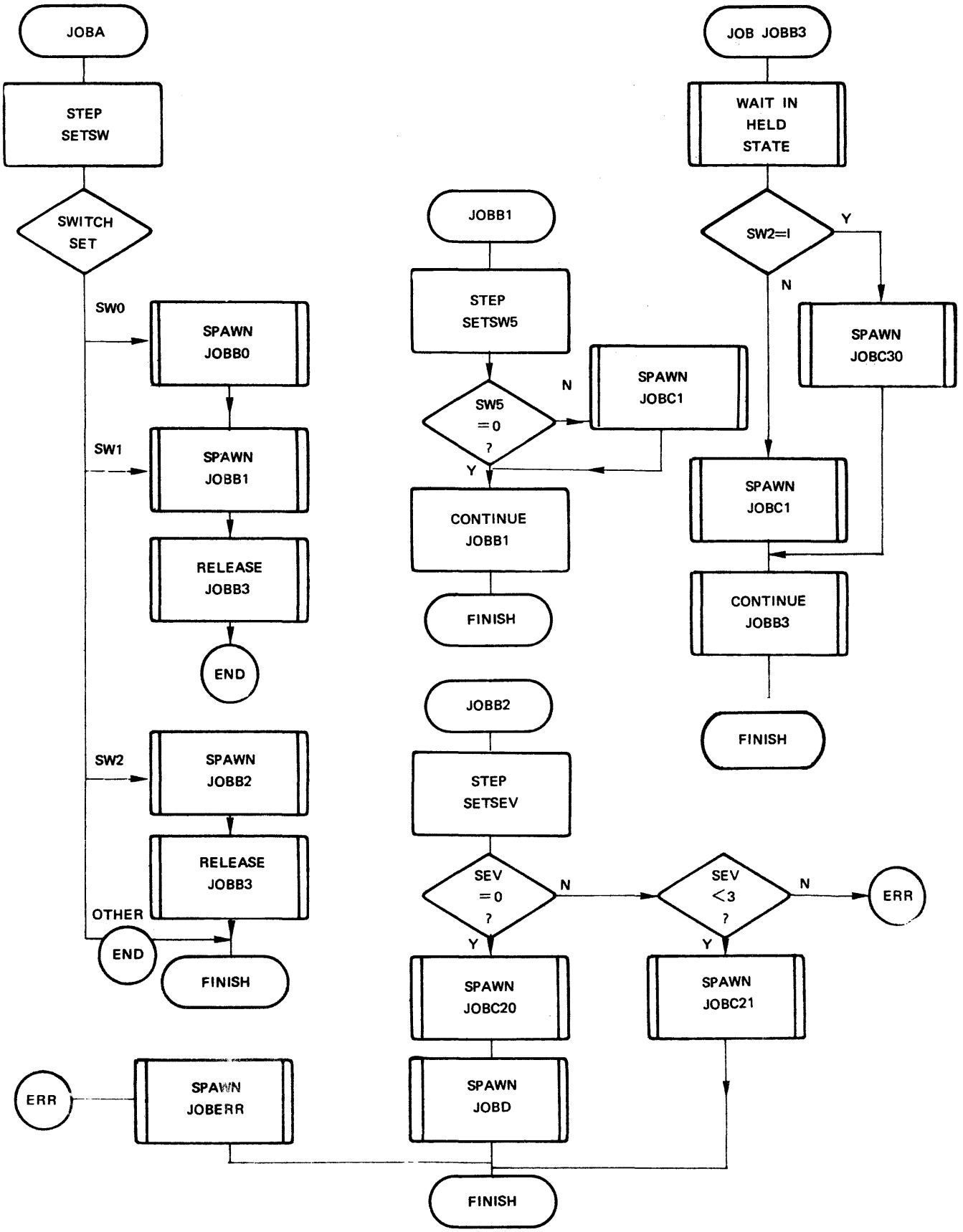


Figure 6-3. Interdependency of Jobs

```
$JOB JOBA, . . . ;
        .
        .
   LET SW, hex-string :
   STEP SETSW, . . . ;
        .
        .
   ENDSTEP ;
  COMM 'SPAWN JOBB0 AND JOBB1 IF SW0 SET' ;
   JUMP BORUN, SW0, EQ, 1 ;
  COMM 'SPAWN ONLY JOBB1 IF SW1 SET' ;
   JUMP BIRUN, SW1, EQ, 1 ;
   JUMP ENDA, SW2, NE, 1 ;
  COMM 'SW2 SET SO SPAWN JOBB2' ;
   RUN JOBB2, . . . ;
   RELEASE JOBB3, SWITCHES=PASS ;
   JUMP END;
  BORUN : RUN JOBB0, . . . ;
  BIRUN : RUN JOBB1, . . . ;
  RELEASE JOBB3, SWITCHES=PASS ;
  ENDA :
 $ENDJOB ;
```

stored job JOBB1

```
      .
      .
LET SW5, binary digit ;
STEP SETSW5, . . . ;
      .
      .
      .
ENDSTEP ;
JUMP NOSPN, SW5, EQ, 0 ;
RUN JOBC1, . . . ;
NOSPN : STEP NEXT, . . . ;
   . . . .
   . . . .
```

Held Job JOBB3

```
$JOB, JOBB3, . . . . . , HOLD ;
JUMP C31RUN,SW2,NE,1 ;
     RUN JOBC30, . . . ;
JUMP NEXT ;
C31RUN : RUN JOBC31
NEXT :
      .
      .
      .
      .
      .
      .
```

stored job JOBB2

```
      .
      .
      .
   STEP SETSEV, . . . ;

   ENDSTEP ;
   JUMP C20, SEV, EQ, 0 ;
   JUMP C21, SEV, LT, 3 ;
COMM 'ERROR PROCESSING HERE' ;
   RUN JOBERR, . . . ;
   JUMP ENDSPN ;


COMM 'SPAWN JOBC21 HERE' ;
C21 : RUN JOBC21, . . . ;
   JUMP ENDSPN ;
COMM 'SPAWN JOBC20 HERE' ;
C20 : RUN JOBC20, . . . ;
   RUN JOBD, . . . ;
ENDSPN : ENDJOB ;
```

Figure 6-4. Example of Interdependent Jobs

CHECKPOINT/RESTART AND FILE JOURNAL

A pair of complimentary facilities known as Checkpoint/Restart and File Journal are provided as recovery aids when a step abnormally terminates due to some incident. The first facility stores at intervals defined by the user an image of all step information which would be required to restart the step if an abnormal termination occurred. The second mechanism allows files to be restated to their exact state prior to step execution in the event of abnormal step termination.

The following paragraphs are intended to give a general introduction to the purpose and use of Checkpoint/Restart and Journalization. The System Management Guide describes at length the principles and theory of the facilities, while guidance on their use in connection with a particular programming language is given in the appropriate language guide, for example the COBOL User Guide.

Taking of Checkpoints

Use of the REPEAT parameter of $STEP (or $JOB) initiates the Checkpoint/Restart mechanism. A checkpoint is established prior to the execution of the first instruction of the step.

Checkpoints can also be taken during the execution of a step. In order to achieve this, the user must include in his program calls to the Checkpoint/Restart mechanism H-CK-UCHKPT. These calls can be initiated either by explicit statements in the program or each time a specified number of records of a named file have been processed (refer to System Management Guide).The latter is achieved by use of the keyword CKPTLIM in the $DEFINE statement and specifying the number of records to be read between each checkpoint.

Calls to H-CK-UMODE in the user program can be used to obtain reports on the current checkpoint status of a step, without causing the checkpoint image to be updated to that point.

Note :

When a step is restarted, it always restarts from the most recent successful checkpoint i.e. in the case of a checkpoint within a step from the last one that occurred in the execution of the step before the abort occurred.

File Journal

The File Journal facility is used to restore the contents of a file processed by an abnormally terminated step to those of the last checkpoint. In cases where no checkpoints occur within a step the file is restored to its state at beginning of the step. The File Journal is used when the re-positioning of a file is not sufficient to return it to its condition prior to step termination. The Journal facility is implemented by use of the JOURNAL option of $DEFINE. The JOURNAL option has two parameters BEFORE and AFTER.

BEFORE JOURNAL

BEFORE is used to restore the contents of a file to those of the last checkpoint, for example :

```
.
.
ASSIGN RTS, LM-LD ;
DEFINE RTS, BLKSIZE=.....
     JOURNAL=BEFORE ;
.
.
```

allows the contents of the file RTS to be «Rolled-Back» to those of the last checkpoint. If the step in which the file is processed is non-repeatable (i.e. the REPEAT parameter is not present in the $STEP statement), the file can still be «rolled-back» if the step aborts ; or at a Warm Restart after a system crash.

AFTER JOURNAL

The AFTER parameter of JOURNAL is a system file on tape or disk in which information is stored concerning all writing operations on a file. This enables the contents of a file to be restored to those of the previous save. The AFTER Journal can only be used in connection with Transaction Driven System sessions, to ensure that the integrity of a file is maintained in the event of abnormal termination of a session.

Notes :

1. The Journal is only available for a single process load module. It cannot be used with a multiprocess load module and in particular a telecommunications load module.
2. A journalized file must not be closed with a deassignment (i.e. a COBOL, CLOSE WITH LOCK), otherwise a roll-back attempt will lead to unpredictable results.

JOURNALIZED FILE ORGANIZATIONS

Table 6-2 shows when the File Journal can be used, with respect to file organization and processing mode.

File organization	Processing mode		
	Output	Append	Update
UFAS			
Sequential tape	NO	NO	-
Sequential disk	NO	YES	YES
Relative	YES	--	YES
Indexed	NO	NO	YES
BFAS			
Sequential tape	NO	NO	–
Sequential disk	NO	NO	YES
Direct	YES	–	YES
Indexed sequential	NO	NO	YES
HFAS			
Random	YES	--	YES
Indexed sequential	NO	NO	YES
Sequential disk	NO	NO	YES
Sequential tape	NO	NO	

Table 6-2. Availability of File Journal

Repeating a Step

When the execution of a step is abnormally terminated and if the checkpoint/Restart facility has been implemented then the operator will be advised to restart the step from the last checkpoint. See System Operator Guide for full details. The system allows the following options :

- Step restart from last checkpoint with or without Rollback
- Just to rollback file (journalized)
- To terminate the step
- To suspend the step by use of HJ (ron).

The operator may take corrective action also, for example by mounting a disk on a different drive.

Warm Restart

The warm restart facility minimizes the effects of a system crash due a Power Off or a hardware/software failure. This is achieve by making it possible to recover all jobs that were in the IN/SCHEDULING/HOLD/OUT STATES. Jobs that were executing at the time of the system crash, with the repeat option of $STEP (or $JOB), unless NRESTART is specified, and with at least one journalized file, are restarted if the operator requests it. See the System Management Guide.

Note :

Utilities are available to the user, namely $FILCHECK, $VOLCHECK and $CATCHECK, which allow :

- The labels ($FILCHECK) of a file to be read to ensure that the file can be processed by a standard access method (available for sequential and library files only).
- The volume table of contents (VTOC) to be checked (VOLCHECK).
- The catalog structure to be checked ($CATCHECK).

PROGRAM CHECKOUT FACILITY (PCF)

A source program which seldom runs perfectly the first time, is usually corrected and improved by a process of trial-and-error in a man/machine dialogue. This method is lengthy and might be extremely expensive in terms of human and machine resources. The Program Checkout Facility is a resident system program which assists the programmer to debug application programs written for Level 64 GCOS. The PCF accepts commands from a user, specifying program checkout functions such as DUMP and TRACE. The commands generally refer to source language items, in COBOL and FORTRAN for instance, which are to be subject to a particular checkout procedure.

The benefits to be derived from the use of PCF are :

- the implementation of a debugging method with powerful tools (such as trace)
- the avoidance of time-consuming clerical errors
- a clear and easy-to-read debugging report stated in source language terms
- a reduction in the number of compilations needed to obtain a productive program.

Use of PCF

The user implements the PCF in three phases. Initially the application program is compiled with the DEBUG parameter present in the appropriate JCL statement.

For example :

```
FORTRAN SOURCE=*MYDECK, DEBUG ;
```

which states that a FORTRAN program on cards in an input enclosure is to be compiled in debugging mode.

```
COBOL SOURCE=MYPROG,INLIB=(MYLIBRARY),DEBUG ;
```

names a source COBOL program in a source program library, MYLIBRARY, which is to be compiled in debugging mode.

The compiler is thus instructed to keep track of program source elements, and to generate a table giving the mapping of source line numbers, labels and variables on memory locations, which is then incorporated in the compile unit.

The second phase takes place when the user wishes to start a checkout session. This is achieved by stating the keyword DEBUG at JCL step level, and providing a description of the file from which the PCF instructions are to be read.

For example :

```
    STEP.....,DEBUG;
    ASSIGN H_DB,*CARDS;
    ENDSTEP;
$INPUT CARDS;
                    PCF commands on cards
$ENDINPUT;
```

This example illustrates the association of an internal file name, H_DB, with PCF instructions on cards.

The PCF commands are read and interpreted. Breakpoints are inserted in the object program, by use of the tables generated during the initial compilation phase. The final phase is then entered in which the program is executed, branching from normal execution sequence occurring at the inserted breakpoints to carry out the specified debugging procedures.

The contents of the PCF command file is discussed fully in the PCF manual.

Notes :

1. If the source program is compiled without the DEBUG option the user may still use PCF but in this case the user must generate the actual core references of the action and object points to which he refers.
2. Only COBOL and FORTRAN source programs can be compiled using the DEBUG option. RPG and GPL may be debugged if the user supplies all the core references. Refer to the PCF User Guide.

7•Job occurrence report

For each job run on the system, a printed report is produced, called the Job Occurrence Report (JOR). The JOR is automatically printed by the Output Writer. The JOR is subdivided into categories of messages according to the phase, during the job processing, in which the information is sent to the JOR :

- Job Introduction and JCL Translation
- Job Execution

In general, a report message is a 120-character line, in which are listed the JCL input, job accounting information, and the body of the JOR messages describing the result of each command, whether successful or not. The JOR messages begin in column 11. The date of the report is given in the same line as the report title. When the time of a particular operation is given, it appears in columns 1 to 8 of the line corresponding to the relevant message. In addition, for accounting purposes, the start and terminate times for the job are given in the report.

The general message format is :

- Columns 1 through 10 may contain one of the following character strings, left-justified : FATAL, SYSTEM, WARNING, hh:mm:ss, a string of blanks.

  Here, hh:mm:ss represents a clock time in hours:minutes:seconds, «FATAL» and «WARNING» indicate user errors, and «SYSTEM» indicates an abnormal system situation.
- Columns 11 through 120 contain the body of the message.
  An error message is designated by the message name (e.g. IN01) appearing as the first item in the message.

JOB INTRODUCTION AND JCL TRANSLATION

The first printed material produced for the job is the Output Writer Banner (see Figure 7-1). This contains :

1 Date on which the JOR was printed.

2 Time at which the JOR was printed.

3 Run Occurrence Number or the job.

4 User-id.

5 Job-id.

6 Project.

7 Software Release.

8 Version of the System shared Modules.

9 Version of the System Load Module.

10 Firmware Version.

11 Machine Identification.

12 Printer Name (on which the report has been printed).

13 Printer Characteristics.

Following the Banner comes the Job Introduction and JCL Translation Listing. It is headed by a line giving the identification of the job, user, account and Run Occurrence Number. This may be followed by a «Message of Today» (MOT) line, which has been established by the operator.

An example of this listing is given in Figure 7-2 and an explanation of the various types of information printed follows here :

14 Time at which translation started.

15 Listing of Source JCL.
The record count is the number of records read. (Note invoked JCL is counted as 2 records).

16 Time at which translation terminated.

17 Input medium for source JCL, with time and date of introduction (CD - card reader, MS - disk drive, MT - tape drive).

18 There are two types of JCL «error». WARNING indicates that default action has been taken. The user should verify that this is the intended action. FATAL indicates that the error is more serious and the job will be aborted at the end of translation.

19 As for 8 except that the fatal error resulted in the job being aborted.

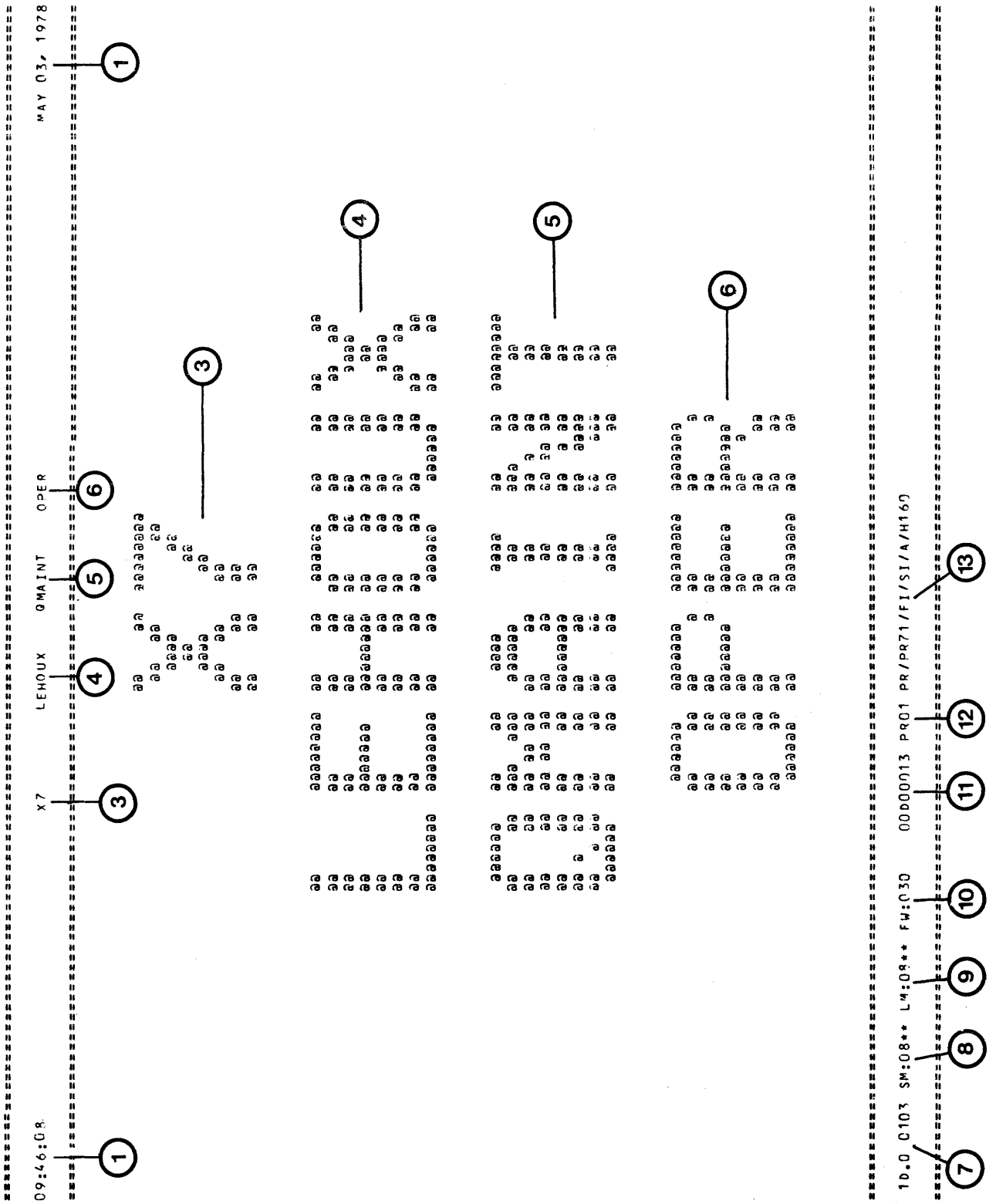


Figure 7-1. Sample Output Writer Banner

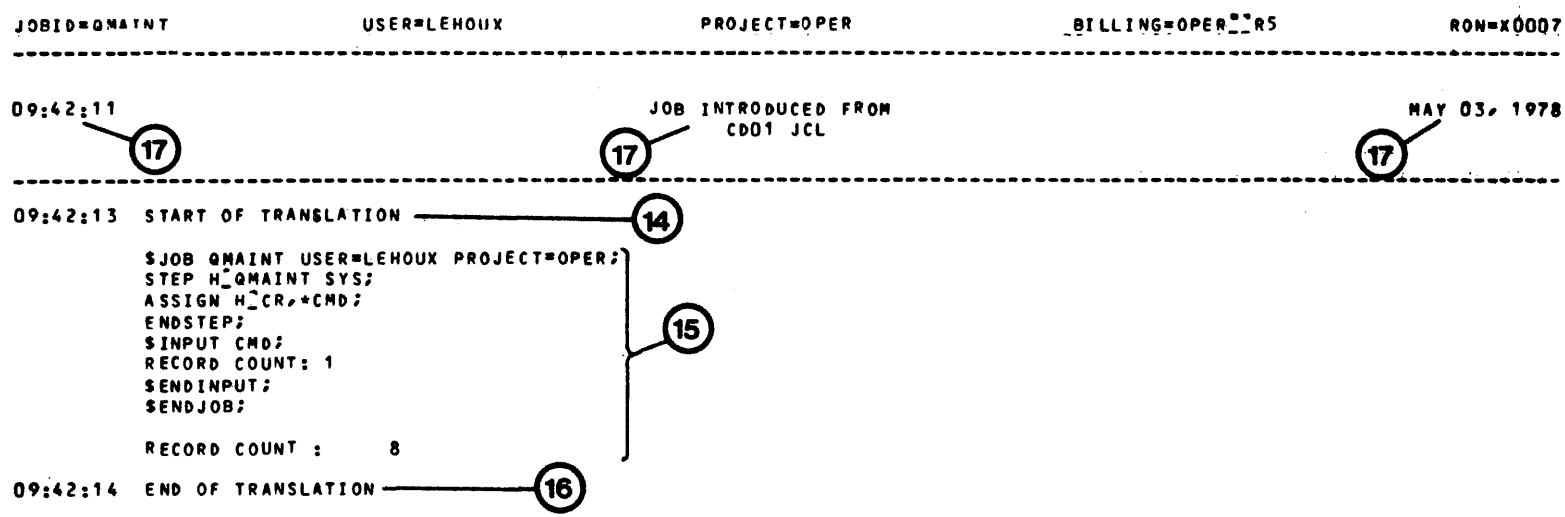


Figure 7-2. Sample Job Introduction and JCL Translation Listing

JOB EXECUTION

The Job Execution messages provide information about the file status and program status and functioning, and of problems and errors encountered during execution of the various job steps.

The corresponding listing (see Figure 7-3) contains :

(20) Step number. If this step is a Utility, the name of the Utility is also given. This is followed by the name of the load-module (which is the same as the subfile name) and if the load-module has been preinitialized this will also be indicated ; and finally, the name of the library from which the load-module was initialized. If a JUMP is made to a previous step, the original number of that step will appear.

(21) The time at which the step execution began, and the J and P numbers and result of the task (J and P are for debugging use in the event of a system failure).

(22) The number of I/O connects for each file (1 per block transfer + file open, close, etc.). The example shown includes the number for the JOURNAL file since the Before Journal is used here.

(23) STACKOV : Number of stack overflows.
The stack contains address information for each segment call. A high value here may indicate, for example, that a program loop has been split between two segments.
MISSING SEGMENTS : Number of segments that have been swapped in.
BACKING STORE : Total size (in bytes) of segments in backing store. It does not include the space for a preinitialized load module.
BUFFER SIZE : Maximum buffer space used.
CPSIZE : Channel program size.

(24) CPU time in minutes.
Elapsed time in minutes.
Number of lines of SYSOUT printer output with the limit, if any.
Number of punched cards (SYSOUT) with the limit, if any.
Time at which the step terminated.

(25) Same as 24, but in this case for the job.

(26) Reason for abnormal step termination (in this case, see «Messages at Restart Time» below).

(27) Messages writen on the execution report in connection with the file access method (in this case for a BFAS indexed sequential file) ; and, in connection with Restart.

(28) Notification of file rollback.

(29) Time of attempted warm restart.

(30) The number of the checkpoint at which the step is restarted.

(31) The disk space occupied by the Journal file.

(32) Information about the checkpoints taken during the step.

```
JOBID=QMAINT          USER=LEHOUX          PROJECT=OPER_          BILLING=OPER__R5          RON=X0007
-------------------------------------------------------------------------------------------------------

12:22:15                                JOB EXECUTION LISTING                          MAY 09, 1978

-------------------------------------------------------------------------------------------------------

          STEP 1
          LOAD MODULE = H_CNC (78/4 /28 10:54)
          LIBRARY = MCS.LMLIB (MD=C122 )
12:22:22  STEP STARTED XPRTY=8
          TASK MAIN J=03 P=00 COMPLETED
          RC TRACE :    RC=4BC6100E->DFPRE  6,TPUNKN   AT 1C3A19A0          RC=4883100E->DFASG  3,TPUNKN   AT 1BB519CC
                        RC=4BCA100E->DFPRE 10,TPUNKN   AT 1C07C15A          RC=4BC6100E->DFPRE  6,TPUNKN   AT 1C3A19A0
          H_QC_FMS  ON  C122    :  NB  OF  IO  CONNECTS=          22
          H_PR      ON  EXP628  :  NB  OF  IO  CONNECTS=           9
          H_CR      ON  EXP628  :  NB  OF  IO  CONNECTS=           1
          CPU          0.028                   PROG MISSING SEGTS       14   STACKOV      51
          ELAPSED      0.105                   SYS  MISSING SEGTS       15
          LINES          66   LIMIT     NOLIM  BACKING STORE         156288
          CARDS           0   LIMIT     NOLIM  BUFFER SIZE            8256   CPSIZE     2272
12:22:29  STEP COMPLETED SEV2


          STEP 2
          LOAD MODULE = TQC_INIT (78/3 /17 09:51)
          LIBRARY = TEST.LMLIB (MD=C122 )
12:22:32  STEP STARTED XPRTY=8
          TASK MAIN J=03 P=00 KILLED
          RC TRACE :    RC=20181878->VMM   24,NOMATCH  AT 0AA402AC          RC=4FDB1008->DQULK 27,SFNUNKN  AT 1AD40448
                        RC=4FD01008->DQULK 16,SFNUNKN  AT 1AE002FC          RC=4BCA100E->DFPRE 10,TPUNKN   AT 1C07015A
          FP07.IFN=H_QC_FMS HAS BEEN CLOSED BY SYSTEM
          FP07.IFN=H_TST_IN HAS BEEN CLOSED BY SYSTEM
          FP07.IFN=H_PR     HAS BEEN CLOSED BY SYSTEM
          H_DPPR    ON  EXP628  :  NB  OF  IO  CONNECTS=         129
          H_QC_FMS  ON  C122    :  NB  OF  IO  CONNECTS=          74
          H_TST_IN  ON  EXP628  :  NB  OF  IO  CONNECTS=           2
          H_PR      ON  EXP628  :  NB  OF  IO  CONNECTS=          12
          CPU          0.235                   PROG MISSING SEGTS       57   STACKOV      19
          ELAPSED      0.808                   SYS  MISSING SEGTS       32
          LINES         2369   LIMIT     NOLIM  BACKING STORE          39072
          CARDS            0   LIMIT     NOLIM  BUFFER SIZE           11040   CPSIZE     2272
12:23:21  STEP KILLED
          CHECKPOINT   1 TIMES CALLED
          CHECKPOINT  LARGEST  SNAPSHOT  LENGTH       50352


          STEP 2 RESTARTED AT CHECKPOINT 1
          SHARED MODULE = H_SM1
          LIBRARY = MCS.SMLIB
12:23:22  STEP STARTED XPRTY=8
WARNING   FP21.RESTART OF IFN "H_QC_FMS": NOT ASSIGNED ANY MORE .RC=00000000->DONE
          ELAPSED      1.320
12:24:42  STEP   ABORTED    BY SYSTEM CRASH
```


Figure 7-3. Job Execution Listing Format

```
          LIST OF FILES ASSIGNED AT CRASH TIME
          EFN                                          VSN     PMD SALVAGED NEEDED
          MS_QUEUE                                     C122    IN  NO       NONE
          STEP 2 RESTARTED AT CHECKPOINT 5
          SHARED MODULE = H_SM1
          LIBRARY = MCS.SMLIB
12:26:26  STEP STARTED XPRTY=8
WARNING   FP21.RESTART OF IFN "H_QC_FMS": NOT ASSIGNED ANY MORE .RC=00000000->DONE
          TASK MAIN J=03 P=00 COMPLETED
          RC TRACE :    RC=4BCA100E->DFPRE 10,TPUNKN   AT 1C07015A            RC=4B83100E->DFASG  3,TPUNKN   AT 1BB519CC
                        RC=4B83100E->DFASG  3,TPUNKN   AT 1BB519CC            RC=4BCA100E->DFPRE 10,TPUNKN   AT 1C07015A
          FP07.IFN=H_TST_IN HAS BEEN CLOSED BY SYSTEM
          FP07.IFN=H_PR     HAS BEEN CLOSED BY SYSTEM
          FP07.IFN=H_QC_FMS HAS BEEN CLOSED BY SYSTEM
          H_TST_IN  ON  EXP628 :  NB  OF  IO  CONNECTS=          8
          H_PR      ON  EXP628 :  NB  OF  IO  CONNECTS=         49
          H_QC_FMS  ON  C122   :  NB  OF  IO  CONNECTS=        164
          CPU          0.347                        PROG MISSING SEGTS      27   STACKOV      18
          ELAPSED      0.380                        SYS  MISSING SEGTS      47
          LINES          305   LIMIT     NOLIM      BACKING STORE            0
          CARDS            0   LIMIT     NOLIM      BUFFER SIZE          12016   CPSIZE     2272
12:26:49  STEP COMPLETED
          CHECKPOINT   0 TIMES CALLED ——(30)

          START    12:22:15   LINES       2740
          STOP     12:26:49   CARDS          0
          CPU         0.611                    ——(25)
          ELAPSE      4.567
12:26:49  RESULT:  JOB COMPLETED
```

Figure 7-3. (cont.) Job Execution Listing Format

```
JOBID=EDIT          USER=ABITBOL          PROJECT=JOBM   BILLING=JOBM          RON=X007

17 : 30 : 10            JOB EXECUTION LISTING                          MAY 25, 1977


  STEP 1 LIBMAINT
  LOAD MODULE = H.LIBMAINT (78/1 /30 10350) PREINITIALIZED
  LIBRARY = SYS.HLMLIB
17 : 30 : 37     STEP STARTED
  TASK MAIN J=02 P=00 NORMALLY COMPLETED
  H.PR       ON  SR3A91   : NB  OF  IO  CONNECTS=912
  COMFILE    ON  ER3A91   : NB  OF  IO  CONNECTS=  1
  INLIB1     ON  B054     : NB  OF  IO  CONNECTS=326
  INLIB2     ON  B054     : NB  OF  IO  CONNECTS= 22
  INLIB3     ON  B054     : NB  OF  IO  CONNECTS= 11
  LIB        ON  B054     : NB  OF  IO  CONNECTS= 98
  CPU            0.504                  PROG MISSING SEGTS      57    STACKOV     22
  ELAPSED        0.919                  SYS SEGMENTS            24
  LINES          13252    LIMIT  NOLIM  BACKING STORE            0
  CARDS              0    LIMIT  NOLIM  BUFFER SIZE           12752   CPSIZE    1536
17 : 31 : 33     STEP COMPLETED STATUS=SEV1

  STEP 2
  LOAD MODULE = H.EDITACT
  LIBRARY = SYS.HLMLIB
17 : 31 : 40     STEP STARTED
  TASK MAIN J=02 P=00 NORMALLY COMPLETED
  USERACTIF  ON  2125     : NB  OF  IO  CONNECTS= 123
  USER       ON  SR3A91   : NB  OF  IO  CONNECTS=6046
  CPU            0.968                  PROG MISSING SEGTS      42    STACKOV     13
  ELAPSED        1.882                  SYS SEGMENTS            22
  LINES          44642    LIMIT  NOLIM  BACKING STORE        39072
  CARDS              0    LIMIT  NOLIM  BUFFER SIZE           10336   CPSIZE    2256
17 : 33 : 32     STEP COMPLETED

  START      17 : 30 : 31    LINES  57894
  STOP       17 : 33 : 33    CARDS      0
  CPU          1.472
  ELAPSE       3.034
17 : 33 : 33     RESULT : JOB COMPLETED
```

Figure 7-4. Job Execution Listing Example

In the above example the library SYS.HLMLIB is specified in the FILE parameter of the $STEP statement. The value between the statements STEP STARTED and STEP COMPLETED represent the accounting information which describes the step execution. The ELAPSE parameter expresses the clock time in thousandths of a minute which elapsed during the execution of the step. The PROG MISSING SEGTS statement for STEP 1 indicates that a total of 24 instances occurred involving the swapping in of a program segment from backing store to memory, while the SYS MISSING SEGTS gives the same information for system segments used by the program ; in addition there were 22 occurrences of stack overflow (STACKOV) in that step. The statements BUFFER SIZE and BACKING STORE express the number of bytes within the user buffers and backing store respectively. CPSIZE gives the ammount of space used by the channel programs. In addition, the number of physical I/O «connects» and the number of log entries made (I/O retries) are listed for each assigned file.

OUTPUT WRITER END BANNER

A banner, similar to the one written at the beginning of the JOR, is written at the end of the JOR by the output writer. The format of this end banner is shown if figure 7-5.

The only difference between the end banner and the one written at the beginning is that at the foot of the page (area 33) information on the printing of the JOR appears in the form of the following messages :

OUTPUT HELD	The output has been retained on SYS.OUT by the system due to a fault on the printer.
OUTPUT HELD BY THE OPERATOR	The output has been retained on SYS.OUT by the operator (HO command).
OUTPUT CANCELLED	The output has been cancelled by the system due to a severe and irrecoverable error.
OUTPUT CANCELLED BY THE OPERATOR	The output has been cancelled by the operator (CO command).
WRITER TERMINATED	All output writer activities have been terminated by the system owing to a severe and irrecoverable error.
WRITER TERMINATED BY THE OPERATOR	All output writer activities have been terminated by the operator owing to a severe and irrecoverable error (TO command).

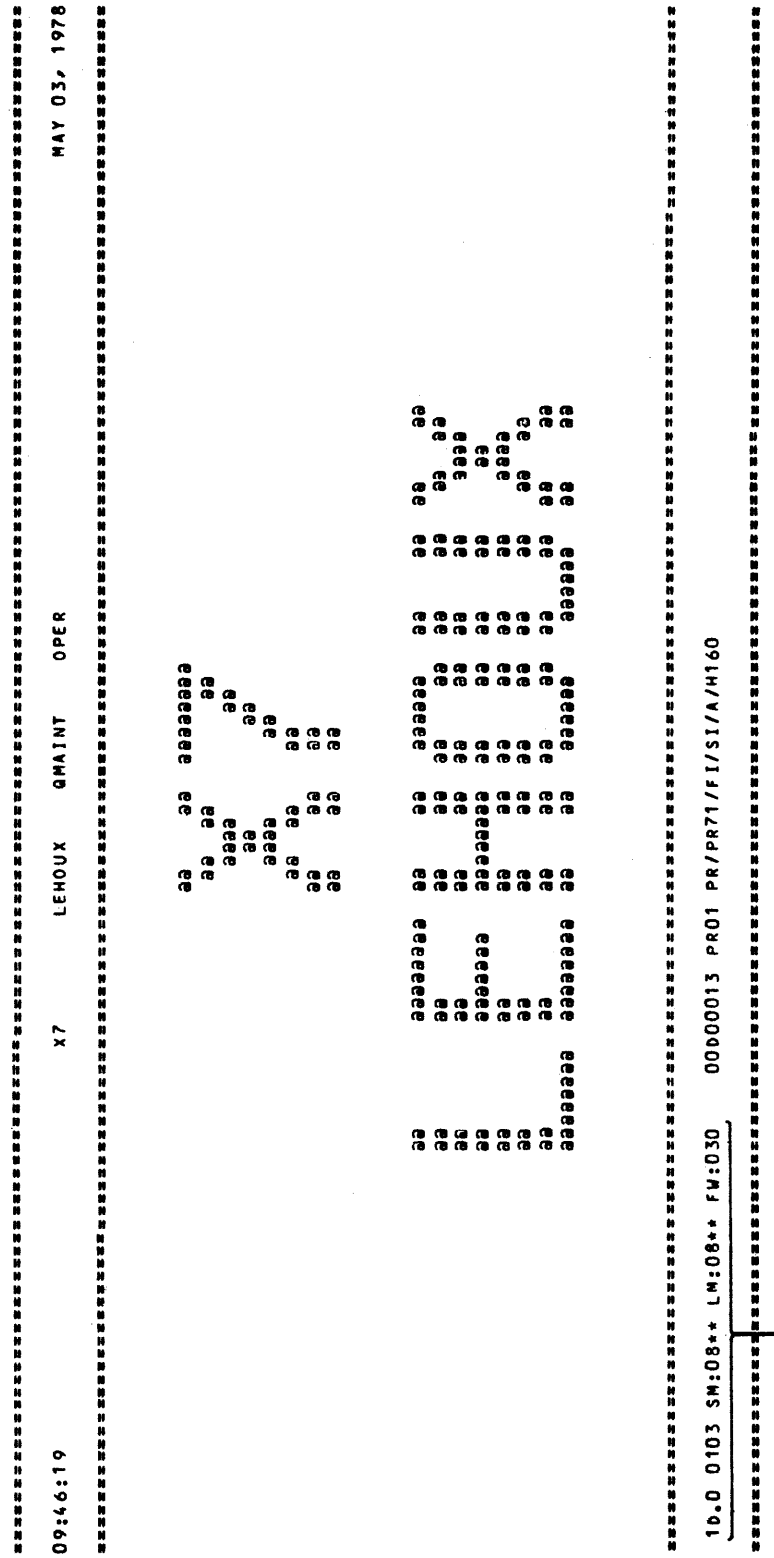


Figure 7-5. Output Writer end Banner

Job Execution Messages

The various types of messages produced during the execution of a job are described in the following paragraphs. Refer to Error Messages and Return Codes manual.

WARNING AND ERROR MESSAGES

A warning or error message is written in the JOR when the system detects an abnormal condition concerning the job processing. A warning message may be produced when actual processing conditions are inconsistent with the expected conditions, but are not severe enough to prevent the continuation of processing. An error message reflects a condition which may be due to a user, operator or system error. Such a condition usually requires that the step be abnormally terminated.

The general format of error messages in the JOR is :

< message code> <message number>. <message text>

For example, DV03.MEDIA volid IS NOT ASSIGNED.

JOB INITIATION AND TERMINATION MESSAGES

The messages in this group are produced during job initiation and termination procedures (note that m.mmm signifies minutes to three decimal places).

JOBID = job identification

USER = username

PROJECT = project name

BILLING = account name

RON = run occurrence number

MOT = message of to-day

JOB EXECUTION LISTING mmm dd,yyyy

START hh:mm:ss

LINES = number of lines registered in the sysouts

STOP hh:mm:ss

CARDS = number of punched cards registered in the sysouts

CPU m.mmm Total for all steps in the job excluding centralized system functions

ELAPSE m.mmm

hh:mm:ss RESULT: job termination code (e.g. COMPLETED/ABORTED/KILLED)

STEP INITIATION MESSAGES

Under normal processing the following messages appear in the JOR :

STEP step-number [utility-name] [{REPEATED / RESTARTED AT CHECKPOINT}]

LOAD MODULE = load-module-name(yy/mm/dd hh:mm) [PREINITIALIZED]

LIBRARY = library-name [(MD = media-name)]

hh:mm:ss STEP STARTED XPRTY = nn

The step number is allocated by the system and represents the order in which each step was executed within the job.

The date and time given with the Load Module name identifies the time of the last linkage of the load module.

XPRTY gives the execution priority of the step at start time.

The following message appears when the operator interrupts the step with an HJ, TJ or END command :

STEP INITIATION {KILLED [SEV5] / DELAYED BY SHUTDOWN}

SEV5 occurs when the interrupt was caused by a TJ without the STRONG option.

The following message appears after a specific step initiation error message (see the Error Messages manual) :

STEP INITIATION ABORTED {SEV3 / SEV4} RC = hhhhhhhh → siu, ic, mnemonic code

SEV3 indicates that the step initiation failed because of a user error, whilst SEV4 means that the failure was due to an irrecoverable system or I/O error. The meaning of the component parts of the return code are explained in the Error Messages and Return Codes manual.

STEP TERMINATION MESSAGES

The following messages are produced at the end of a step and are split into those at task level those at step level and those which refer to checkpoints.

Task Level

	COMPLETED RC = hhhhhhhh → siu, ic,mnemonic-code
TASK task-name	ABORTED BY SYSTEM RC = hhhhhhhh → siu, ic,mnemonic-code
J = jj P = pp	ABORTED BY USER . TERMINATION CODE = status value
	KILLED

[RC TRACE : hhhhhhhh → siu, ic,mnemonic-code AT kkkkkkkk . . .]

The terms employed have the following meanings :

COMPLETED	The task has reached a normal (logical) end but may nevertheless produce an abnormal return code.
ABORTED BY SYSTEM	The task has been aborted by the system due to a severe error (program exception) which is described in a previous message (EXOR).
ABORTED BY USER	The task has been aborted by the program itself through a call to the $H.ABTSK primitive. The termination code gives the status value in decimal.
RC TRACE	A trace of the last four abnormal return codes set by functions called by the program. This information is useful to the field engineer in case of a system malfunction.

Step Level

```
TEMPORARY FILES USED = nn TRACKS
ifn ON media-name : NB OF IO CONNECTS = nn [NB OF LOG EVENTS = nn]
CPU m.mmm          PROG MISSING SEGTS nn          STACKOV nn
ELAPSED m.mmm      SYS MISSING SEGTS nn
LINES nn  LIMIT  NOLIM   BACKING STORE nn
                 nn
CARDS nn LIMIT   NOLIM   BUFFER SIZE nnn          CPSIZE nn
                 nn
```

The terms employed have the following meaning :

NB OF CONNECTS	The total number of physical I/Os issued by the user program.
NB OF LOG EVENTS	The number of I/O events that have been logged onto the SYS.LOG file. Such events occur in the case of I/O retries. They have nothing to do with success or failure of the I/O.
CPU	The CPU time taken to process the user program. It starts timing at the end of the step initiation (start program) and finishes at the end of step termination and includes system functions called by the program.
ELAPSED	The elapsed time between the end of step initiation and the end of step termination.
CPSIZE	The total size of the channel program pages in bytes.
SYS MISSING SEGTS	The number of missing segments related to system procedures called by the program.
PROG MISSING SEGTS	The number of missing segments related to the program itself.
BUFFER SIZE	Total buffer size in bytes.

BACKING STORE	The total backing store space used by the program in bytes.
LINES	The number of lines in the sysout files relating to this step.
CARDS	The number of card images in the sysout files relating to this step.
LIMIT	The physical maximum number of LINES or CARDS.

Checkpoint Messages :

CHECKPOINT nn TIMES CALLED
CHECKPOINT LARGEST SNAPSHOT LENGTH 111

A checkpoint snapshot is an image of the virtual memory address space saved by the checkpoint facility. The size of the largest snapshot is given in bytes.

JOB EXECUTION TRACE

Events which have modified the sequential execution of the JCL program are logged in the JOR. In particular each time a JUMP statement is executed with a true condition it is logged with the following message.

JUMP DONE TO label

Also all actions taken by the operator which affect the job are logged as follows :

COMMAND	JOR MESSAGE
FJ	hh:mm:ss JOB FORCED BY OPERATOR
RJ	hh:mm:ss JOB RELEASED BY OPERATOR
HJ	hh:mm:ss JOB HELD BY OPERATOR
MJ	hh:mm:ss JOB MODIFIED NY OPERATOR : CLASS = { class / * } SCH = { n / * } DPR = { n / * } SW = { * / n ... n }

Where * means not modified by the command.

MESSAGES AT RESTART TIME

If a crash occurs during the initiation or termination of a job or a step (process group) one of the following messages will be produced.

JOBINIT RESTARTED AFTER A SYSTEM CRASH
JOBTERM RESTARTED AFTER A SYSTEM CRASH
PGINIT RESTARTED AFTER A SYSTEM CRASH
PGTERM RESTARTED AFTER A SYSTEM CRASH

If a crash occurs between steps, for example during the processing of a $LABEL, $JUMP or $WRITER JCL statement, warm restart performs the necessary operations

to allow the interstep statement to be executed. In this case the following message is written on the JOR :

JOB RESTARTED AFTER A SYSTEM CRASH

If a warm restart aborts the job which was being executed or was suspended at the time of the crash because of irrecoverable inconsistencies found in its structure, the following message is written on the JOR :

JOB TERMINATED BY SYSTEM CRASH

If a system crash occurs while a step is being processed, the step is aborted and the following message appears in the JOR :

STEP ABORTED BY SYSTEM CRASH

This message is followed by recovery information about the files that were currently assigned to the step :

LIST OF FILES ASSIGNED AT CRASH TIME

EFN	VSN	PMD	SALVAGED	NEEDED
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Where :

EFN — Heads the list of file names.

VSN — Identifies the volume that contains the file in question.

PMD — Processing mode.

SALVAGED — Will indicate either YES or NO depending on whether or not the file was salvaged.

NEEDED — Specifies what is required for the recovery with the following significance :

NONE	No action required
FILREST	A file restore should be performed
VOLREST	A volume restore should be performed
VOLCHECK	Volume checking is required
DEALLOC	De-allocate the file
REALLOC	Re-allocate the file
VOLPREP	A volume preparation should be performed
UNKNOWN	Some damage has been done but the system is unable to establish the type of recovery action necessary.

Appendix A The mini-editor

INTRODUCTION

Parameter substitution allows the user to update JCL statements within an input enclosure. However substitution can only take place at predefined points within the statements, (i.e. wherever the value reference & nn or & keyord appears).

Parameter substitution, allows the user to change the JCL using the VALUES command in $VALUES, $INVOKE or $RUN without a thorough knowledge of the JCL to be changed. This is useful where a sequence of JCL is to be used in different circumstances with minor modifications ; for example where the MEDIA and the DEVCLASS change from one job to another it may be useful to use parameter substitution in the $ASSIGN statement. A full description of parameter substitution can be found in Appendix E of the JCL Reference Manual.

It is sometimes useful to be able to update a sequence of JCL statements on a «one-off» basis, (e.g. where the stored JCL sequence has to be modified for a single test run, or where an unexpected situation has to be tested), where no provision for change has been made in advance.

The Mini-Editor on the other hand gives the user a limited editing capability for stored JCL or for JCL in an input enclosure (where TYPE DATASSF appears in the $INPUT statement). The updating of a JCL sequence is done at translation time (immediately before parameter substitution is applied). The editing commands available to the Mini-Editor user is the following subset of the $LIBMAINT editing commands :

A - Creates a new line after the line number specified.

C - Changes the whole of the specified lines for the text following the C.

D - Deletes the lines specified.

I - Inserts new lines of text before the line number specified.

S - Substitutes one character string for another on the specified line.

Use of the Mini-Editor requires a knowledge of the JCL to be changed and the line number of each statement. This information can be obtained using the PRINT command of $LIBMAINT.

LINE NUMBERS

Each line in the stored JCL or input enclosure has an associated line number. The first line of an input enclosure is line number 10, the second is 20 and so on in steps of 10, e.g.

```
 10  VALUES WORK;
 20  STEP  LM1,LM.LIB;
 30  ASSIGN  IN,  *INPUT;
 40  ASSIGN  OUT,  WORKFILE,  FILESTAT = TEMPRY,
 50  END = PASS,  DEVCLASS = MT/T9,  MEDIA = &1;
 60  ENDSTEP;
 70  STEP  LM2,  LM.LIB;
 80  ASSIGN  IN,  WORKFILE,  FILESTAT = TEMPRY;
 90  ASSIGN  PRINT,  SYS.OUT;
100  ENDSTEP;
```

Each edit command must be preceded by either a line number on which the editing is to take place, or a pair of line numbers separated by a comma. In the latter case the editing command will operate on all the lines between the two specified including those specified. e.g.

```
50C
    ....
¢F
```

This will change the whole of line 50 to whatever follows the C, and :

```
30, 60S ....
```

will make whatever substitution is specified after the S on lines 30 to 60 inclusive.

The maximum line number is 999990.

THE MINI-EDITOR COMMANDS

There follows a description of each Mini-Editor commands in alphabetic order.

The Append Command

A -to append one or more lines after the given line number.

STATEMENT FORMAT

```
adA
line or lines to by appended after the line ad.
¢F
```

Example 2 :

```
30A
ASSIGN FRED, EXTFIL;
¢F
```

This will have the effect of inserting the text ASSIGN FRED, EXTFIL; between line 30 and the next line, leaving the original text unchanged. In Example 2 the result would be :

Example 3 :

```
30   ASSIGN IN, *INPUT;
     ASSIGN FRED, EXTFIL;
40   ASSIGN OUT, WORKFILE, FILESTAT = TEMPRY, . . .
```

The append command (A) must be written in exactly the same way as it is presented above. That is, with the line number and the A command on the first line, the text to be appended in the next lines and on the last line an end of text marker, ¢F. The end of text marker is necessary to prevent the Append command micking up the next edit command and appending it to the rest of the text. The user may append several lines in the same append command.

The Change Command

C -to change one or more lines completely.

STATEMENT FORMAT

ad1[,ad2]C

replacement line

¢F

Example 4 :

30C

ASSIGN BILL, *DATA;

¢F

Example 5 :

150,210C

MEDIA = &1

¢F

The result of the edit command in example 4 will be to delete the existing line 30 and to replace it with ASSIGN WILL, *DATA;. The result of the edit command given in example 5 will be to delete lines 150 to 210 inclusive and replace them all with MEDIA = &1.

The Change command must be written exactly as it is shown above occupying one line for the line numbers and the change command C, one line for each line to be replaced and one line for the end of text command ¢F. The end of text command is necessary since the user may replace one line with many and the editor needs to be informed when the end of the replacement text has been reached.

The Delete Command

D -to delete one or more lines.

STATEMENT FORMAT

ad1[,ad2]D

The Insert Command

I -to insert one or more lines immediately before the specified line or lines.

STATEMENT FORMAT

ad1 I

lines to be inserted

¢F

Example 6 :

```
30I
ASSIGN FRED, EXTFIL;
¢F
```

Example 7 :

```
170,
DEVCLASS = MT/T7,
MEDIA = &2;
¢F
```

The result of the edit command given in example 6 will be to put the line ASSIGN FRED, EXTFIL; immediatly before line 30, leaving all other lines unchanged. The result of the edit command given in example 7 will be to put the two lines :

```
DEVCLASS = MT/T7,
MEDIA = &2;
```

before the line 170. Apart from this insertion the rest of the text will remain unchanged. The end of text marker ¢F is mandatory.

The Substitute Command

S -to substitute a character string for a new one.

STATEMENT FORMAT

ad1[,ad2] S/string 1/string 2/

The string delimiting character «/» may be replaced by any other character so long as it is not contained in either string 1 or string 2. It is necessary to change the string delimiter when either string 1 or string 2 contain a «/».

The $LIBMAINT editor makes use of the following special characters :

. (dot)

* (asterisk)

^ (not)

$ (dollar)

& (ampersand)

Although these characters are not used by the Mini-Editor we recommend that when used they be protected by preceding them with the character protection symbol ¢C. If this has been done the Mini-Editor commands will be available for use on the $LIBMAINT editor. In string 1 only the characters . (dot), *(asterisk), ^(not) and $ (dollar) need be protected. In string 2 only the character & (ampersand) needs to be protected.

Example 8 :

```
30S/N IN/N FRED/
```

This will have the effect of replacing the character string IN with the character string FRED. If this edit command were applied to example 1 the internal file name on line 30 would be FRED in place of IN. Note that it is necessary to make the string 1 characters explicit within the line, hence in example 8 IN and FRED are preceded by an N plus a blank space. Had this N blank been omitted both the IN character strings would have been replaced by FRED, giving :

Example 9 :

```
30 ASSIGN FRED, *FREDPUT;
130,250S(MT/T9(MS/M452))
```

The result of the edit command in example 9 will be to change all occurrences of MT/T9 between lines 130 and 250 inclusive. Note that because string 1 and string 2 contain a / this cannot be used as a string delimiter hence the character (has been used.

THE JCL FOR EDITING

To execute the edit commands the user must build them into an input enclosure and then reference the input enclosure via UPDATE parameter of a $INVOKE statement.

Example 10 :

Suppose we wish to change the JCL in an input enclosure called JIM with the following editing commands which have been written into an input enclosure called BILL :

```
$INPUT BILL;
50S/X384/Y9872/
110A
  ASSIGN IN, WORKFILE, TEMPRY,
  END = PASS, DEVCLASS = MT/T7, MEDIA = &4;
¢F
$ENDINPUT;
```

The $INVOKE statement required to effect these changes is as follows :

```
INVOKE *JIM, UPDATE = *BILL;
```

If the JCL to be changed was stored in a source library file called ROD, the $INVOKE statement to execute the editing commands in example 10 would be as follows :

```
INVOKE ROD, AD.LIB, UPDATE = BILL;
```

Notes On The Edit Commands

All edit commands in the input enclosure must be in increasing order of line. Where a command uses two line numbers it is the first line number which counts for this purpose.

All commands must start in the first character position of the line (or the first column in the case of card input).

All blank spaces are considered as meaningful in all commands, character strings, and text to be inserted or appended. This means that the Mini-Editor will not recognize, for example, the command 80 C ; it must be written as 80C.

The same line number may be used in various edit commands. It so, the commands must appear in a specific sequence and only certain combinations are permitted within a record.

The possible combinations are :

[I] [S] A

[I] [S] D

[I] [S] C

[I] [S]

[I]

The input enclosure which contains the editing commands must be in DATA or DATASSF.

The $INPUT statement at the beginning of the input enclosure containing the edit commands must not contain a parameter for parameter substitution (i.e. an &n or a keyword).

There are no restrictions when embedded invoked JCL sequences require Mini-Editor updating. This means, for example, that an invoked JCL sequence which requires editing can contain a $INVOKE statement which itself requires editing.

If more than one $INVOKE statement updates a particular input enclosure, each $INVOKE statement (i.e. each application of edit commands), will create a different version of the input enclosure.

THE UPDATE SEQUENCE

The syntax of the edit commands is checked according to the above rules and any errors will cause the $INVOKE statement to be ignored. Error messages concerning the Mini-Editor commands will appear on the JOR. For a full description of these, see the Error Messages and Return Codes manual.

If the edit commands pass one syntax test, the editing is performed, and then any parameter substitution is done after the editing.

The JCL translator handles the input records after editing as if the updated records were originally the stored sequence (or input enclosure). Therefore, if the LIST = ALL command is given in $JOB the edited JCL will be printed on the JOR (but without the substituted values in the case of parameter substitution).

An error condition arises if a command is not used.

Examples

Example 11 :

The JCL to be changed is stored in a source library member called SEQ1 of the JCL library VAST.LIB. The contents of SEQ1 are :

```
10  VALUES WORK;
```

```
 20  STEP LM1, LM.LIB;
 30  ASSIGN IN, *INPUT;
 40  ASSIGN OUT, WORKFILE, TEMPRY,
 50  END = PASS, DEVCLASS = MT/T9, MEDIA = &1;
 60  ENDSTEP;
 70  STEP LM2, LM.LIB;
 80  ASSIGN IN, WORKFILE, TEMPRY;
 90  SYSOUT PRINT;
100  ENDSTEP;
```

The commands to edit SEQ1 are contained in the following input enclosure called UP :

```
$INPUT UP;
50S/T9/T7/
90C
ASSIGN PRINT, PRINFILE, DEVCLASS = MT/T9, MEDIA = PR5;
¢F
$ENDINPUT;
```

The $INVOKE statement required to execute the edit commands in the input enclosure UP, and to carry out the parameter substitution is as follows :

```
INVOKE SEQ1, VAST.LIB, VALUES = X1234, UPDATE = *UP;
```

After editing and parameter replacement we get the following JCL.

```
STEP LM1, LM.LIB;
ASSIGN IN, *INPUT;
ASSIGN OUT, WORKFILE, FILESTAT = TEMPRY,
END = PASS, DEVCLASS = MT/T7, MEDIA = X1234;
ENDSTEP;
STEP LM2, LM.LIB;
ASSIGN IN, WORKFILE, TEMPRY;
ASSIGN PRINT, PRINFILE, DEVCLASS = MT/T9, MEDIA = PR5:
ENDSTEP;
```

If LIST = ALL was specified in the $JOB statement the JCL printed on the JOR will be :

```
  VALUES WORK;
  STEP LM1, LM.LIB;
  ASSIGN IN, *INPUT;
  ASSIGN OUT, WORKFILE, TEMPRY,
• END = PASS, DEVCLASS = MT/T7, MEDIA = &1;
  ENDSTEP;
  STEP LM2, LM.LIB;
```

```
ASSIGN IN, WORKFILE, FILESTAT = TEMPRY;
• ASSIGN PRINT, PRINFILE, DEVCLASS = MT/T9, MEDIA = PR5;
ENDSTEP;
```

all modified lines are flagged using a dot.

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE: SERIES 60 (LEVEL 64) JOB CONTROL LANGUAGE USER GUIDE

ORDER NO. AQ11, REV. 1

DATED SEPTEMBER 1978

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here and furnish complete mailing address below. ☐

FROM: NAME _____ **DATE** _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

PLEASE FOLD AND TAPE –
NOTE: U. S. Postal Service will not deliver stapled forms

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 2025 Sheppard Avenue East, Willowdale, Ontario M2J 1W5
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

21792, 1.5978, Printed in U.S.A.

AQ11, Rev. 1