JCL

User's Guide

DPS7000/XTA
NOVASCALE 7000

Job Control and IOF

REFERENCE
47 A2 12UJ 03

DPS7000/XTA NOVASCALE 7000

JCL

User's Guide

Job Control and IOF

September 1999

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

REFERENCE
47 A2 12UJ 03

Printed in France

Suggestions and criticisms concerning the form, content, and presentatio1n of this book are invited. A form is provided at the end of this book for this purpose.

To order additional copies of this book or other Bull Technical Publications, you are invited to use the Ordering Form also provided at the end of this book.

Trademarks and Acknowledgements

We acknowledge the right of proprietors of trademarks mentioned in this book.

Intel® and Itanium® are registered trademarks of Intel Corporation.

Windows® and Microsoft® software are registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark in the United States of America and other countries licensed exclusively through the Open Group.

Linux® is a registered trademark of Linus Torvalds.

The information in this document is subject to change without notice. Bull will not be liable for errors contained herein, or for incidental or consequential damages in connection with the use of this material.

Preface

Scope and Objectives

This manual provides information on the use of JCL in the GCOS 7 operating system.

Intended Readers

This manual is intended for programmers and operators.

Structure

Section 1 introduces the basic concepts of a job and its components.

Section 2 describes the management of unit record input and output.

Section 3 discusses the assignment and allocation of files on various media.

Section 4 considers some general aspects of resource management.

Section 5 deals with parameter substitution and modification of stored JCL.

Section 6 explains how to change the order of processing of JCL statements.

Section 7 describes the Job Occurrence Report.

Appendix A provides some notes on managing resident volumes.

Appendix B explains how the user can define parameter values within certain JCL statements.

Bibliography

JCL Reference Manual	*47 A2 11UJ*

Pages iii and iv of the above manual provide a list of all manuals which may concern the JCL user.

Those directly referenced in this User's Guide are:

IOF Terminal User's Reference Manual (parts 1, 2 and 3)	*47 A2 38/39/40UJ*
System Operator's Guide	*47 A2 53US*
System Administrator's Manual	*47 A2 54US*
Library Maintenance Reference Manual	*47 A2 01UP*
Catalog Management User's Guide	*47 A2 35UF*
Administrating the Storage Manager Administrator's Guide	*47 A2 36UF*
Data Management Utilities User's Guide	*47 A2 26UF*
Cartridge Tape Library User's Guide	*47 A2 62UU*
Error Messages and Return Codes Directory	*47 A2 10UJ*

Syntax Notation

The following notation conventions for JCL statement formats are used in this manual:

UPPERCASE — The keyword item is coded exactly as shown.

Lower case — Indicates a user-supplied parameter value.

[item] — An item within square bracket is optional.

{item 1}
{item 2}
{item 3} — A column of items within braces means that one value must be selected if the associated parameter is specified. If the parameter is not specified the underlined item is taken as the default value.

() — Parentheses must be coded if they enclose more than one item.

... — An ellipsis indicates that the preceding item may be repeated one or more times.

Table of Contents

1. Job Management

2. Input/Output Management

6. Sequence Modification and Error Processing

7. Job Occurrence Report

A. RESIDENT Volumes

B. Parameter Substitution

Index

Table of Graphics

Figures

Tables

1. Job Management

1.1 Introduction

When you input work to the system, it is executed under control of GCOS (General Comprehensive Operating Supervisor). This work, which may consist of many separate jobs, is submitted to the system in the form of an input stream.

During a GCOS session, everything that is visible to the user is concerned with jobs. These jobs may have been delivered as part of the GCOS software, or they may be constructed by users. Each job has an identification, to which certain rights are allocated.

A job consists of one or more steps, which are executed consecutively.

Job Control Language (JCL) is used to control the flow of a job in the system. A job description consists of a sequence of JCL statements bounded by $JOB and $ENDJOB statements.

JCL statements may be basic statements or extended statements.

Basic statements are used to describe the details of steps constructed by the user.

Extended statements are used to describe, in a single statement, a utility step delivered with GCOS (for example, a compiler or data management utility).

This manual describes the facilities offered by Basic JCL statements. Extended JCL statements are described in other manuals. The *JCL Reference Manual* contains a list of Extended JCL statements (Section 5) together with the titles of their associated manuals.

1.1.1 Job Structure

Each user job in an input stream is delimited by JCL $JOB ....$ENDJOB statements.

The system resource requirements are defined by JCL statements and enclosed by STEP ...ENDSTEP statements.

Data in the input stream is delimited by $INPUT ....$ENDINPUT statements.

The structure of an input stream therefore consists of three levels of enclosure (see Figure 1.1). Their functions are to:

- Make the job known to the system; JOB enclosure.
- Describe the handling of each step to the operating system; STEP enclosure.
- Define data input in a job stream; INPUT enclosure.

```
                                  }
$JOB                              }
 -                                }
 -                                }
STEP         }                    }
 -           } Step Enclosure     }
ENDSTEP      }                    }
                                  }    Job Enclosure
$INPUT       }                    }
 -           }Input Enclosure     }
$ENDINPUT    }                    }
 -                                }
 -                                }
$SENDJOB                          }
                                  }
```

Figure 1-1. Job Description

1.1.2 Job Submission

Jobs can be submitted to the operating system in the following ways:

- From a job stream stored on disk or tape (RUN statement or the operator commands SJ/SI).
- From a remote station, under RBF (Remote Batch Facility).

NOTE:

IOF (Interactive Operation Facility) terminal users can enter certain commands directly to the system (that is, outside a job enclosure). In particular, an IOF terminal user may issue a RUN statement (JCL) or an EJR statement (GCL) to submit a stored job stream to the system. For further details, refer to the *IOF Terminal User's Manual, Part2.*

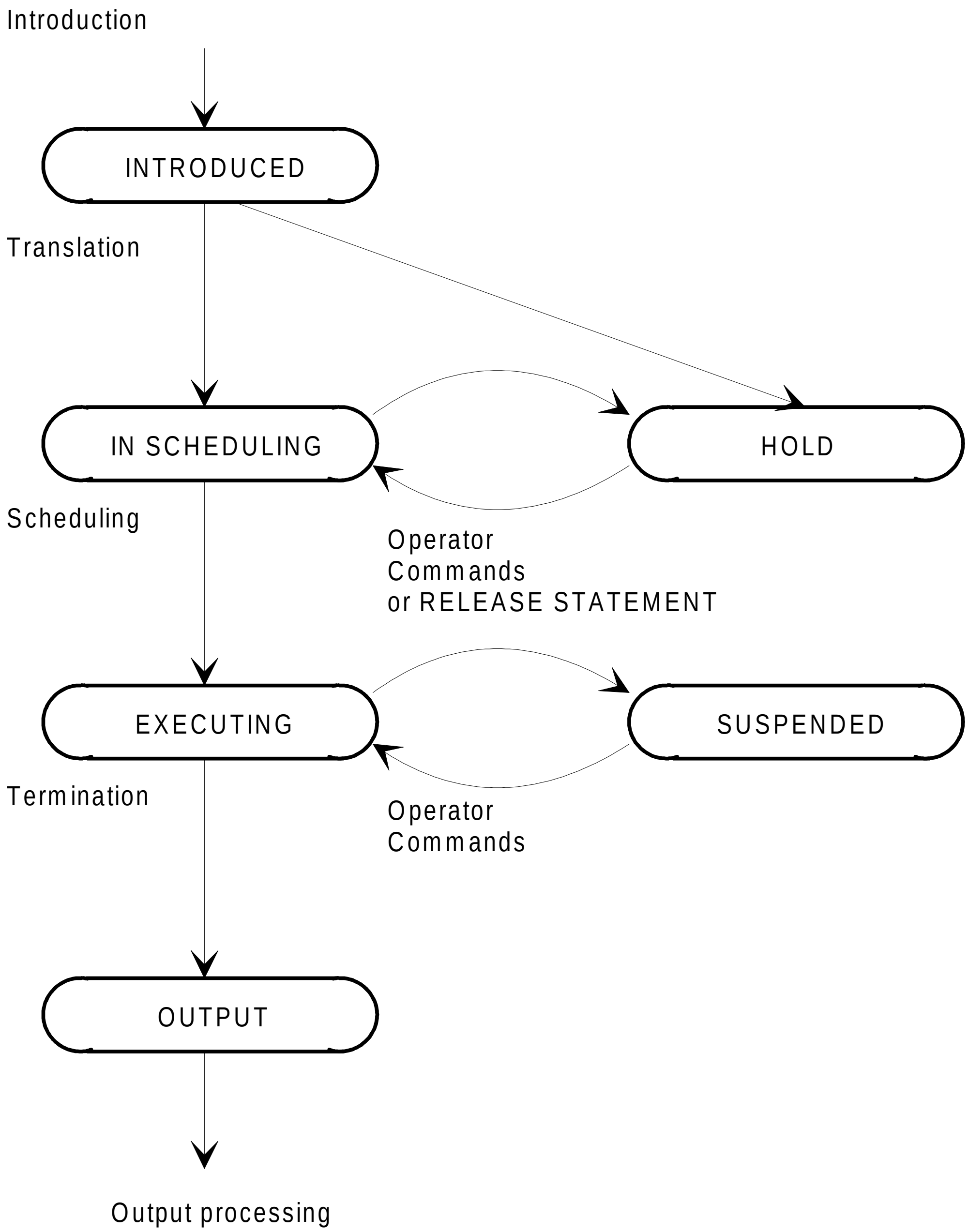


Figure 1-2. Stages of a Job Run

1.2 A Job Run

The whole life cycle of a job, as opposed to the period 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 (abbreviated to "ron"). The Run Occurrence Number is a number of up to four digits, which is always preceded by the letter X (for example X2384, X56, and X112). Steps of a job are referenced by their physical order in the job description, the step number.

1.2.1 Stages of a Job Run

From the time it is submitted until it is output a job is known to the system in different stages. These stages are represented in Figure 1.2. First the job is introduced to GCOS; this means that a request is sent to GCOS to execute the job. At this point 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. 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 which establishes the executing hierarchy for jobs currently "known" to the system. The Job Scheduler selects the next job that is to be executed from the IN SCHEDULING queue. 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 output onto the user-defined media.

Two other job states exist: the HOLD state and the SUSPENDED state. Jobs in the HOLD state 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)". 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.

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

The operator command "HJ ron ENDSTEP" can be used to makes the resources allocated to an executing job available to another job. The executing job is suspended at the end of the current step, and the resources are freed for use by the other job.

A job in the HOLD state can be put IN SCHEDULING by the operator command "RELEASE JOB (RJ)", or by a RELEASE statement in another job.

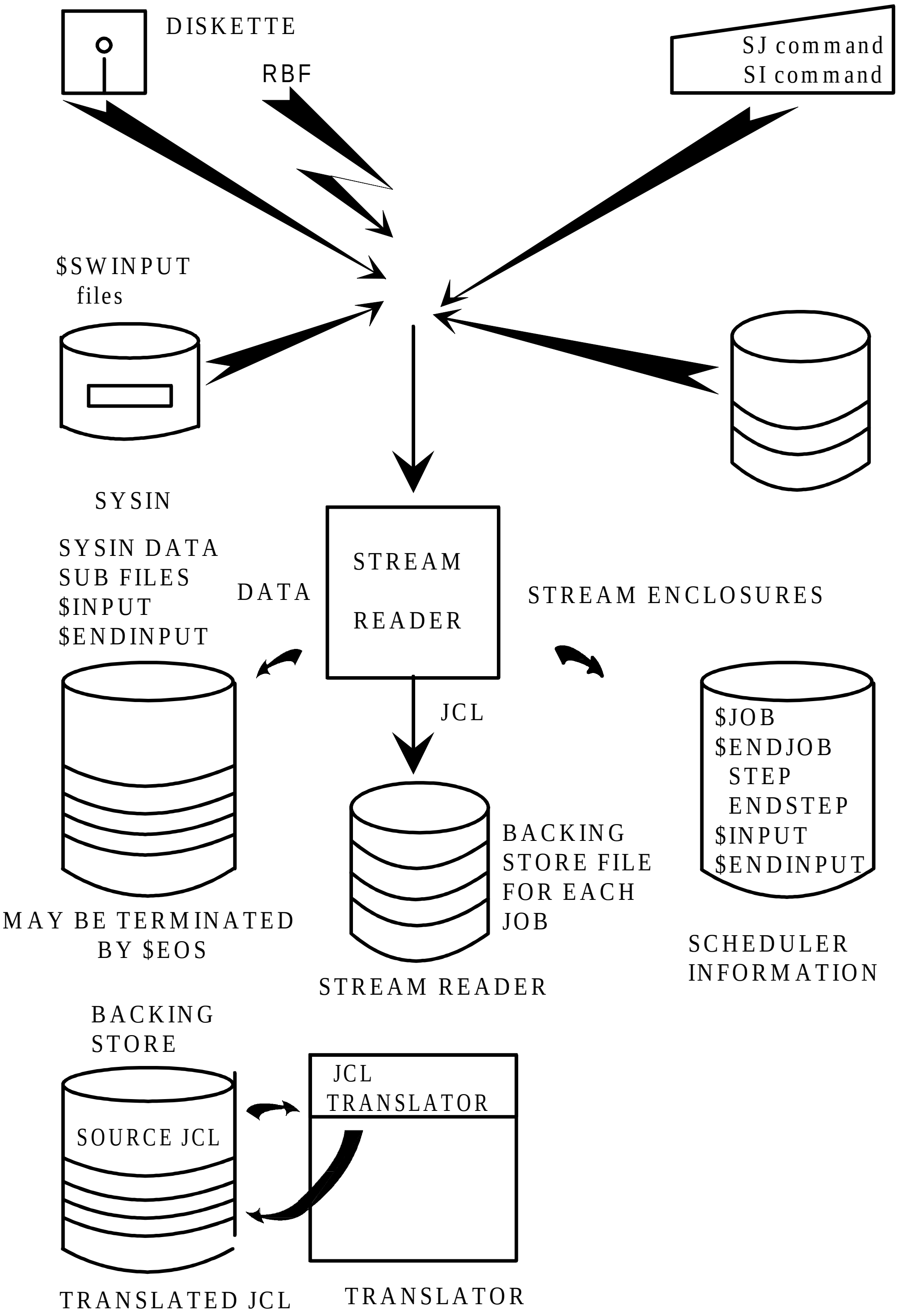


Figure 1-3. Stream Reader and JCL Translator

1.2.2 Input Reader

The Input Reader is initialized by an operator command, or by loading a specific diskette. The Input Reader controls the introduction of a job stream into the system by reading, analyzing, translating and storing data and JCL for later scheduling and execution. The input reader consists of two separate processes Stream Reader and JCL Translator.

1.2.2.1 STREAM Reader

When the Stream Reader is requested to read an input stream 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 the 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, creating one JCL file in backing store and one or more SYSIN subfiles for the 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 the end of input stream is signaled to the Translator and the translation is initiated.

1.2.2.2 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 list of all errors is sent for later printing in the job Occurrence Report.

1.2.3 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 that can be known to the system is set during system configuration time. If this limit is exceeded the Stream Reader stops further jobs from being introduced to the system.

1.2.4 Input Stream

The $SWINPUT statement can be used to switch the Stream Reader from the current stream to the file referenced by $SWINPUT. Logically, the result is equivalent to the replacement of the $SWINPUT statement by the contents of the file which it references. The file referenced must be available to the Stream Reader.

1.3 Job Descriptions

1.3.1 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;
```

If the following parameters are not specified in the $JOB statement, default values are taken from the user profile when the job is submitted:

&JOBID	Job identifier
&RON	Run Occurrence Number
&USER	User name
&PROJECT	Project identifier
&BILLING	Billing identifier
&H_DATE	Current date (in format YYMMDD)

If the site catalog is available and has been validated, it contains a list of users and associated project and billing information; and in this case only, the USER parameter is used.

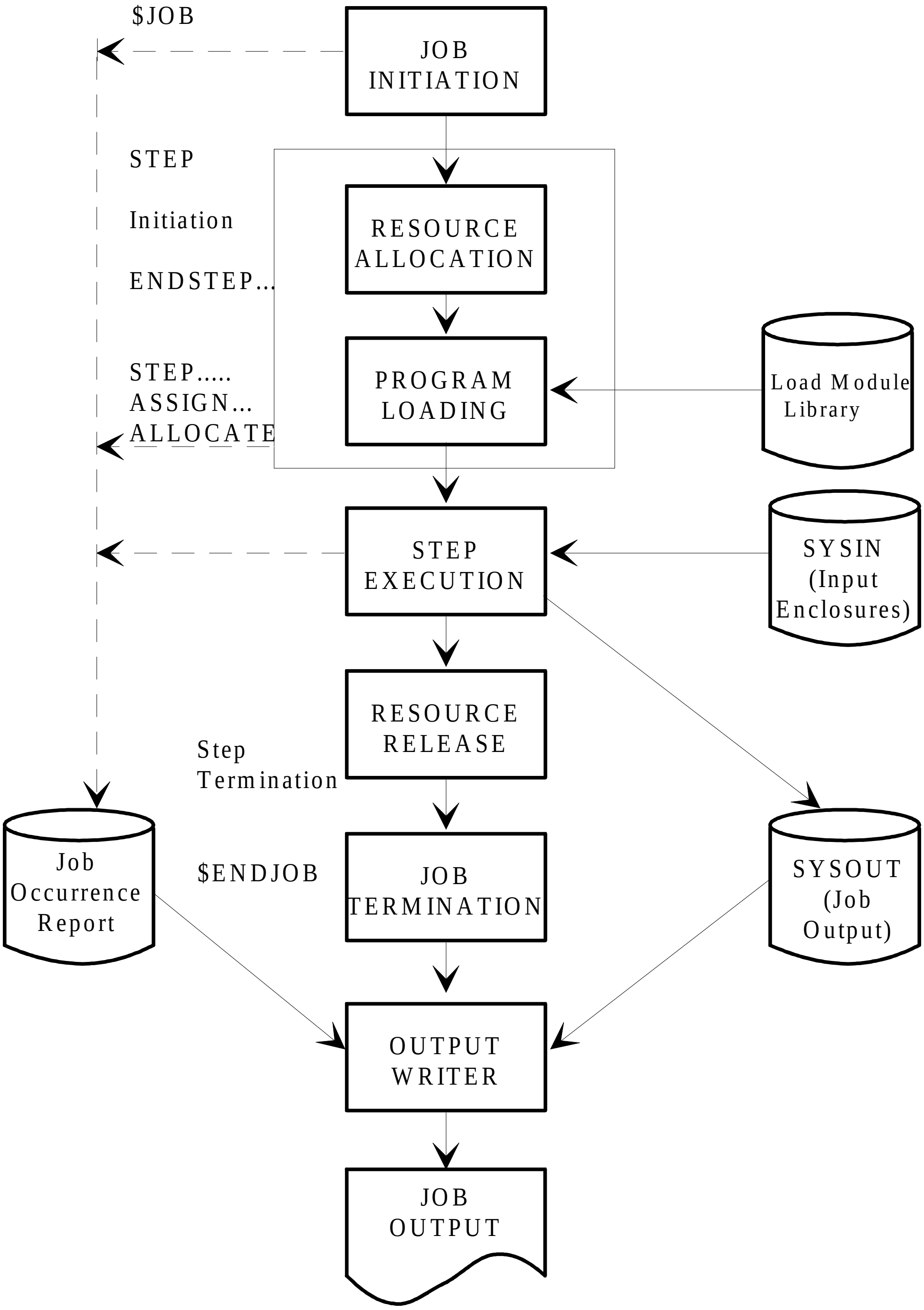


Figure 1-4. Step Execution for a Single Step Job

The job description is always terminated by $ENDJOB, which has no parameters.

1.3.2 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.

1.3.3 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-name (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 to a later step, for details, see section 3), 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;

For further information, refer to section 3, File Assignment.

1.4 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 enables you to plan for efficient processing of workload. The following paragraphs summarize the techniques available. For further details, see the *System Operator's Guide* and the *System Administrator's Manual*.

1.4.1 Scheduling Priority

A job is assigned a scheduling priority, which indicates its urgency relative to other jobs. Scheduling priorities range from 0 (highest) to 7 (lowest). 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, 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 (known as the job class multiprogramming limit). At this point, the remaining jobs are left in the IN SCHEDULING state until the termination of a job frees a multiprogramming slot.

You can specify the scheduling priority of a user job using the PRIORITY parameter in the $JOB or in the RUN statement. If the PRIORITY parameter is not present a default value is assigned to the job according to its job class. During system configuration, it is possible to specify the highest priority that a job class can have.

NOTE:

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

1.4.2 Job Classes

You can influence the order in which jobs are executed by assigning a job class in the $JOB statement. 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.

The class of a job may be restricted through the site catalog based on its project identification. A project default job class may be specified for a project, and this value is used for jobs which have no class or whose class violates the restriction specified in the site catalog. If no default class is given in the catalog, the default job class is P.

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, 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.

1.4.3 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 in the step enclosure and allocates the appropriate system resources (or the step is queued until all necessary resources are available). The load module is loaded from the load module library and step execution begins when the ENDSTEP statement for that step is encountered.

1.4.4 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 < = n < = 9 (0 = highest priority) and can be specified using the XPRTY parameter of the STEP statement, or the operator command MJ (Modify Job). If the XPRTY parameter is not specified, the default value is determined by the job class.

NOTE:

> The dispatching priority of a step can be modified by using the operator command MJ (Modify Job). The MDPR (Modify Dispatching Priority) command is used to control what specific CPU allocation is available to steps that are executing with a certain dispatching priority.

1.5 Holding And Releasing Jobs

You can influence the timing or order of job execution by holding a particular job (that is, preventing it from being executed) and then releasing it at an appropriate time. This is achieved using the HOLD parameter and the RELEASE statement, each of which is described below.

1.5.1 The HOLD Parameter

When a job is introduced with the HOLD parameter specified in the $JOB statement, it is not scheduled for execution until it is released either by an RJ operator command or a RELEASE statement in a job that is currently executing. There are two forms of the HOLD parameter and each is discussed separately below.

1.5.1.1 HOLD

If this form is used the job can be released by a single RELEASE statement or an RJ operator command. The format is:

```
$JOB MYJOB, USER = MYSELF, HOLD;
```

The job MYJOB will stay in the HOLD state until either:

- the operator issues the command RJ X123, where X123 is the ron of MYJOB, or
- another job executes a statement of the form RELEASE MYJOB.

1.5.1.2 HOLD = n

In this case the job is released by the execution of n RELEASE statements or an RJ operator command with the STRONG option. This facility enables the release of a job to be made dependent on the execution of several other jobs or job steps.

The format is,

```
$JOB MYJOB, USER = MYSELF, HOLD = 3;
```

Without operator intervention (via RJ) the job MYJOB will stay in the HOLD state until 3 RELEASE statements have been executed. These RELEASE statements will be of the form:

```
RELEASE MYJOB;
```

These RELEASE statements could be in separate jobs or different steps of the same job.

EXAMPLE:

```
    job A                   job B             job C
    .                       .                 .
    .                       .                 .
    .                       .                 .
RELEASE MYJOB;          RELEASE MYJOB;    RELEASE MYJOB;
    .                       .                 .
    .                       .                 .
    .                       .                 .
    .                       .                 .
```

❑

The job MYJOB will be released for execution after the three RELEASE statements (in jobs A, B, and C) have been executed. This ensures that MYJOB cannot start executing until after jobs A, B and C.

If the operator attempts to release the job MYJOB with the RJ command,

```
RJ X123
```

where X123 is the ron of MYJOB, then the message

```
RJ STRONG REQUIRED FOR X123: HOLD COUNT = 3
```

is given. This is to warn the operator that he is trying to interfere with automatic job synchronization. The job MYJOB is not released. The value of HOLD COUNT is the number of RELEASE statements still needed to release the job. This count is reduced by one by each RELEASE statement and the job is released when the count becomes zero.

The operator can release the job at any time (irrespective of the value of the hold count) by issuing an RJ command with the STRONG option as follows:

```
RJ X123 STRONG
```

It should be noted that HOLD = 1 is not exactly equivalent to HOLD. Even though a single RELEASE statement releases the job in both cases, the operator can release the HOLD job with an RJ command without STRONG, whereas RJ ....STRONG is needed to release a HOLD = 1 job.

1.5.2 RELEASE Statement

The RELEASE statement in a job enclosure can be used to release a job which has been suspended (by a JCL HOLD parameter in $JOB statement, or by the operator command HJ). The job in which the statement appears and the job to be released must have the same characteristics, namely USER and PROJECT. If the job being held has the parameter HOLD = n in its $JOB statement then n RELEASE statements must be executed before the job is released (that is, each RELEASE reduces n by one and the job is released when n becomes zero). 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 keyword is omitted from RELEASE then all 32 switches are set to zero.

1.5.3 Control of Interdependent Jobs

The testing of switch or status values can be used to control the order in which interdependent jobs are executed. The user can do this by interspersing the job 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 job descriptions.

1.6 Start Up Procedures

Startup procedures applicable in a batch environment are described below. There is a similar facility available in an interactive environment and this is described in the *I0F Terminal User's Reference Manual*. Startup procedures may be mandatory or optional. For more information, refer to the *System Administrator's Manual*.

1.6.1 Description

A startup sequence consists of JCL statements that are inserted automatically in every job without further user action. The sequence is stored in the source library named SITE.STARTUP and there is usually a member for the site (member-name = SITE_B) and members for selected projects (member-name = < project-name_B). The site sequence (if present) is inserted immediately after the $JOB statement and the project sequence (if present) then follows. Startup sequences can also be associated with particular users (member-name = project_user_B). The NSTARTUP parameter, described below, can be used to explicitly request the suppression of the optional startup sequence(s).

1.6.2 Application

Startup sequences can be used to achieve the following objectives:

- Specification of a common working environment at either the installation level (site startup) or the project level (project startup). For example, the VALUES statement can be used to specify parameter values, and the LIB statement can be used to specify libraries, thereby relieving user jobs of these tasks.
- Users of a given project can be presented immediately with a given processor (e.g., Library Maintenance, QUERY) without explicitly calling it. In such a case, the JCL necessary to call and execute the processor concerned would be stored in the project or user startup.
- Some special steps (for accounting or initialization purposes) can be executed automatically at the installation level (site startup) and/or the project level (project startup), and/or the user level (user startup).

1.6.3 Creation and Maintenance

The SITE.STARTUP library must be on a resident disk or cataloged. The members of the library are created and maintained using the standard library maintenance facilities of Library Maintenance. These are described in the *Library Maintenance Reference Manual*. In installations which have implemented Access Rights, the user must have at least the EXECUTE access right to the SITE.STARTUP library. In such installations, it is desirable to restrict the write access on this library to the SYSADMIN project. If Access Rights have not been implemented then, for security reasons, it is recommended that SITE.STARTUP be kept under the control of system operations.

1.6.4 NSTARTUP Parameter

Startup sequences (if they exist) are inserted automatically. Users can explicitly inhibit them by specifying the parameter NSTARTUP in the $JOB statement. If the job's project is SYSADMIN, then NSTARTUP inhibits the startup sequences. Note that this facility is necessary to enable an erroneous site startup sequence to be bypassed and corrected. If the project is not SYSADMIN, then NSTARTUP inhibits just the optional startup sequence, the site startup sequence is still inserted.

1.6.5 Example of a Batch Project Startup

The member MYPROJ_B of the SITE.STARTUP library contains the following:

```
ATTACH CATALOG1 = MYPROJ.CATALOG
       CATALOG2 = OUR.CATALOG
       CATALOG3 = SITE.CATALOG;

VALUES DISK1 = 'DEVCLASS = MS/D500, MEDIA = VOL3'
       DISK2 = 'DEVCLASS = MS/D500, MEDIA = VOL5';
```

The effect of these statements is to define a catalog search path (ATTACH) and define values for the JCL keyword parameters (see Section 4) &DISK1 and &DISK2. All jobs whose project is MYPROJ will have these statements automatically inserted in their job description unless the NSTARTUP parameter is used to request suppression of the startup.

Thus a job which starts,

```
$JOB MYJOB, USER = MYSELF, PROJECT = MYPROJ;

xxxx

xxxx

xxxx
```

is executed as if it were,

```
$JOB MYJOB, USER = MYSELF, PROJECT = MYPROJ;
ATTACH CATALOG1 = MYPROJ.CATALOG
       CATALOG2 = OUR.CATALOG
       CATALOG3 = SITE.CATALOG;

VALUES DISK1 = 'DEVCLASS = MS/D500, MEDIA = VOL3'
       DISK2 = 'DEVCLASS = MS/D500, MEDIA = VOL5';

xxxx
xxxx
xxxx
```

This relieves the user of the task of specifying ATTACH and VALUES in his job and ensures that all jobs in the project use the correct catalogs and disk volumes. By changing the project startup sequence, the project manager can implement the change for all jobs without further action.

A job which starts,

```
$JOB MYJOB, USER = MYSELF, PROJECT = MYPROJ, NSTARTUP;

xxxx

xxxx

xxxx
```

will not have the project startup sequence included, if it is optional.

2. Input/Output Management

2.1 Introduction

To separate the input/output function from that of job translation and execution, and thus prevent the build-up of queues, GCOS employs a spooling system. This facility makes use of intermediate storage space 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 access a device directly.

2.2 Handling Input Data

There are three ways in which input data 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 CREATE utility, or by storing it as a member subfile) of a source library by using the LIBMAINT utility. This facility is known as "permanent input file".
- by storing it in a library using the $DATA statement.

2.2.1 Input Enclosures - $INPUT and $ENDINPUT

For standard SYSIN, the input 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 of the $INPUT statement allow the user to select characters from the data, and to concatenate records together.

2.2.1.1 ENDCHAR

When the ENDCHAR parameter is used, consecutive input data are concatenated to the same record until the character specified in the ENDCHAR parameter appears as the last non-blank character of the input record.

FOR EXAMPLE:

```
ENDCHAR  = /

Input data:        : ABC/DE...            (80 characters)
                     FGHIJ/...            (80 characters)
resulting
record             : ABC/DE ...FGHIJ      (85 characters)
```

❑

2.2.1.2 CONTCHAR

When the character specified in the CONTCHAR parameter appears as the last non-blank character of the input record:

- only the characters preceding it in the current data are copied to the record,
- the next card will be concatenated to the current one in the same input record.

FOR EXAMPLE:

```
  CONTCHAR  =  +


  Input data    : ABC/DE + ...      (80 characters)
                  FGHIJ             (80 characters)

  resulting
  record        : ABC/DEFGHIJ ...   (86 characters)
```

❑

2.2.1.3 ENDCHAR and CONTCHAR used together

The ENDCHAR and CONTCHAR parameters can be used together.

FOR EXAMPLE:

```
ENDCHAR = /       CONTCHAR = +

Input data:    : ABCDE + ...      (80 characters)

                 FGHIJKL/ ...     (80 characters)

resulting
records        : ABCDEFGHIJKL     (12 characters)
```

❑

If the input data to be read does not have the $ character in column 1 of any of the records, the $ENDINPUT statement need not be written, and the END parameter of the $INPUT statement can be used.

FOR EXAMPLE:

```
$INPUT INDECK, END = DOLLAR;

input data

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

❑

If the input data to be read contains a $ENDINPUT statement, the match facility can be used.

FOR EXAMPLE:

```
$INPUT INDECK, END = MATCH;

$INPUT DATA            }
other data to be read  } DATA
$ENDINPUT DATA;        }
$ENDINPUT INDECK;      }
```

❑

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

NOTE:

If the ENDCHAR is the blank character, all the characters in the input data up to the last non-blank character are transferred to the 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 data must always be blank, otherwise this data will be concatenated to the next 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.

2.2.2 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 an ASSIGN statement of the form:

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

Other parameters of 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
```

2.2.3 Permanent Input File

With a permanent input file, the intermediate file that contains the input data may be:

- a permanent sequential disk or tape file (filled by using the CREATE utility, for example)
- a subfile of a source library, (filled by using the LIBMAINT utility, for example).

CREATE can be used in two ways:

1. **If the file does not exist, OUTDEF can be used to specify the characteristics required**

```
CREATE INFILE = *INDECK
       OUTFILE = (CARD, DEVCLASS = MT/T9, MEDIA = TAPE03),
       OUTDEF = (RECFORM = FB, RECSIZE = 80, BLKSIZE = 2400);

$INPUT INDECK;

       data to be read

$ENDINPUT;
```

If the program is to process card images then RECFORM = FB and RECSIZE = 80 is specified either in OUTDEF of CREATE or when the file is preallocated.

2. **If the file already exists then CREATE can be used to load the file and OUTDEF is not needed.**

```
 CREATE INFILE = *INDECK
        OUTFILE = (CARD, DEVCLASS = MT/T9, MEDIA = TAPE03);

$INPUT INDECK;

        data to be read

$ENDINPUT;
```

2.2.4 Using a Permanent Input File

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

AN EXAMPLE 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.
...
...
...
```

❑

NOTE:

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.

2.2.5 Input Data Types

The following data formats are discussed:

- Standard Access Record Format (SARF)

  In this format each record is composed exclusively of data without any special heading information. This is the format normally used in data files or subfiles that are passed between COBOL programs.

- System Standard Format (SSF)

  In this format each record comprises an eight-byte header followed by data. The function of this header is to make the file or subfile device-independent: a file or subfile in system standard format may be routed from the disk or tape to any kind of I/O device.

  With the permanent input 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 = TAPE03);

$INPUT INDECK, TYPE = DATASSF;

    input data

$ENDINPUT;
```

Note that the format of a permanent input file can be exactly the same as that of the standard SYSIN subfile.

❑

2.2.6 Reading SSF Input

The COBOL programmer can 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 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:

STEP PROGA …;

ASSIGN CARDFILE, *INDECK;
ENDSTEP;
$INPUT INDECK,
TYPE = DATASSF;

COBOL Program:

SELECT CARD
ASSIGN TO CARDFILE-SYSIN WITH SARF.
FD CARD
RECORD CONTAINS 88 CHARACTERS.
01 INREC.
   02 SSFHEAD PIC X(8).
   02 USERDATA PIC X(80).
```

❑

2.2.7 Data Enclosures-$DATA and $ENDDATA

A data enclosure is delimited by $DATA and $ENDDATA statements. The $DATA statement is processed by the Stream Reader and causes the loading of the data enclosure into a member of a source library. $DATA facilitates the loading of input data and provides an alternative to CREATE or LIBMAINT where the target file is a member of a permanent library. The file concerned must be available at stream reading time i.e., there can be no waiting for volume mounting or other resources). As a data enclosure is functionally equivalent to a job, $DATA cannot appear within a job enclosure. If there is a large amount of data to be loaded, it can be split into batches with each being loaded into a separate subfile of the library. Each subfile can be edited separately.

EXAMPLE 1:

```
$DATA SALES01, LIBA, USER = ABC, PROJECT = XYZ;

XXXXXX }
XXXXXX } data (batch 1)
XXXXXX }
XXXXXX }

$ENDDATA;
```

❑

This loads the first batch of sales data. Similarly, the other batches can be loaded into the subfiles SALES02, SALES03, etc. Each batch (subfile) can be edited separately using the LIBMAINT editor facility or a user program. After editing, the batches can be processed either individually or concatenated to form one large batch (e.g., using the star convention described in LIBMAINT).

EXAMPLE 2:

```
$DATA SALES*, LIBA, USER=ABC, PROJECT=XYZ, PRINT;

XXXXXX }
XXXXXX } data
XXXXXX }
XXXXXX }

$ENDDATA;
```

❑

Instead of numbering data batches as in Example 1, the automatic date-time stamping facility of $DATA can be used. This facility is employed by placing an asterisk after the subfile name (e.g., SALES*). On being processed by the Stream Reader, this asterisk is removed and replaced by the date and time (prefixed by D) at which the $DATA statement is executed. If execution takes place at 15 minutes 20 seconds after 10 a.m. on 13 November 1980, the data is loaded into the subfile named SALES D801113-101520 of the library LIBA. Because of the format of data-time stamp, subfiles named in this way will have a collating sequence (of names) corresponding to their chronological order of loading. Thus, data loaded at 25 minutes after 10 a.m. on 13 November 1980 will be stored in a subfile named SALES D801113-102500 that follows SALES D801113-101520 in collating sequence. This fact facilitates concatenation of the subfiles using the LIBMAINT star convention. The PRINT parameter specified on the $DATA statement causes a listing of the input data to be produced.

2.3 Handling of Printed Output

2.3.1 The GCOS Output Facilities

Many jobs will produce some output that is to be sent to a printer, although this may only be the Job Occurrence Report (JOR). A user program can access a printer directly; however, to avoid having to assign a printer 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 from that file by the system component known as the Output Writer.

The maximum number of reports known to the Output Writer for a job is 254. However, the reports from a job can be printed while the job is running, and when a report has been printed, its description is deleted, so there is often no limit to the number of reports that a job may output.

The theoretical number of outputs for a given job is 9999 JOBOUT members and 9998 non-JOBOUT members. Only 254 can be known to the Output Writer at a given time. The maximum number of subfiles "XRON_IIII_NNNN" of the standard SYSOUT for a job is 9999.

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

1. when a file that is to be output is being created (that is, written) by means of a file access method,
2. when the contents of the file are being printed by means of the Output Writer

A special access method, known as the SYSOUT Mechanism, is available for the writing of output files. 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. You can modify the standard output parameters (print belt, number of copies, etc.), by using an OUTVAL statement within the job enclosure or a SYSOUT statement in the corresponding step enclosure. Alternatively, you 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 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, you 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 user program.

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, 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 and may be in SSF or SARF format. Any editing requirements in these circumstances will be incorporated when the file contents are printed. However, you can choose to use the SYSOUT mechanism (see below) to edit the file when it is being created.

2.3.2 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 or BFAS (unedited permanent SYSOUT files),
3. direct use of the output device (no intermediate file).

Refer to *Figure 2.3* towards the end of this Section for an illustration of the functions of the various output facilities.

2.4 Sysout Mechanism

2.4.1 Description

As described previously, 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 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 from each record the trailing information 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".

2.4.2 Use

To use the SYSOUT Mechanism for the creation of a SYSOUT file or subfile, you 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.

2.4.3 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.

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; 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.

❑

2.4.4 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 in 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 that uses the SYSOUT facility:

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

2.4.5 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.

NOTE:

The DEFINE statement is normally used for the purpose of overriding, for a particular step execution, certain options that have been specified in a program. For the use of the SYSOUT Mechanism, you 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 your program, but you wish 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).

2.4.6 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 SYSOUT is above this minimum value. However, in the case of permanent SYSOUT files, you 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 recommended record format is VB.

EXAMPLE:

```
PREALLOC R2D2, DEVCLASS = MS/D500,
        GLOBAL = (MEDIA = VADAR, SIZE = 10), FILESTAT = UNCAT,
        BFAS = (SEQ = (BLKSIZE = 5000, RECSIZE = 1000,
        RECFORM =    VB));
```

❑

If an 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.

EXAMPLE (for disk):

```
STEP PROGA, ...;
     ASSIGN OUTFILE, R2D2, DEVCLASS = MS/D500, 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.
FD OUT
  RECORD CONTAINS 108 CHARACTERS
  LABEL RECORD IS STANDARD.
  ...
  OPEN OUTPUT OUT.
  ...
  WRITE record name.
  ...
  CLOSE OUT.
```

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.

2.5 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 printed (column headed Output).

NOTES:

1. 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).
2. A hyphen (-) in the column headed -SYSOUT indicates that either YES or NO can apply (the case where the column headed SYSOUT contains YES).
3. A hyphen in the column headed RECSIZE indicates that the entry is not applicable (the case where there is no permanent file).
4. 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.
5. 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

2.6 Avoiding the Use of the SYSOUT Mechanism for Output Editing

2.6.1 When Use of the Mechanism is Unsuitable

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 (for example, 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 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, or by specifying RECFORM not equal to VB. If in this case a SYSOUT statement is specified for the file, the SYSOUT Mechanism will take account of the enqueuing requested (that is, the value of the WHEN parameter), but the file will not be edited.

NOTE:
This method cannot be used for uncataloged tape files because 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. 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 in SSF, SARF, or ASA format (see the COBOL User's Guide).

- If the SYSOUT file is on tape and therefore cannot be preallocated, the SYSOUT mechanism can be avoided at installation level by specifying OWDFLT tape = NSYSOUT in the CONFIG statement.

2.6.2 Overriding Rules for the SYSOUT Mechanism

Figure 2-1 illustrates, in the form of a flow diagram, the overriding rules that decide whether a SYSOUT file will be edited or not.

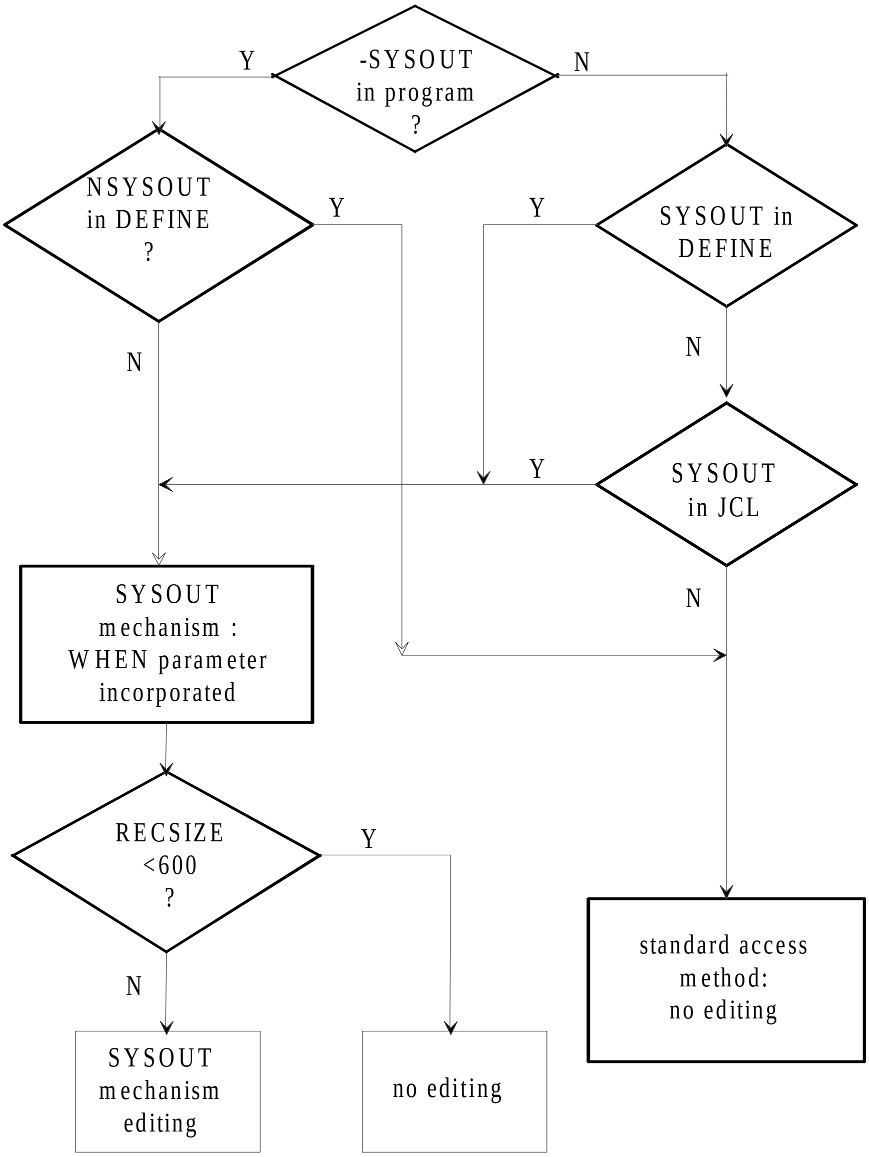


Figure 2-1. Overriding Rules for SYSOUT Mechanism

2.7 Standard Sysout Subfiles

2.7.1 Most Frequent Use

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 simple 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 an ASSIGN statement; in fact, as explained previously, an ASSIGN statement is only relevant for permanent SYSOUT files. 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 (for example, margin setting, form control), a DEFINE statement is necessary. Both types of parameters 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.

You should not open a standard SYSOUT subfile several times within a step. If you do, 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.

2.7.2 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, each copy having different characteristics.

2.8 Permanent Sysout Files

In some cases, a permanent copy of a report 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 a very large 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 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 copy of the file, a WRITER statement will be necessary; for example, for a printer listing:

```
WRITER OUT.PRFL1;
```

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.

In the following statement,

```
WRITER efn DEVCLASS = ...MEDIA = ...)
```

DEVCLASS and MEDIA are residency parameters for the file whose efn is specified.

However, in the form,

```
WRITER ...DEVCLASS = ...MEDIA = ...
```

DEVCLASS and MEDIA are editing parameters.

2.8.1 Writing to a Permanent SYSOUT File in Several Steps

You can write to a permanent SYSOUT file in several steps and request the Output Writer to print the complete file, as follows:

1. Open the file in OUTPUT processing mode in the first step that writes to it, and 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 (that is, 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 even if one of the job steps aborts, do one (or both) of the following:

- Make use of the JUMP statement (see *Section 6*).

- Put the WRITER statement immediately after the $JOB statement.

2.8.2 Partial Output of Files

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

EXAMPLE:

```
WRITER    (PFILE, DEVCLASS = MS/D500, MEDIA = CO18),
          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 you to output specified members of a permanent SYSOUT library file.

EXAMPLE 1:

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

❑

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

EXAMPLE 2:

```
WRITER MY.LIB, SUBFILES = (SUB*);
```

❑

Prints the members SUB1, SUBB, SUBC, SUBD etc. of the library file MY.LIB.

2.8.3 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, you must take care before attempting to deallocate a permanent SYSOUT file after an output request has been made for it. You are advised never to use the DEALLOC statement for a file in the same job as it is output, nor even in a later job unless it is certain the file has already been printed. Otherwise, there is a danger that the file will be deallocated before it has been output. In order to overcome the problem, you 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 permanent library file as the SYSOUT file and include the DELETE parameter in the SYSOUT or WRITER (or OUTVAL) statement.

2.9 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.

2.9.1 Output Editing

The parameters that are used to specify output 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)

2.9.2 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 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.

2.9.3 Lines Limits

To limit the amount of output produced by a program, for example to anticipate the occurrence of an infinite loop, you can specify in the STEP statement a maximum number of lines printed (LINES parameter). When the 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 you want to print the SYSOUT report for a particular production run, use a STEP statement of the form given below:

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

NOTE:

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. However, if several copies of a SYSOUT file are made (COPIES parameter), the additional lines produced by the extra copies are not included and neither are lines produced by a dump after an abort. The lines printed in the Job Occurrence Report are also independent of the LINES limit.

❑

2.9.4 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.

2.9.4.1 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.

2.9.4.2 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.
You can supply alternative editing parameter values in the WRITER statement, but these will be ignored unless the FPARAM (Force Editing Parameters) 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. To print the file later with standard characteristics, use the following WRITER statement:

```
WRITER (MYFILE, DEVCLASS = MS/D500, MEDIA = V1), FPARAM;
```

Similarly, if a standard form height and print density is required but you want a different margin setting:

```
WRITER (MYFILE, DEVCLASS = MS/D500, MEDIA = V1),
       FPARAM, PRINTER = (MARGIN = 10);
```

❑

2.9.4.3 Unedited Permanent SYSOUT Files

These are files that are not edited by the SYSOUT mechanism at file creation time, and about which the message: " 0U28 ifn IS NOT IN SYSOUT FILE FORMAT" has been issued in the JOR.
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, you must specify the format in the DATAFORM parameter.

2.9.4.4 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 in ASA format, you must specify it in the DATAFORM parameter. Other formats, i.e. SSF and DOF, are automatically recognized, or SARF is assumed.

2.9.5 Printer Characteristics

The user can specify the following printer characteristics:

- character set
- paper form
- logical page size
- stop levels within a page.

Details on these topics can be found in the *System Operator's Guide*.

2.10 Output Handling Parameters

2.10.1 Enqueing 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 the JCL statement SYSOUT only);

  ```
  WHEN = STEP
  ```

  For a job that contains several steps, this option allows the output of one step to be printed concurrently with the execution of later steps (depending on the current activity of the Output Writer, see note 4 below).

- early delivery, each time a (user specified) number of pages are ready

  ```
  WHEN = digits 5
  ```

  A separate SYSOUT file is created each time this number of pages has been output. After each such file is completed, it may be edited by the Output Writer depending on the output class and priority. For example, if WHEN = 100 each 100 pages of output constitutes a separate SYSOUT file. The Output Writer can process each file (i.e., 100 pages) as soon as it is complete. The early mechanism only applies to outputs created in the standard SYSOUT.

- as soon as the SYSOUT file is closed (i.e., "immediately");

  ```
  WHEN = IMMED
  ```

- no Output Writer request made at this point

  ```
  WHEN = DEFER
  ```

  for permanent SYSOUT files only, with the JCL statement SYSOUT.

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 to edit the file. This means that, if a SYSOUT statement is specified for a permanent file, the enqueuing request is observed even though the editing parameters are not. Editing parameters must be restated if a future output is required (see SYSOUT Mechanism, above).

2. The WHEN = DEFER option is useful when a program creates a permanent SYSOUT file and you wish to delay the output of the file content, but 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), you can add 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. 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, (that is, WHEN = JOB). Where the utility statement contains the PRTOUT parameter, you 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 may still be delayed beyond the termination of the step or job. Refer also to Deallocation of a Permanent SYSOUT File, above.
5. For repeatable jobs or steps, the enqueuing of output is delayed until the end of the job or the end of the step respectively (that is, the WHEN option is forced to JOB or STEP). This does not apply to dumps however.

2.10.2 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 of different categories 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. You 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. You 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).

You 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:

1. 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 by the Output Writer, but the request is made and an entry is put in the output queue.
2. If you specify 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;
```

❑

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

2.10.3 Production of Several Copies

With the COPIES parameter you can make several copies of a SYSOUT file.

EXAMPLE:

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

❑

2.10.4 Output Banners

You can suppress the standard output banners that appear on listings and, control how often banners are output, or supply alternative values for the items they contain (e.g., Run Occurrence Number, user-name). The appropriate parameters are NBANNER, BANLEVEL and BANINF respectively. Details of the format of printer listing banners are given in Section 8

Suppression of Skip Function

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

2.10.5 Use of the OUTVAL Statement

By means of the OUTVAL statement, you 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 affect at the time of the execution of the job and not at JCL translation time.

The scope of the parameter values defined by an OUTVAL statement is as follows:

- Only parameters, which do not depend on the destination station or the output class, remain applicable to the SYSOUT or WRITER request statements.
- If SYSOUT or WRITER specify an output, which has a destination other than that specified in the OUTVAL statement, the media, device class and priority are re-evaluated.
- In the same way, if a new class is specified, the media and priority are re-evaluated.

EXAMPLE:

```
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;
     .
     .
   INDSTEP;
   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.

2.11 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 an OUTVAL statement without any parameters is specified, the Job Occurrence Report will be printed according to the attached station values, the default values being as follows:

- DEST, depending on the attached station.
- default class, device class and media, depending on the attached station.
- priority, depending on the class. By default, the option WHEN = JOB, COPIES = 1 applies.

```
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 JOB is considered as a single file for output purposes and 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 JOB 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 = 5, COPIES = 1.

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

In the above example, the JOB is the only listing that has characteristics CLASS = D, PRIORITY = 4, WHEN = JOB, COPIES = 2 (assuming that there is no other OUTVAL statement in the job description, and that no SYSOUT nor WRITER statement specifies those four values).

❑

2.12 Example of the Use of Sysout and Writer in a Job

The following example contains a variety of output requests. An illustration of the different effects of these statements is shown in *Figure 2.2*.

```
$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;
      DEFINE OP1, SYSOUT;
     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 INCRDS;
    .
    .
    .
$ENDINPUT;
$ENDJOB;
```

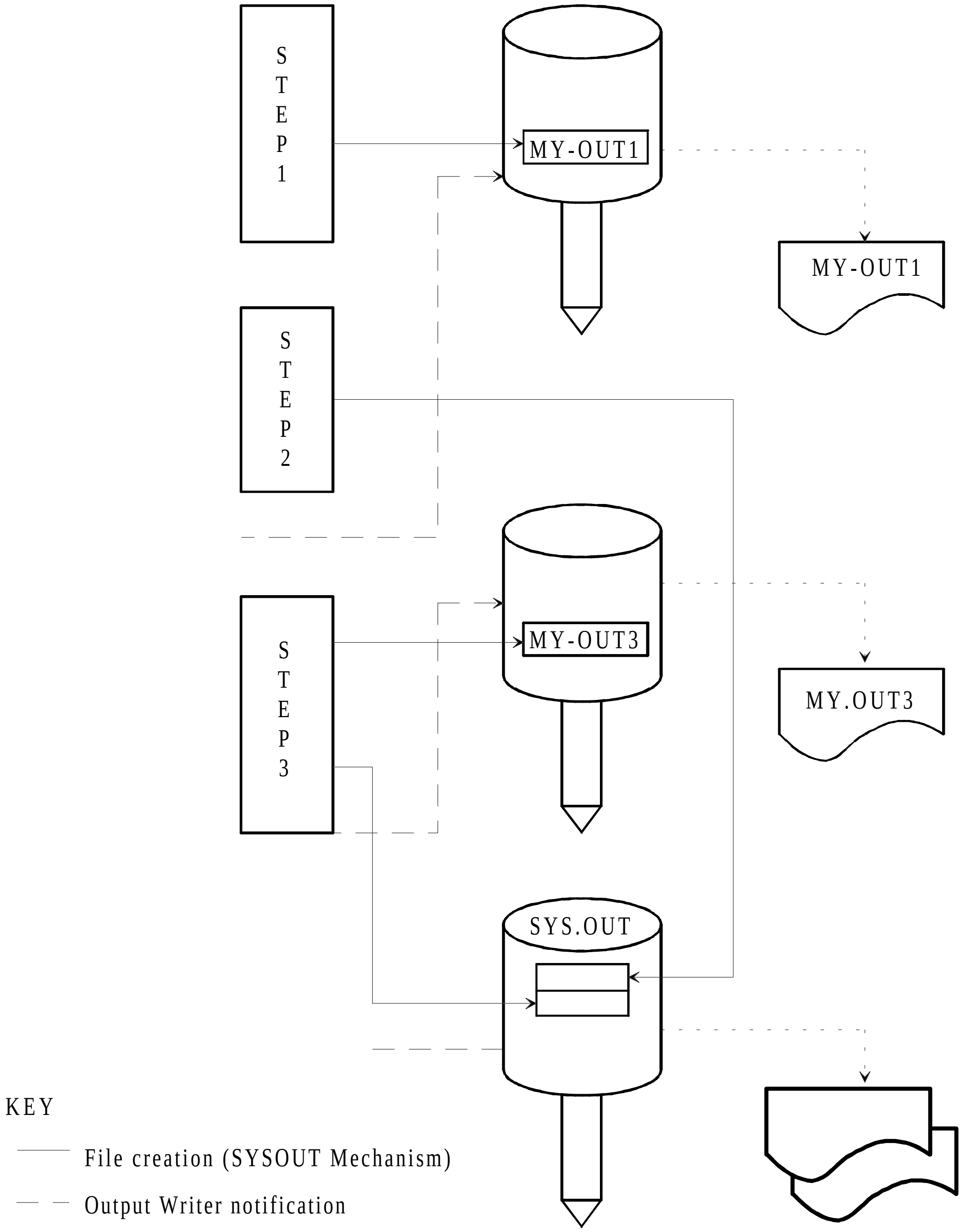


Figure 2-2. Job Output

2.12.1 Direct Use of the Printer

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. You have 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 directly you must specify the device in an ASSIGN statement. In addition, the COBOL program should contain the suffix -PRINTER in the ASSIGN clause of the SELECT statement. The ASSIGN statement can also specify the paper form for a printer (MEDIA parameter).

EXAMPLE:

```
1. $JOB DIRECT, ...;
   STEP STEPA, ...;
     ASSIGN PROUT, DEVCLASS = PR, MEDIA = 150000;
   ENDSTEP;

$ENDJOB;
```

❑

NOTE:

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.

2.12.2 Summary of Output Facilities

Figure 2-3 summarizes the action of the different output methods that are available.

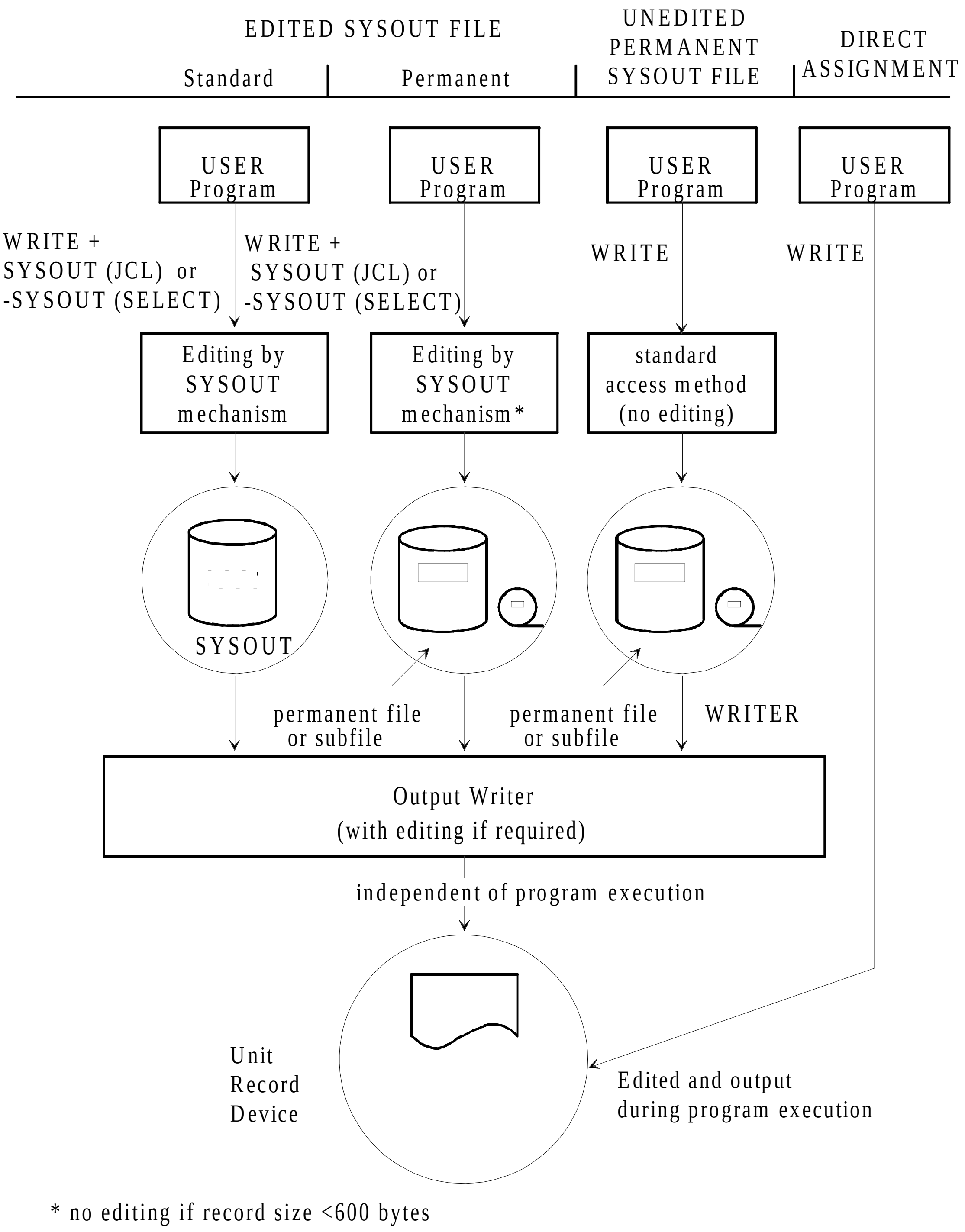

EDITED SYSOUT FILE
Standard
Permanent
UNEDITED PERMANENT SYSOUT FILE
DIRECT ASSIGNMENT
USER Program
USER Program
USER Program
USER Program
WRITE + SYSOUT (JCL) or -SYSOUT (SELECT)
WRITE + SYSOUT (JCL) or -SYSOUT (SELECT)
WRITE
WRITE
Editing by SYSOUT mechanism
Editing by SYSOUT mechanism*
standard access method (no editing)
SYSOUT
permanent file or subfile
permanent file or subfile
WRITER
Output Writer (with editing if required)
independent of program execution
Unit Record Device
Edited and output during program execution
* no editing if record size <600 bytes

Figure 2-3. Summary of Output Facilities

2.13 Summary of Output Writer Usage

2.13.1 PRTFILE, PRTDEF and PRTOUT

Extended JCL statements (e.g., COBOL, PREALLOC) implicitly define a step; that is, a step is created even though no STEP or ENDSTEP JCL statements appear (and consequently there is no step enclosure). Such extended JCL statements can be considered as SDS's (Step Defining Statements). In the case of an SDS, the absence of a step enclosure prohibits the use of JCL statements that would normally appear within the step enclosure e.g., ASSIGN), and the information concerned is usually supplied via a parameter or parameter group of the SDS.

In the case of output that will be processed by the Output Writer (sooner or later), the following parameter groups are of interest.

PRTFILE	description of a permanent SYSOUT file; i.e., parameters of the JCL statement ASSIGN.
PRTDEF	file and printer parameters; i.e., parameters of the JCL statement DEFINE.
PRTOUT	output parameters; i.e., parameters of the JCL statement SYSOUT.

2.13.2 OUTVAL, SYSOUT and WRITER JCL Statements

OUTVAL

Defines the default output parameters which will be applied to all output created by the job concerned. For example, if all the outputs of the job are to have the same output class, or to be printed on the same special paper, OUTVAL can be used to accomplish this.

SYSOUT

Defines the output parameters applicable to a single output (within a step). It takes effect at the moment of creation of the output in the user step. SYSOUT is associated with a given ifn. SYSOUT enables the user to specify the output class, priority, queuing, output device, paper, and causes editing of the output when possible. As a complement to SYSOUT, the JCL statement DEFINE can be used to specify editing parameters (e.g., form height, print density).

WRITER

Requests the printing of a file that has been created either in a preceding step or in another job. WRITER also permits the specification of output and editing parameters.

2.13.2.1 Difference Between SYSOUT and WRITER

SYSOUT permits the creation of output in edited format (depending on the value of RECSIZE) and requests the printing of the output. In contrast, WRITER requests the printing of a file already in existence. It is possible to create on output file in "non-edited" format and to supply the editing parameters later at the time of printing (via WRITER).

2.13.2.2 Difference Between SYSOUT and DEFINE

These two JCL statements do not define the same type of parameter; in fact their parameters are complementary. SYSOUT gives the output parameters, device class and media. DEFINE gives editing parameters (e.g., form length) which override those of media.

The printer parameters of DEFINE are rarely used in practice, as the media parameters of SYSOUT are sufficient. The characteristics of special paper (which could be supplied via DEFINE) are usually stored in the file SYS.URCINIT (This can be done using the URCINIT utility).

SYSOUT implies the use of the SYSOUT access (with or without editing) and causes the printing of this output even if SYSOUT has not been specified in the user program file description (e.g., COBOL SELECT ... -SYSOUT).

2.13.2.3 Summary Table

Table 2-2 shows the differences between standard SYSOUT and permanent SYSOUT.

Table 2-2. Differences Between Standard and Permanent SYSOUT

Standard SYSOUT	Permanent SYSOUT	
Conditions No ASSIGN statement or ASSIGN ifn, SYSOUT;	Conditions ASSIGN statement or PRTFILE parameter	
or SELECT ... -SYSOUT or SYSOUT ifn; or DEFINE ifn, SYSOUT	COBOL FD -SYSOUT or SYSOUT ifn ...; or DEFINE ifn, SYSOUT;	
Result Always edited. Always printed. Parameters from SYSOUT and/or DEFINE	Result Edited or not depending on RECSIZE. Printing automatic if SYSOUT (JCL) specified. Otherwise printing via WRITER JCL statement.	Result Not edited. Printing via WRITER JCL statement.

3. File Assignment and Allocation

3.1 Introduction

A number of system resources such as memory space, physical files and devices are associated with each job step. You can control the handling of files and devices by means of JCL statements.

This section explores the various means by which you can allocate space and assign files for user jobs. The catalog facility is also explained.

The allocation of file space is carried out by either the ALLOCATE statement or the data management utilities (PREALLOC, FILALLOC, LIBALLOC ...) and the allocation of device and volume is performed by the ASSIGN statement. The following paragraphs explain the concepts involved in resource allocation.

Files have characteristics which depend on their structure, and can be processed in different ways, depending on the application. Files are known internally to a program by an internal file name (ifn). Externally, i.e., to the GCOS system, files are known by an external file name (efn).

The association of a given efn to an ifn is made using the JCL statement ASSIGN. JCL is also used to specify other characteristics such as residency, sharing, access and journalization (if these are not contained in the catalog). The characteristics of a file are also contained in:

- The file label
- The VTOC
- The catalog

Three major classes of files are available to the GCOS user, namely:

- temporary files
- permanent cataloged files
- permanent uncataloged files

These three classes include the standard libraries and UFAS,), file organizations (e.g., sequential, indexed-sequential) and media types (disk, tape, cartridge). See *Table 3-1* below for restrictions.

Table 3-1. File Class, Organization and media

File Type		**UFAS S**	**UFAS IS**	**UFAS R**	**Libraries and LINKQD Files**
Temporary	T	X			
	D	X	*I	*I	X
Cataloged	T	X			
	D	X	X	X	X
Uncataloged	T	X			
	D	X	X	X	X
	Dt	X			

*I Not normally used.
S= SEQUENTIAL, IS=INDEXED SEQUENTIAL, R=RELATIVE,
T=TAPE, D=DISK, Dt=DISKETTE

You can use a temporary file when a work area is required in which the data is not needed after the end of the step. Such a file exists only for the duration of the step and is deleted at step termination. If you require such a file to be retained for more than one step within a job, but feel 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 organizations 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 you make no explicit space allocation, a default size is computed according to the volume organization and the file characteristics.

Permanent files are of two types, cataloged and uncataloged. 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 external-file-name (see ASSIGN).

Space for a permanent file can be made available using ALLOCATE (within a step) or in a job enclosure using the PREALLOC utility. For 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, a physical file must be accessed. The link between the file name known by the program, and the file name known by GCOS has to be established.

This link is provided using the ASSIGN statement in which the file name known to the program (internal-file-name, or ifn) and the file name known to GCOS (external-file-name, efn) are associated within a step. The ASSIGN statement allows any user program to see an input enclosure as an ordinary sequential file.

```
$JOB TEST, USER = TD, PROJECT = ED;
$INP UT CRDR;
     XXX
     XXX
     XXX
$ENDINPUT;
   STEP LIMI, LMLIB.EXS;
        ASSIGN CRDN1, INENC1, DEVCLASS = MS/D500, MEDIA = NI;
COMMENT ' IFN CRDNI IS ASSIGNED TO EFN INENC1, WHICH
        IS AN UNCATALOGED DISK FILE';
        ASSIGN DATA1, *CRDR;
COMMENT 'INPUT ENCLOSURE CRDR IS ASSIGNED TO DATA1';
        ASSIGN STAS, CDEX.DCOM, TEMPRY;
COMMENT ' TEMPORARY FILE CDEX.DCOM IS ASSIGNED TO STAS
        AND SINCE NO PASS PARAMETER IS STATED THE FILE IS
        DELETED AT THE END OF THE STEP';
        ASSIGN IN3, TD.COM;
COMMENT 'TD.COM IS CATALOGED IN THE SITE CATALOG';
   ENDSTEP;
   STEP LMODA, LMLIB.NEW;
        ASSIGN INA, MY.FILE;
   ENDSTEP;

$ENDJOB;
```

3.2 Catalog Overview

The fundamental function of the JCL statement ASSIGN is to associate an efn (external-file-name) with an ifn (internal-file-name). In addition, ASSIGN is used to specify a number of parameters associated with a file. These parameters can be divided into two classes, as follows:

Variable — Those which reflect the particular type of processing of the file in the step under consideration (and which vary from step to step); for example, END (= DEASSIGN, PASS or UNLOAD), ACCESS (= READ, WRITE etc.).

Fixed — Those which describe the characteristics of the file itself (and which do not vary from step to step) for example DEVCLASS, MEDIA.

In an installation it is desirable to have a control file which contains the fixed characteristics of each file in current use. This control file is structured such that the system can access this information given the efn of any file. This control file is known as a catalog, and the files whose characteristics are contained in the catalog are known as cataloged files.

3.2.1 Simplification of JCL

The use of cataloged files leads to considerable simplification of JCL, for example, the ASSIGN statement for a cataloged file can often be reduced to just

```
ASSIGN ifn, efn;
```

The user does not have to supply the values of the fixed characteristics of the file.

The existence of a catalog does not necessarily imply that all files are cataloged. For uncataloged files, the ASSIGN statement must supply the necessary parameters. Note that cataloged files may be contained in resident disks. In this case the parameter RESIDENT must not be specified in the ASSIGN statement, nor must DEVCLASS, MEDIA or DVIDLIST. (If any of these four parameters is specified and FILESTAT = CAT is also specified, the job is aborted at translation. If any of these four parameters is specified and FILESTAT = CAT is not specified, then the job aborts at execution with a return code of CATERR).

3.2.2 Generation Groups

In many applications, there are files which are updated at fixed intervals (e.g., daily, weekly) and are accessed only in READ mode between such updates. For security reasons, it is generally desirable to retain several generations of each such file. Each update uses the same JCL; that is, the input file (to be updated) and the output file (updated file) have the same file name. The output file of an update run becomes the input file to the next update.

This is achieved by using the catalog to define a generation group of files. The JCL statement SHIFT causes a cyclic rotation of generations. It is only necessary to use SHIFT before each update. The same JCL can be used for each update and the number of generations retained is constant. The most recently created generation is retained in place of the oldest.

3.2.3 Access to the System

In an IOF environment where there are many terminals available which permit interactive use of the system, it is essential to control access to the system and, given access, to control the activities of each user. Such control can best be exercised by the system itself.

To control access, the USER, PROJECT and BILLING parameters are used. In batch mode, these appear on the $JOB statement and in IOF they are given at log-on.

The USER is the person who is permitted to submit work via an IOF terminal or to submit a job in batch. A password associated with each user prevents unauthorized persons from logging on under IOF.

The PROJECT is a set of users to which a number of access rights are associated. These include the right to access a file or use a terminal.

The BILLING is the account to which the work is charged.

A USER can have several PROJECTS (of which one may be the default).

A PROJECT can have several BILLINGS (of which one may be the default).

The USER, PROJECT and BILLING values are stored in the catalog. At log-on (IOF), or job submission (batch), the system automatically checks the consistency of these parameters as supplied by the user. Any inconsistency will lead to denial of log-on (IOF) or rejection of the job (batch).

For more details, see the *Catalog Management Manual*.

3.2.4 Assignment of Cataloged Files

The resolution of the file reference is performed according to the catalog "search rules" mechanism. This mechanism can make use of several catalogs, successively scanned until the reference is resolved.

The catalogs are searched in the following order:

1. the catalogs which are explicitly attached to the user context; this can be done using the extended JCL statement ATTACH
   (up to five catalog-file-descriptions can be given).
2. the site catalog (SITE.CATALOG).
3. the system catalog (SYS.CATALOG).
4. the catalogs which are registered system-wide as "auto-attachable".

NOTE:
The search path can be overridden by use of CATALOG parameter in ASSIGN. For example,

```
$JOB ....
  ATTACH CATALOG1 = DEPT.CATALOG
         CATALOG2 = SITE.CATALOG
         CATALOG3 = INV.CATALOG;
STEP RST,LMLIB ...;
  ASSIGN CT1, INV.TOWN.STREET, CATALOG = 3;
.
.
  ASSIGN .........;
 .
 .

ENDSTEP;
```

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

However, the recommended practice is to use private catalogs and to make them auto-attachable. (Refer to the GCOS7 *Catalog Management User's Guide* for further details.)

3.2.5 Assignment of Uncataloged Files

As mentioned previously, information about a file such as its residency, journalization and sharing parameters can be stored in the catalog. For uncataloged files this information must be supplied, as follows:

- residency, via the MEDIA and DEVCLASS, or DVIDLIST or RESIDENT parameters of the ASSIGN JCL statement.
  When MEDIA is used, the name of the media must generally be specified by the user. However, this can be done by the operator if the special value * (MEDIA = *) has been specified.
- journalization, via the JOURNAL parameter of the DEFINE JCL statement.
- sharing, via the SHARE and ACCESS parameters of the ASSIGN JCL statement.

FOR EXAMPLE:

```
ASSIGN FILE-1, MYFILE, DEVCLASS = MS/D500, MEDIA = V1
       SHARE = MONITOR, ACCESS = WRITE;
```

❑

This assigns the file with external-file-name MYFILE to the internal-file-name FILE-1 (by which the file will be referenced in the user program). The device class is a disk drive (MS/D500) and the file resides on the volume named V1. File sharing is to be controlled by GAC (as specified by SHARE = MONITOR) and the file is to be written.

The following DEFINE statement,

```
DEFINE FILE-1 JOURNAL = BEFORE;
```

requests the use of the BEFORE journal for the file. Note that the DEFINE statement is associated with the ASSIGN statement by means of the internal-file-name FILE-1.

As a special case, uncataloged tape files can be assigned without reference to an external-file-name. An asterisk is specified instead of the external-file-name. For tapes, the file is identified by its position on the tape volume, specified by FSN (on the ASSIGN statement) or if FSN is not given, the first file on the tape volume is taken.

For example:

```
ASSIGN FILE-2,c, FSN = 3, DEVCLASS = MT/T9, MEDIA = TAPE3;
```

assigns the third file on the tape volume TAPE3 to the internal-file-name FILE-2.

In the case of uncataloged cassette and tape files, the value MEDIA = * can be specified. This means that the volume names are not known at job submission but will be supplied by the operator at run time. As the volumes and their sequence of mounting are under the complete control of the operator, care must be taken to ensure that he has a list of volumes in the correct sequence and that he can readily identify each volume.

3.3 File Allocation and Preallocation

ALLOCATE and PREALLOC (extended JCL statements) 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 an ASSIGN in the same STEP enclosure.

The VOLSET facility may be used for allocating cataloged and temporary files on mass storage. This provides a simplification of the JCL. Instead of specifying the media name and device class of a particular volume, the user can omit any location parameter, or enter the name of a volset. (For more information, refer to the *Administering the Storage Manager Guide*)

3.3.1 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. Temporary file organizations can only be UFAS, or LINKQD mono-subfile. PREALLOC can be used in cases where insufficient 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 4). 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 utility. Consequently, if a temporary file is not opened during the execution of a particular job step, the space will not be allocated.
- If no residency parameter is specified (RESIDENT, DEVCLASS, MEDIA, or VOLSET), the file is located on the job submitter's default volset, provided that the volset facility is active. If the volset facility is not active, the resident volumes are used.
- 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 e.g., 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 (normal termination) or by ABEND = PASS (abnormal termination).
- If an increment size is specified in an ALLOCATE for 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.

3.3.2 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 preallocation. It is the recommended procedure for all permanent disk files.
- 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 UFAS files or LINKQD mono-subfiles. The ALLOCATE statement can be used to specify the amount of space to be allocated to the file.

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 jobstep 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 *Section 4, 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 CREATES AND USES A TEMPORARY
DISK FILE SCR AND REFERENCES AN EXISTING PERMANENT FILE
ABC.PR';

STEP ST01, ABC.LD1;

   ASSIGN FILE1, SCR, TEMPRY, DEVCLASS = MS/D500, MEDIA =  12345;

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

   ALLOCATE FILE1, SIZE = 10, UNIT = TRACK;

   ASSIGN FILE2, ABC.PR, DEVCLASS = MS/D500, MEDIA = X42;


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

COMMENT 'AS TEMPORARY FILE SCR OF STEP ST01 NO LONGER EXISTS, THE
FILE NAME CAN BE USED IN ANOTHER STEP';

   STEP ST02, ABC.L02;

   ASSIGN NEX, SCR, TEMPRY;

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

ENDSTEP;

$ENDJOB;
```

❑

3.3.2.1 Preallocation of a Permanent Disk File

For UFAS, permanent files, space may be reserved and file labels may be created using the PREALLOC utility. This utility is described in detail in the *Data Management Utilities manual* and in the *UFAS User's Guide* as appropriate.

3.3.2.2 Preallocation of Cataloged Disk Files

For cataloged disk files, the catalog entry can be created using the CATALOG statement before the file is preallocated (see Catalog Management), or the CATNOW parameter of PREALLOC can be used instead (see the *Data Management Utilities User's Guide* for more details).

3.3.2.3 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. 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 UFAS sequential, 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, unless the file was created (ALLOCATE OR PREALLOC) with an INCRSIZE value specified.

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

The execution of an 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.N01, DEVCLASS=MS/D500, 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

LEVEL-64 SEQUENTIAL

FLR.

FD MISAJ

   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 UFAS 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.

3.3.2.4 Comparison of PREALLOC and ALLOCATE

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

Table 3-2. Comparison of PREALLOC and ALLOCATE statements

PREALLOC	ALLOCATE
Permanent files (cataloged or uncataloged) and temporary files	Permanent uncataloged or temporary files
Step Defining Statement	Placed inside a step enclosure with associated ASSIGN statement
Must be used for indexed sequential files	Only allocates sequential or direct BFAS files and all UFAS organizations
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 the 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, RECFORM, and RECSIZE)	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
If the file exists, PREALLOC terminates abnormally	CHECK feature for existing files

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

3.3.3 Tape Files

The term "tape" is used generically in the following paragraphs to denote both tape and cartridge.

At the beginning of a job step, system resources, such as 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.

The file label contains information from the ASSIGN statement (external-file-name, expiry date) and the file definition in the generating program, or DEFINE.

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, still 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 expiry date applying to the file (see *Section 4*). Cataloged tape files must be preallocated using the PREALLOC utility.

Work Tapes

A WORK tape is a tape volume that has been prepared by the Data Management Utility VOLPREP or TAPEPREP (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 an 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 tape. 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, that is, 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 the 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;
```

❑

Work tapes are temporarily allocated for the duration of the first step, and retain their work status at the end of the step. The tape used in the second step will lose its work status and become permanent.

3.3.4 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, 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 multi-volume file and treated as if the new tape has 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 (multi-volume) 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.

3.4 Use of Multi-volume 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, that is, all disk, same disk type (and for FSA disks, the same protection level: HRD, or HRD RDF, or HRD RDS, or HDR RD1), or all tape, same tape type. You must always supply volume names, in the ASSIGN statement, in the same order as they were specified when the file was written.

For sequentially organized files, you can indicate how many volumes of a multi-volume 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.

Multi-volume files can be temporary or permanent.

3.4.1 Partial Processing

If you require records from a subset of the volumes of a multi-volume file (e.g., in APPEND processing mode) you only need to specify the volumes required. Partial processing can be requested by using the FIRSTVOL and LASTVOL parameters of the ASSIGN statement to specify (respectively) the position in the media list of the "first" and "last" volumes to be processed.

EXAMPLE 1:

A cataloged multi-volume file.

The file MYFILE is contained in the volumes

A, B, C, D, E and F.

```
ASSIGN IN1, MYFILE, FIRSTVOL = 3, LASTVOL = 4;
```

The volumes C and D are the only ones processed.

❑

EXAMPLE 2:

An uncataloged multi-volume file.

The file OURFILE is contained in the volumes

V, W, X and Y.

```
ASSIGN IN2, OURFILE, LASTVOL = 2, DEVCLASS = MT/T9/D1600,
MEDIA = (V,W);
```

The volumes V and W are processed.

For tape, cassette files, the default value for LASTVOL is EOF (i.e., the last volume processed must contain the end of file). In the above ASSIGN statement if LASTVOL were omitted then the step would terminate abnormally as volume W does not contain the end of file.

❑

EXAMPLE 3:

File extension of an uncataloged multi-volume file. The file OURFILE is as described in *Example 2*.

```
ASSIGN IN3, OURFILE, DEVCLASS = MT/T9/D1600, MEDIA = (V, W, X, Y);
```

This leads to normal processing of the whole file (i.e., all the volumes).

Suppose that the file is subsequently extended onto another volume Z. The above ASSIGN statement will lead to an abnormal termination as volume Y no longer contains the end of file. This warns you (when you have forgotten to update your media list with the new volume Z) that you have not processed the whole file.

If you wish to process volumes V, W, X and Y only, then LASTVOL must be specified as follows:

```
AGGIGN IN4, OURFILE, DEVCLASS = MT/T9/D1600, MEDIA = (V, W, X, Y)
LASTVOL = 4;
```

Here LASTVOL is necessary, but should only be used when the processing mode is input.

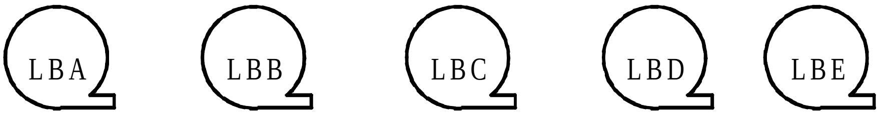
❑

3.4.2 Multi-volume Work Tapes

A multi-volume temporary tape file can consist entirely of WORK tapes. If the file is permanent, and 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 to be used in references to the file in subsequent jobs.

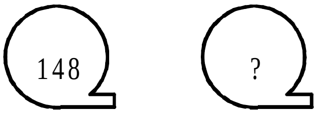
A multi-volume permanent uncataloged tape file can be entirely on WORK tapes, but such tapes cease to be WORK tapes once the file has been written on them.

File FNAL.A



Program only reads records within the volume LBC and LBD. Does not read to end of file so LBE is not needed.LASTVOL must be specified as the volume LBD does not contain the EOF. If LASTVOL is omitted, then LASTVOL=EOF is assumed and the step terminates abnormally.

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/M452, MEDIA=MSD);
ASSIGN     FLA, FNAL.A, DEVCLASS=MT/T9, MEDIA=(LBC, LBD) LASTVOL=2;
ASSIGN     FLB, NCU.BX, DEVCLASS=MT/T9, MEDIA=148;
ENDSTEP;
.
.
```

Figure 3-1. Partial/Extensible Multi-volume Processing

EXAMPLE:

```
COMMENT 'THE NEXT STATEMENT ASSIGNS A MULTIVOLUME DISK FILE';
ASSIGN FILA, MST.PLN, DEVCLASS = MS/D500

   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.MTSPLN, 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 4*.

❑

3.5 Multi-file Tape Volumes

A tape volume may contain one file (a mono-volume file), part of a file (multi-volume file) or it may contain more than one file, in which case it is known as a multi-file volume. All files on a standard multi-file tape volume must be uncataloged. Non-standard multi-file tape volumes are not supported. A file on a multi-file volume cannot have an expiry date that is later than the expiry date of any of the files that precede it on the volume.

3.5.1 Useful Parameters of ASSIGN

There are parameters of the ASSIGN statement that are useful in processing of multi-file tape volumes. They are the END, ABEND, and FSN parameters.

With the END and ABEND keywords, the special value LEAVE ensures that a multi-file 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, which gives the file sequence number of the file to be assigned, must be specified for multi-file tapes. Sequence numbers of files start at 1 and must not normally exceed 253. (The exception to this is the INFILE parameter of the CREATE command, where FSN may be specified as any value up to and including 258.)

There are two special values for FSN: NEXT and ANY.

The NEXT value is available only in output mode when using multi-volume files on multi-file tapes. Specify ANY or a number to process multi-volume files in input mode.

If FSN = ANY 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. All files which follow the over-written file are lost. If there is no file of the specified name, 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 NEXT 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.

3.5.2 File Concatenation

3.5.2.1 Omitting Internal File Name on ASSIGN

Several UFAS standard sequential or cassette files can be accessed as if they were a single sequential file. File concatenation, as it is called, can be specified by successive ASSIGN statements in the 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.

Instead of a using a comma, the missing internal file name can be indicated by the symbol #. Consequently the last two ASSIGN statements above could be written,

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

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

3.5.2.2 Uncataloged Tape Files

In the case of uncataloged tape files and diskette files, concatenation may also be achieved by use of the NBEFN parameter. This parameter gives the number of files to be concatenated starting from the "first" file which is specified by its external-file-name or by FSN. On multi-file tapes the sequence of concatenation is the physical order in which the files are on the volume.

EXAMPLE 1:

```
ASSIGN INT1, FILE3, NBEFN = 5, FSN = 3, DEVCLASS = MT/T9/D1600,

       MEDIA = A5;
```

This concatenates 5 files on a multi-file tape. The concatenation starts at the 3rd file on the tape.

❑

3.5.2.3 Restrictions

Whatever the method of concatenation used, the following restrictions apply:

- All the files must be sequential.
- There may also be restrictions on the difference in attributes (e.g., record size, record format) between the files concerned. These restrictions are dependent on the programming language being used.
- If a UFAS file with SHARE = MONITOR (specified in the catalog or in the ASSIGN statement) is included in the concatenation process then

  Either (i) Specify ACCESS = SPREAD (ASSIGN statement)

  Or (ii) Specify READLOCK = STAT (DEFINE statement)

  Or (iii) the step must be repeatable (REPEAT parameter of STEP or $JOB statement)

3.6 Deallocation of File Space

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), tape file (cataloged or uncataloged).

3.6.1 Uncataloged Tape File

Tape file deletion can be performed by overwriting the file contents after the expiry date However, if you also wish to change the name or owner of the file (and have the right to do so), use the VOLPREP utility.

3.6.2 Cataloged Tape Files

A cataloged tape file can be deallocated using DEALLOC. The file name remains in the catalog unless the UNCATNOW parameter of DEALLOC is used. Alternatively, the catalog entry can be deleted using the UNCAT statement. If only the file itself is to be deleted, you can use VOLPREP.

When a cataloged file (on tape or cartridge) is opened in write mode, if a previous version of that file is referenced in the catalog, the existing list of media is used. The system asks for all media to be mounted (even if the new file is smaller than the previous one, and requires less media). When the file is closed, any unused space is deallocated; and the list of media is updated in the catalog.

3.6.3 Permanent Disk Files

A file is deleted when its file label is deleted from all its volumes. A file may be deleted in a step using utilities DEALLOC or LIBDELET. With the volume preparation utility program (VOLPREP) it is possible to release the space occupied by all files on a volume.

Cataloged Disk Files

The space is released, as for permanent disk files, but the file name and description will remain in the appropriate catalog until it is deleted by use of the UNCAT statement (see Catalog Management), or the UNCATNOW parameter is specified with DEALLOC (Data Management Utilities). Both methods are subject to expiry date checks, if appropriate

3.7 Duplicate File and Volume Names

GCOS supports duplicate file names on different volumes. For example, FILE A on volume X is distinguished from FILE A on volume Y. The residency of the files (which in this case is the means of distinguishing between them) can be specified by the ASSIGN statement or by cataloging the files on different catalogs.

The use of such duplicate file names is not recommended for the following reasons:

- in the case of uncataloged files, an error in specifying the residency parameters can lead to the assignment of the incorrect file,
- in the case of cataloged files, an error in the order of the ATTACH statements will affect the search rules and will lead to the assignment of the incorrect file.

This recommendation also applies to catalogs themselves. To apply this principle to catalogs, the following rules should be observed;

- all private catalogs are given different names so that they can all be cataloged in the site catalog,
- master directories in all catalogs, including the site catalog are given different names,
- it is recommended that there should be only one master directory in each private catalog, and in that case, the master directory name is the same as the catalog name (e.g., the master directory in the catalog C1.CATALOG is named C1).

GCOS supports duplicate volume names which have different attributes, for example, volume X (CT/M5 device-class) is distinguished from volume X (CT/M6 device-class).

The use of such duplicate volume names is not recommended. You can request the incorrect volume by making an error in the device-class specification. Such an error will not be detected by GCOS. Duplicate volume names are also undesirable from the operator's point of view.

In addition to the problems discussed above, duplicate file and volume names can cause confusion in user documentation.

3.8 Overview of the DEFINE Statement

The DEFINE statement allows you to supply information about 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, but the contents of the file label will override the DEFINE information. The exception to this is when a non-native tape file is indicated (FILEFORM = NSTD), when any file labels are ignored.

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 (UFAS only) and control area size (UFAS only)
- control interval (UFAS only) and control area free space (UFAS only)
- the frequency with which checkpoints are taken
- unit record device control options
- file journalization can be requested.

3.9 GCOS Overriding Rules

When specified, and if DEFINE does not indicate a non-native tape file, the file label provides the following:

- file configuration
- record length
- block size
- for UFAS tape files, the specification or omission of block sequence numbers
- size and location of the key
- whether deleted records are to be allowed or not
- CI and CA size available space (UFAS)
- whether the file is cataloged 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 (if not retrieved from the catalog)
- the level of sharing and access allowed to the file (if not retrieved from the catalog).
- whether the file is temporary, permanent uncataloged or permanent cataloged
- whether the file is multi-volume (if not retrieved from the catalog).

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.

3.10 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, as described below.

3.10.1 Using the Master Directory

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.

Prefixing using the master directory 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.

3.10.2 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 were frequent reference to files whose first two component names were DEPT1 and SECT2, (DEPT1.SECT2. INVENTORY, 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 need 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.

❑

4. Resource Management

4.1 Introduction

GCOS provides you with JCL facilities that 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 you can influence the allocation of resources within a step enclosure and from job step to job step.

Before a job step is initiated, the system refers to your 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, and is not released if the job is held or suspended (see *section 1*).

4.2 Memory Management

4.2.1 Concept in GCOS 7

The virtual memory concept implemented in GCOS frees you from problems associated with program structure (e.g., segmentation, transaction sequences), since it will appear that you have one large area of memory for your exclusive use. (For further details, see the *System Administrator's Manual*.) 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 an increase in the number of segment transfers between the backing store and main memory.

Information is made available in the Job Occurrence Report (JOR) from which you 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 SEGMENTS number and PROG MISSING SEGMENTS number (see Section 8, Job Occurrence Report).

The MISSING SEGMENTS number indicates the number of system segment and user program segment 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.

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.

4.2.2 Declared Working Set

The declared working set (DWS) is the physical memory size in units of 1K (1024 bytes), required by the code/data segments and the control structures of a job step. This figure should be the optimal requirement that allows a program to execute without any significant loss in performance (or excessive memory allocation).

Information appears in the Linker listing and JOR that 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. The JOR gives the number of buffers used, channel program page size and the number of missing segments.

Tuning of the DWS value is described in the *System Administrator's Manual*.

4.3 File Passing

4.3.1 Description

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 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.

In order to overcome these problems, files may be passed from one job step to a later one, using the END and ABEND parameters (with value PASS) in an 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/D500,
                          MEDIA = (5DX143,DX127);

. . .

. . .

ENDSTEP;

STEP       ST02,   ABC.LD1;

ASSIGN     JACK,   ABC.SCR, TEMPRY;

ASSIGN     JILL,   ABC.PR, DEVCLASS=MS/D500,
                   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/D500
                  MEDIA=(DX143,DX127), END=PASS;
...
...
ENDSTEP;
STEP     ST02,    ABC.LD1;
ASSIGN   JACK,    ABC.SCR, TEMPRY;
ASSIGN   JILL,    ABC.PR;
...
...
ENDSTEP;
```

In the modified example, both files are passed 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 in 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, because END = PASS has not been specified for the file in that step.

The following general points apply to the passing of files:

- "Passing the file" means that the name, attributes, resource requirements and sharing mode (SHARE, ACCESS) apply until the next ASSIGN of the relevant external-file-name (efn), is performed in the job. (However, note that disk files are not protected against the VOLPREP utility).
- 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 that contains a passed file is not normally reserved. Therefore another job could acquire access to the same volumes to use a different file or to share the same file (see "*File Sharing*", later in this Section).
- A file may be passed across job steps that do not refer to the file at all. In these cases, the resources are reserved throughout the job until an ASSIGN statement that refers to the file is encountered (see the example job TRY below).

4.3.2 Rules for Passed Files

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

1. The system looks for a catalogued file of the given name in the currently attached catalog(s).
2. If there is no cataloged file of 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.

File passing is therefore the means of ensuring that temporary information produced or used in one job can be accessed in a later step in the job.

EXAMPLE:

```
$JOB TRY, ...;

STEP LM1, ...;

ASSIGN MAN1,    SUNDAY.REP
                DEVCLASS=CT/M5,   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.

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 (if LM2 does not access this file). 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 6), 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.

❑

4.3.3 Deadlock Situation

When the 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 4-1*.

JOB STREAM A	JOB STREAM B
`...`	`...`
`STEP   LM5 YY.MA;`	`STEP   LM3 YY.MB;`

`ASSIGN  I1 FILEA`
`END=PASS...;`
	`ASSIGN  J1 FILEX`
	`END=PASS...;`

`...`	`...`
`...`	`...`
`ENDSTEP;`	`ENDSTEP;`
`STEP  LM6 YY.MA;`	`STEP  LM4 YY.BB;`
`ASSIGN 12 FILEA...;`	`ASSIGN J2 FILEX...;`
`ASSIGN 12J FILEA...;`	`ASSIGN J21 FILEA...;`
`...`	`...`
`...`	`...`
`ENDSTEP;`	`ENDSTEP;`
`...`	`...`

Figure 4-1. 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" file FILEA, cannot start execution until it can access file FILEX, but will not release FILEA (to step LM4) until it has completed execution. Correspondingly, step LM4 cannot start execution until step LM6 has released FILEA. 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.

4.4 File Protection

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

GCOS provides this function in two main ways:

- By assignment, files are protected against non-requested access, since no file can be processed unless the user supplies its name and supporting volume identification.
- 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 provided 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 (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). For cataloged files, access rights can be used to restrict the type of access to a file.

4.5 File Sharing Without GAC

The following paragraphs describe file sharing where GAC (General Access Control) is not active. 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 some 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 the 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, and only the ACCESS parameter of ASSIGN need be used. *Figure 4-2* illustrates the simultaneous access of the same file by two concurrent jobs.

Job R1 ...

```
STEP .......;

COMMENT 'REQUEST READ ACCESS TO HJ.OMN
         AN UNCATALOGED FILE ALLOWING
         OTHERS READ-ACCESS ONLY';
ASSIGN   COMP, HJ.OMN, DEVCLASS=MS/D500 MEDIA =
(DX143,DX127)
         ACCESS = READ, SHARE=NORMAL ...;
...
...
...
ENDSTEP;
$ENDJOB;
```

Job R2 ...

```
STEP .......;
ASSIGN   COMM, HJ.OMN, DEVCLASS=MS/D500 MEDIA =
(DX143,DX127)
         ACCESS = READ, SHARE=NORMAL ...;
ENDSTEP;
$ENDJOB;
```

Figure 4-2. Inter-job File Sharing

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 files. 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 cannot be granted, the job is aborted.

Two or more concurrent assignments with different SHARE values are not permitted except between SHARE=NORMAL and SHARE=ONEWRITE. For example, if a job is running with the values SHARE=FREE, ACCESS=WRITE (see ACCESS below), a request for SHARE=ONEWRITE, ACCESS=READ will not be granted until the first job has released the file. The required type of concurrent access would be permitted if the two requests were:

```
SHARE=ONEWRITE, ACCESS=WRITE
```

and

```
SHARE=ONEWRITE, ACCESS=READ
```

The ACCESS parameter can have one of six values, WRITE, READ, ALLREAD, SPWRITE, SPREAD, RECOVERY. Of the possible SHARE values, the ones recommended for standard use are NORMAL and ONEWRITE. *Table 4-1* shows the possible combination 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, (i.e., within the step the file cannot be assigned with another internal file name).

Table 4-1. File Sharing Requests

Keyword Values		**Type of Sharing Requested**
ACCESS	**SHARE =**	
	NORMAL	Exclusive use (default)
WRITE		
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
ALLREAD	NORMAL	Read while any job reads
ALLREAD	ONEWRITE	Read while any job reads, no job writes
RECOVERY	NORMAL	Exclusive access for file recovery purposes

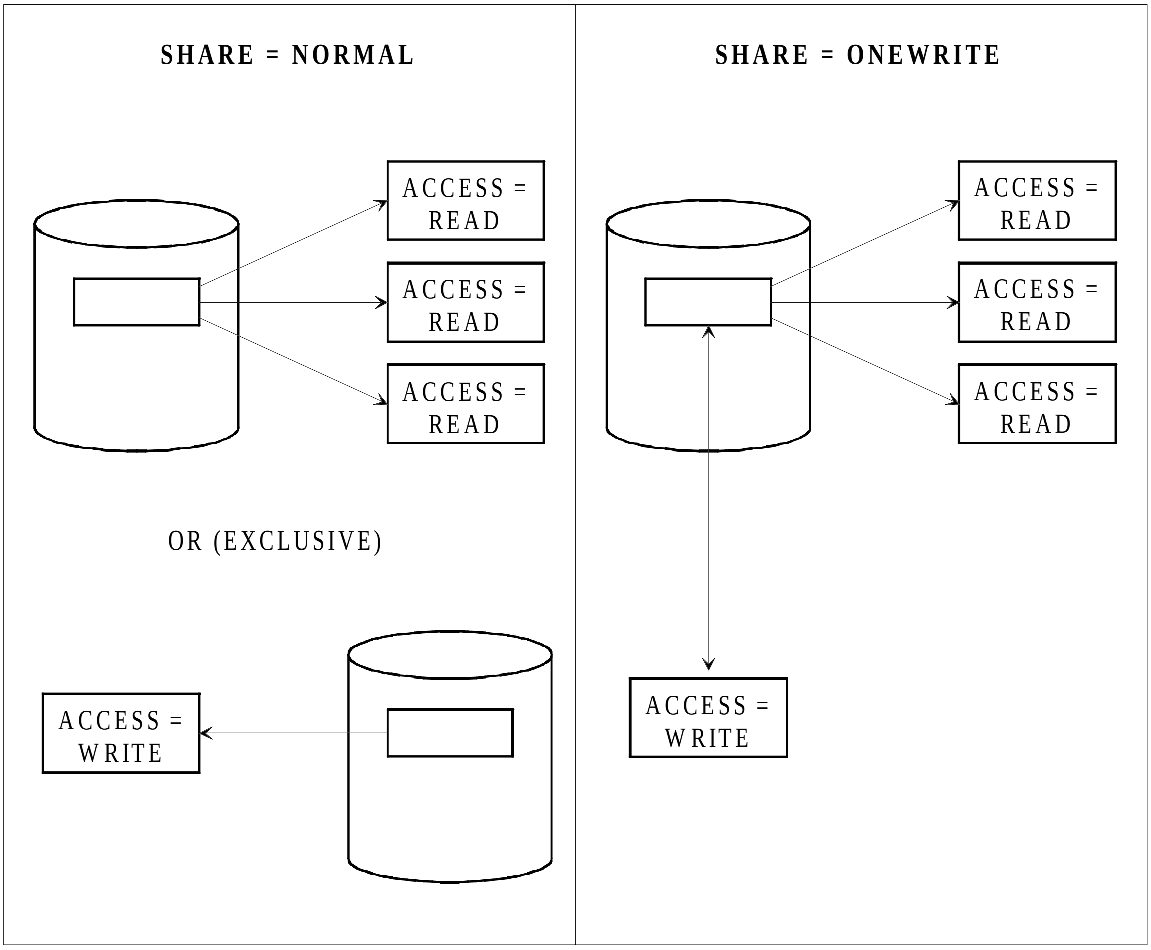


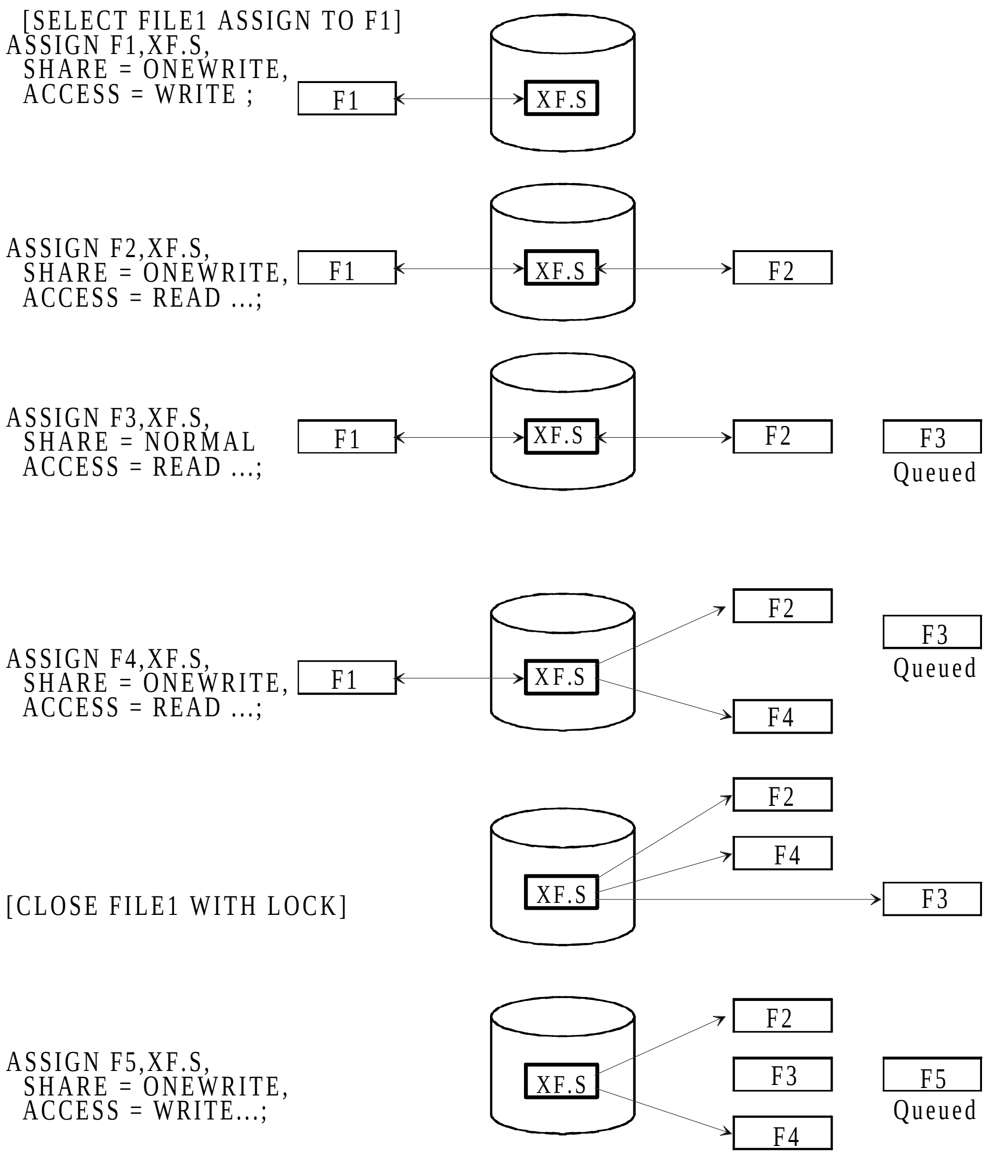
Figure 4-3. Shared Access to File

The SHARE parameter defines a protocol for sharing the file. Generally, users must specify the same protocol if the file is to be shared by them. For cataloged files, the protocol is enforced via the catalog.

Figure 4-3 illustrates the rules that the system follows when it tests whether 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 of the 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, except that other accesses to the file are restricted to the same step (i.e., multiple assignments).

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



```
  [SELECT FILE1 ASSIGN TO F1]
ASSIGN F1,XF.S,
  SHARE = ONEWRITE,
  ACCESS = WRITE ;

ASSIGN F2,XF.S,
  SHARE = ONEWRITE,
  ACCESS = READ ...;

ASSIGN F3,XF.S,
  SHARE = NORMAL
  ACCESS = READ ...;

ASSIGN F4,XF.S,
  SHARE = ONEWRITE,
  ACCESS = READ ...;

[CLOSE FILE1 WITH LOCK]

ASSIGN F5,XF.S,
  SHARE = ONEWRITE,
  ACCESS = WRITE...;
```

Figure 4-4. **File Sharing Example**

NOTES:

1. Sharing is possible on disk only. A tape user always has exclusive read access or exclusive write access.
2. If space is being allocated for a file within a step (ALLOCATE), the file can only be 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 for this file in the step.
3. When a file is passed (END=PASS), the sharing mode of the file remains until the next ASSIGN for the file. This ASSIGN may declare a new (or the same) sharing mode.
4. The system does not check that the sharing mode requested is supported by the file organization. You should make sure that the two are compatible.
5. The catalog contains SHARE information. If the value of SHARE specified via ASSIGN (either explicitly or implicitly default) is different from 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.
6. 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).

The following apply when GAC is not used for file sharing:

- Sharing in batch, IOF and under QUERY is controlled at file level; that is, a user is granted or denied access to the entire file. Therefore, if you want exclusive access to a single record in a file, once this access is granted other users are prevented from accessing any record in the file.
- Sharing in batch, IOF and QUERY is controlled for the duration of the current step on the basis of the assignment of a file; that is, conflicting requests are queued until the completion of the step currently accessing the file (or until the file is deassigned).
- Under batch, IOF and QUERY, the default rule (if nothing else is specified) is that no more than one user can write to a file at any one time.

  You may override this rule if you choose, but if you do, then to preserve the integrity of the data you must either (i) take care that your actions do not lead to inconsistent data or (ii) establish synchronization controls.
- Under TDS, no concurrent write access is permitted to non-controlled files (i.e., where GAC does not apply), even among users (transactions) in the same TDS Job.

Any sharing of a file among users in different processing environments (e.g., between batch and QUERY users or between batch and TDS users) is subject to the above constraints: only one writer at a time, control at the level of the file and the step.

4.6 File Sharing With GAC

File sharing where GAC is active is discussed in the following paragraphs. GAC is activated by specifying SHARE=MONITOR in the ASSIGN JCL statement for uncataloged files or in the catalog for cataloged files. Concurrent access to a file between GAC users and non-GAC users is not possible. Consequently, if a file is assigned to a user under SHARE=MONITOR, access will be denied (until the file is released) to users who wish to assign the file with any other value of SHARE. Conversely, if a file is currently assigned to a user with SHARE=any value (except MONITOR), users wishing to assign the file under SHARE=MONITOR will be denied access.

This fact should be borne in mind during job and job class scheduling. Jobs which share a file under SHARE=MONITOR should not be run concurrently with jobs which share the same file under any other value of SHARE.

GAC permits several simultaneous accesses (reads, writes) from several programs to the same files. Each program has a coherent view of the files irrespective of what the other programs are doing.

While a file is being shared under GAC (that is, SHARE=MONITOR), sharing is controlled at the CI (Control Interval) level at access time. This should be contrasted with the situation without GAC where file sharing is applied at the file level (i.e., access is granted or not granted to the whole file) at assign time. Under GAC, a file can be accessed and updated concurrently by batch, transactional (TDS), IOF, and QUERY users. Even though users do not have exclusive access to the file, the consistency and integrity of data is ensured. Deadlock situations (e.g., where User A cannot proceed until User B releases a CI and User B cannot proceed until User A releases a CI) are resolved automatically.

4.7 Expiration Dates

4.7.1 Introduction

The existence of an expiry date on a file provides security against an accidental deallocation of the file (or against a simple overwriting with a new file if an uncataloged tape file is concerned). However, note that it does not prevent a program from modifying the contents of the file (in output, append, or update processing mode); it is the file space that is protected, not the file contents. Expiration dates apply to permanent files only. The date is recorded in the file label.

4.7.2 Uncataloged Tape Files

The expiry date is set, through the EXPDATE parameter of the ASSIGN statement, when the file is opened in output processing mode. For example:

```
ASSIGN IFN, FILE1, DEVCLASS=MT/T9, MEDIA=1600
       EXPDATE=94/10/1;
```

Later on, the same parameter can be used to modify the expiry date when the file is assigned in output processing mode.

If you want to re-utilize the same tape for another file before the expiration date of the file is over, you must use the VOLPREP utility with the BYPASS parameter. For example:

```
VOLPREP OLD=(DEVCLASS=MT/T9, MEDIA =1600)
        NEW=(DEVCLASS=MT/T9, MEDIA =1600)
        BYPASS;
```

4.7.3 Cataloged Tape Files

A cataloged tape file must be preallocated. The expiry date can be set at that time. For example:

```
PREALLOC PROJECT.FILEA, DVC=MT/T9
    BFAS=(SEQ=(BLKSIZE=1024, RECSIZE=200))
    GLOBAL=(MEDIA=1600)
    FILESTAT=CAT, CATALOG=1, CATNOW
    EXPDATE=94/10/1;
```

It can be set later (or modified if set at allocation time) through the EXPDATE parameter of the ASSIGN statement, when the file is opened in output mode.

If you want to deallocate the file before the expiry date is over, you must use the BYPASS parameter of the DEALLOC utility. For example:

```
DEALLOC PROJECT.FILEA, UNCATNOW, BYPASS;
```

4.7.4 Uncataloged Disk Files

There are two ways of setting (at allocation time) the expiry date of uncataloged disk files, depending on the method chosen to allocate the file: using ALLOCATE or PREALLOC. For example:

- the first way, using ALLOCATE

```
ASSIGN IFN, FILE1, DVC=MS/D500, MD=K047
       EXPDATE=94/10/1;
ALLOCATE IFN, SIZE=50, UNIT=CYL;
```

- the second way, using PREALLOC

```
PREALLOC FILE1, DVC=MS/D500
    BFAS     = (SEQ= (BLKSIZE=1024, RECSIZE=200))
    GLOBAL   = (MEDIA=K047, SIZE=50)
    FILESTAT = UNCAT
    EXPDATE  = 94/10/1;
```

The expiry date can be set later (or modified if set at allocation time) by using the EXPDATE parameter of the ASSIGN statement when the file is opened in output or update mode. It is also possible to modify the expiration-date or/and the efn of the file using the utility program FILMODIF. For example:

```
FILMODIF
   INFILE = (PAYFILE, DVC=MS/D500, MD=K047)
   OUTFILE = (PAYROLL, EXPDATE=94/1/1);
```

The efn of the file is changed from PAYFILE to PAYROLL and the new expiry date is 94/1/1.

If you want to deallocate the file before the expiry date is over you must use the BYPASS parameter of the DEALLOC utility. If you want to carry out volume preparation of a disk containing files that have unexpired expiration dates, you must also use the BYPASS parameter in the VOLPREP utility.

4.7.5 Cataloged Disk Files

There is only one way of setting, at allocation time, the expiry date of a cataloged disk file since the only way to allocate space for such a file is via the PREALLOC utility. For example:

```
PREALLOC PLANT1.PAYFILE, DVC=MS/D500
     BFAS = (SEQ = (BLKSIZE=1024, RECSIZE=200))
     GLOBAL = (MEDIA=K047, SIZE=50)
     FILESTAT = CAT, CATALOG=1, CATNOW
     EXPDATE = 94/10/1;
```

Expiry date and efn can be modified in the same way as for uncataloged disk files. The BYPASS parameter is necessary to deallocate the file before the expiration date is over. However, you cannot use VOLPREP on a disk volume containing cataloged files (except for recovery purposes by the SYSADMIN project using the FORCE parameter). Such files should be deallocated before VOLPREP is used.

EXPDATE can be expressed in three different ways:

1. YY/MM/DD year, month, day
2. YY/DDD where DDD is the number of the day in the year
3. DDD where DDD is an expiry period, i.e., the number of days added to the date of the run that sets the expiry date.

EXAMPLE:

If a program is run on the 95/1/14, then EXPDATEs of 95/2/14, 95/45 or 31 are equivalent.

EXPDATE should not be confused with RETPER (Retention Period) which is recorded in the catalog entry of a generation-group. RETPER is used every time you record a new-generation to automatically calculate the expiry date of the run. Since every generation of a group has a different name (due to the different generation number), the protection of the space also provides a protection of the contents. It is possible to override the RETPER mechanism by giving an EXPDATE in the ASSIGN statement of a generation when you write it. For more details refer to the *Catalog Management User's Guide Manual.*

❑

4.8 Device Management

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 in the step description. In practice, there are two areas of application where device utilization can be minimized. These concern the mounting of multi-volume files and the situation where different files using the same (type of) device are processed one after another rather than simultaneously.

NOTE:
For specific use of cartridge devices via a cartridge library, refer to the *Cartridge Tape Library User's Guide* or *CTL-UNIX Server User's Guide*.

4.8.1 Mounting of Multi-volume Files

The default is that just one volume is mounted at a time. The default options correspond to the default values of the MOUNT parameter of the ASSIGN statement.

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);
```

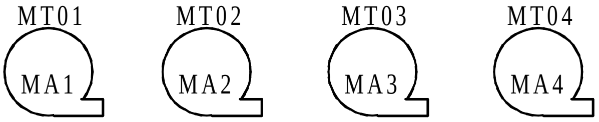
so that only one drive is requested instead of four. In this case, as MOUNT has not been specified, the default MOUNT=1 applies. The MOUNT parameter in the ASSIGN statement indicates the number of volumes to be mounted simultaneously. The use of MOUNT is obviously essential when there are not sufficient devices available for all the volumes of a file.

The most useful values of MOUNT for multi-volume tape files are MOUNT = 1 and MOUNT = 2.

IF MOUNT = 1 is specified, or MOUNT is omitted, 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 it minimizes 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 each volume in advance, so switching is not delayed by operator intervention (see *Figure 4-5*).

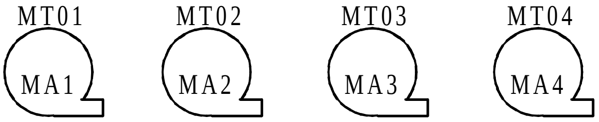
```
COMM     'ALL VOLUMES MOUNTED SIMULTANEOUSLY';
ASSIGN   GLBE,REL.X,FILESTAT=UNCAT,DEVCLASS=MT/T9
         MEDIA=(MA1,MA2,MA3,MA4),MOUNT=4;
```



4 drives used

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

NOTE: As MOUNT=1 is the default for tape, this parameter has been omitted from the above ASSIGN statement.



only 1 drive used

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

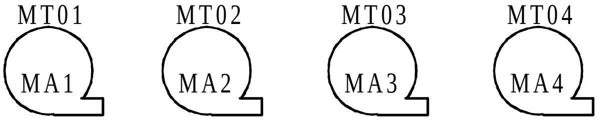


Figure 4-5. Multi-volume Device Management

The use of MOUNT applies to both permanent and temporary tape files.

The MOUNT parameter continues to have effect when a normal tape file overflows onto a WORK volume.

The MOUNT parameter can also be applied to the use of ASSIGN for multi-volume files. MOUNT cannot be used on a file which has multiple assignments (i.e., more that one ASSIGN in a job step, each having the same external file name but a different internal file name). MOUNT is not allowed when a multi-volume file is being allocated (for example, in conjunction with an ALLOCATE statement).

The MOUNT parameter may be used with a device pool (see below). If an 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.

4.8.2 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. You can inform GCOS that this is possible by using the POOL statement in conjunction with the POOL parameter in ASSIGN:

```
POOL 1*CT/M5;

ASSIGN F1, MAX.Z, DEVCLASS=CT/M5, MEDIA=VOL1, POOL, FIRST, ...;

ASSIGN F2, BMY.I, DEVCLASS=CT/M5, MEDIA=VOL2, POOL, FIRST, ...;
```

Thus only one cartridge device will be reserved for the use of the pooled files.

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 indication 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=VOLA

     POOL, FIRST;

  ASSIGN IFN2, SEC.FILE, DEVCLASS= MT/T9, MEDIA=VOLB

     POOL, NEXT;

  ASSIGN IFN3, THIRD-FILE, DEVCLASS= MT/T9, MEDIA=VOLC

     POOL, NEXT;

  ENDSTEP;

$ENDJOB;
```

The mounting of volume VOLA will be requested before the load module execution is initiated. When the OPEN FC2 is executed, volume VOLB will be mounted and when OPEN FC3 is executed, volume VOLC will be mounted. 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 cartridge files MAX.Z and/or BMY. I were each on two volumes, the POOL statement would be:

```
POOL 2*CT/M5
```

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 multi-volume 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...;

POOL2*MS/D500;

POOL1*MT/T9;

ASSIGN F1, AX.BM, DEVCLASS=MS/D500, MEDIA=4D2S, POOL, FIRST;

ASSIGN F3, AX.BT, DEVCLASS=MS/D500, MEDIA=4D2E, POOL, FIRST;

ASSIGN F2, AX.BP, DEVCLASS=MS/D500, MEDIA=4D20, POOL, NEXT;

ASSIGN F4, AX.BZ, DEVCLASS=MS/D500, MEDIA=4D2P, POOL, NEXT;

ASSIGN F5, AX.CD, DEVCLASS=MS/D500, MEDIA=4D2Z;

ASSIGN F6, P4.DI, DEVCLASS=MT/T9,   MEDIA=(T41, T42, T43),
POOL FIRST;

ASSIGN F7, P4.DP, DEVCLASS=MT/T9,   MEDIA=T63, POOL, NEXT;

ASSIGN F8, P4.D3, DEVCLASS=MT/T9,   MEDIA=T71;
ENDSTEP;
```

❑

In the above example, if POOL had not been used we would require 5 disk drives and 3 tape drives. With POOL 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 file P4.D3 do not access pooled devices.

NOTES:

1. You 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.
2. All ASSIGN statements for a particular device pool may specify POOL, NEXT. This is particularly useful when the order in which the opening of the files will take place is not necessarily known in advance.

5. Maintenance of Stored JCL and Parameter Substitution

5.1 Introduction

When a set of JCL statements is to be used more than once, it is convenient to store it in a member of a library 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, you can submit to the Stream Reader a stored job stream for translation and execution. Alternatively, you 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*.

Job streams and JCL sequences can be stored in members of permanent library files. You can create and maintain each library member by means of LIBMAINT, which has its own comprehensive text editing facility, or by the Full Screen Editor (FSE). 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 and a parameter substitution facility, so that you can modify stored JCL sequences and job streams dynamically to suit the requirements of a particular job run.

Figure 5-1 gives an example of the corresponding job description. Note that since only one job is involved here, an INVOKE or an EXECUTE statement could be used (provided the $JOB and $ENDJOB statements are removed from the stored file and there are no input enclosures).

```
$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=*INDATA;
$INPUT INDATA;
            MOVE COMFILE:NEWJCL, TYPE=JCL, NUMBER;
             . . .
             . . .
          < JCL statement>
             . . .
             . . .
$ENDINPUT;
$ENDJOB;
```

Figure 5-1. Example of Storing JCL in a Library Member

5.2 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" job) 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 that 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.

Note that a JUMP statement cannot be used to jump to an INVOKE statement, nor to any other translator statement.

Unlike the INVOKE statement, EXECUTE is activated at execution time; no replacement is made at JCL 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. Therefore EXECUTE represents at least two steps at job execution time, the first step is the translation of the sequence, then execution of enclosed steps; i.e., translation and execution.

The operator SJ and SI commands perform the same function as RUN but may be entered from the operator's console or by an IOF user. The operator can exercise subsequent control over any job using other operator commands (in particular, the MJ command).

NOTES:

1. 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.
2. In this Section, the terms 'INVOKEd' and 'EXECUTEd' will be used to identify stored JCL sequences that are accessed by INVOKE and EXECUTE respectively.

5.3 Use of Run

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 $JOB statement (if any)
- parameters of the RUN statement
- parameters of the $JOB statement of the spawning job or log-on parameters.

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. If member-name exceeds 8 characters, then only the first 8 characters are used, the user-name is assumed to be the same as that of the RUN statement, and the REPEAT parameter cannot be specified.

The parameters of the $JOB statement for each stored job override those of the RUN statement and those of the $JOB statement of the spawning job or log-on parameters. The following rules apply:

job-name
Job-name in stored $JOB statement.

user-name
User-name in stored $JOB statement. Note that user-name is optional in a 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, default class of project as stored in the catalog;
if none is specified in the catalog, class P.

scheduling-priority
Parameter of RUN statement;
if none, parameter of stored $JOB statement;
if none, default associated with spawned jobclass.

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.

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

The above rules can also be expressed 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.

**Table 5-1.	Example of Storing JCL in a Library Member**

PARAMETER	**RUN SJ/SI**	**STORED JOB**	**SPAWING JOB**	**DEFAULT**
Job-id	2	1	-	member-name*
User-id	-	1	2	-
Project	-	1	2	catalog
Billing	-	1	2	catalog
Job-class	1	2	-	default class of project
Sch-pr	1	2	-	class default
HOLD	1	1	-	not held
HOLDOUT	1	1	-	not held
REPEAT	-	1	-	not repeat

The spawning job can pass information to spawned jobs through the use of switches (see the *JCL Reference Manual*, RUN statement).

In installations that have implemented Access Rights, only the main operator may launch jobs for other users. For example:

if USER = X in the $JOB statement of the stored job, a user with USER not equal to X cannot launch the stored job.

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

EXAMPLE 1:

Assume that a job stream containing a single job with job name T32 (in its $JOB statement) 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;
```

where T32 comes from the stored $JOB, X123 and INVHQ come from the spawning $JOB, and P is the default.

Now, if instead the RUN statement were:

```
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,

HOLD, PRIORITY = 2;
```

where T32 comes from the stored $JOB, X123 and INVHQ come from the spawning $JOB, and K, HOLD, 2 come from 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 statement 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;

where MONDAY, AQ47, BRW, M come from the stored $JOB.

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

where TUESDAY, AQ48 come from the stored $JOB, INVHQ comes from the spawning $JOB, and P is the 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 described in Section 6.

5.4 Use of Invoke And Execute

The INVOKE statement refers to a stored JCL sequence. The INVOKE statement is replaced by the referenced JCL sequence as described above. Note that the stored JCL sequence must not contain JCL statements which are handled by the Stream Reader ($JOB, $ENDJOB, $INPUT, $ENDINPUT, $DATA, $ENDDATA or $SWINPUT).

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

```
PREALLOC MYFILE.INV

   DEVCLASS=MS/D500, 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/D500, FILESTAT = CAT

   GLOBAL = (MEDIA = C112, SIZE = 5)

UFAS = (SEQ = (CISIZE = 1024, RECSIZE = 100));

   STEP LM1...;

ENDSTEP;

$ENDJOB;
```

5.4.1 Input Enclosures Referenced from Stored JCL

As $INPUT and $ENDINPUT cannot be used in an 'INVOKEd' stored JCL sequence, then, an input enclosure cannot be contained in such JCL sequences. However, the sequence 'INVOKEd' can reference input enclosures of the job containing the INVOKE statement, 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.

5.4.2 Independence of INVOKEd JCL Sequences

Labels

A label which is defined inside an INVOKEd JCL sequence cannot be referenced outside the sequence. In addition, it is not possible to jump outside the sequence. Therefore you 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 MYJCL:

```
   STEP LM1...;

   ...

ENDSTEP;

JUMP A, STATUS, NE, 0;

   STEP LM2...;

   ...

ENDSTEP;

A:
```

The following job description:

```
$JOB,,,;

   A: STEP S1,...;

...

ENDSTEP;

   JUMP A, STATUS, EQ, 12;

   INVOKE MYJCL;

$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.

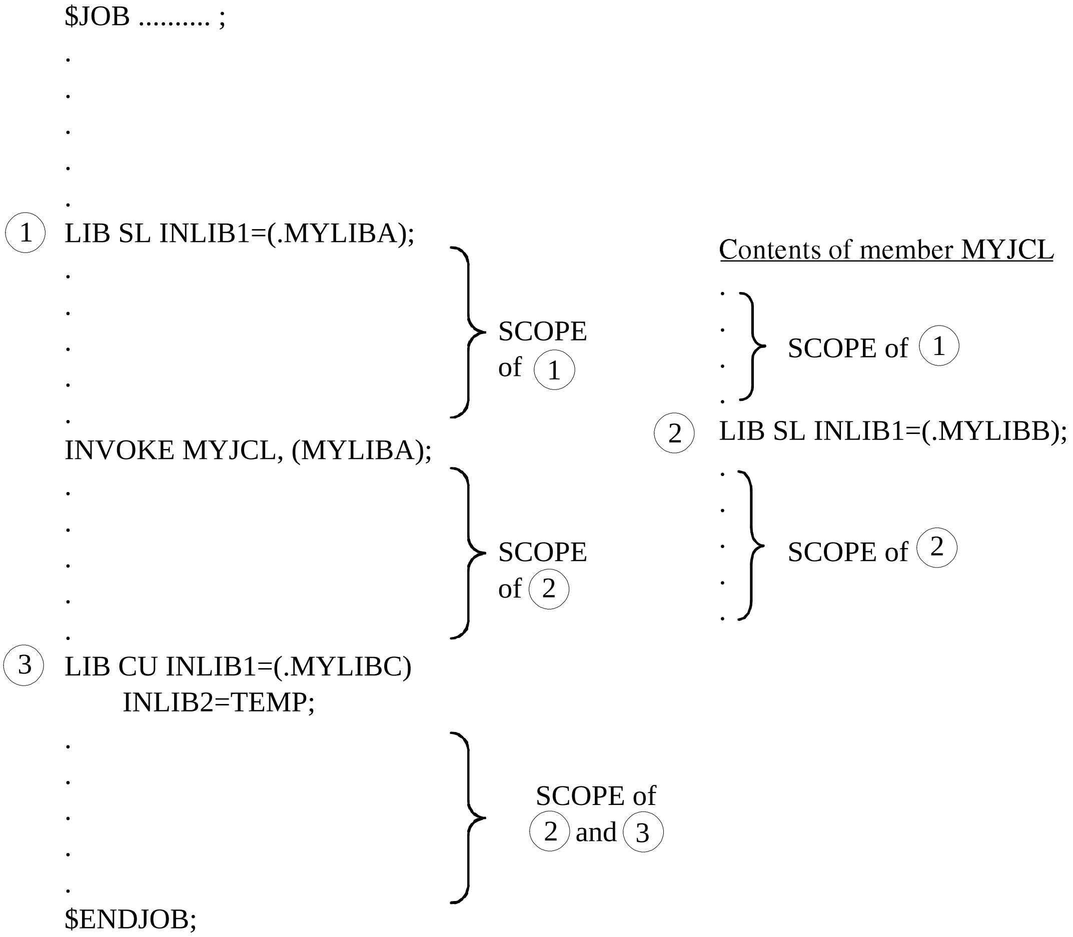
Independence of INVOKEd sequences also applies to parameter value substitution by means of the VALUES statement. If a VALUES statement appears in an INVOKEd sequence, it applies only to that JCL sequence.

LIB Statement

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. However, a LIB statement within the invoked sequence (i.e., sequence invoked via the INVOKE) is effective only within this sequence. Thus after the INVOKE statement, the last LIB statement of the main JCL sequence is again applicable to the main sequence.

A LIB statement in a main JCL sequence is not applicable to an executed JCL sequence (i.e., a sequence executed via the EXECUTE statement). A LIB statement within the executed sequence is effective only within this sequence (it is not applicable to the main sequence after the EXECUTE statement).

The effect of a LIB statement with INVOKE, EXECUTE, and $SWINPUT (described later in this section) is shown in the following examples.



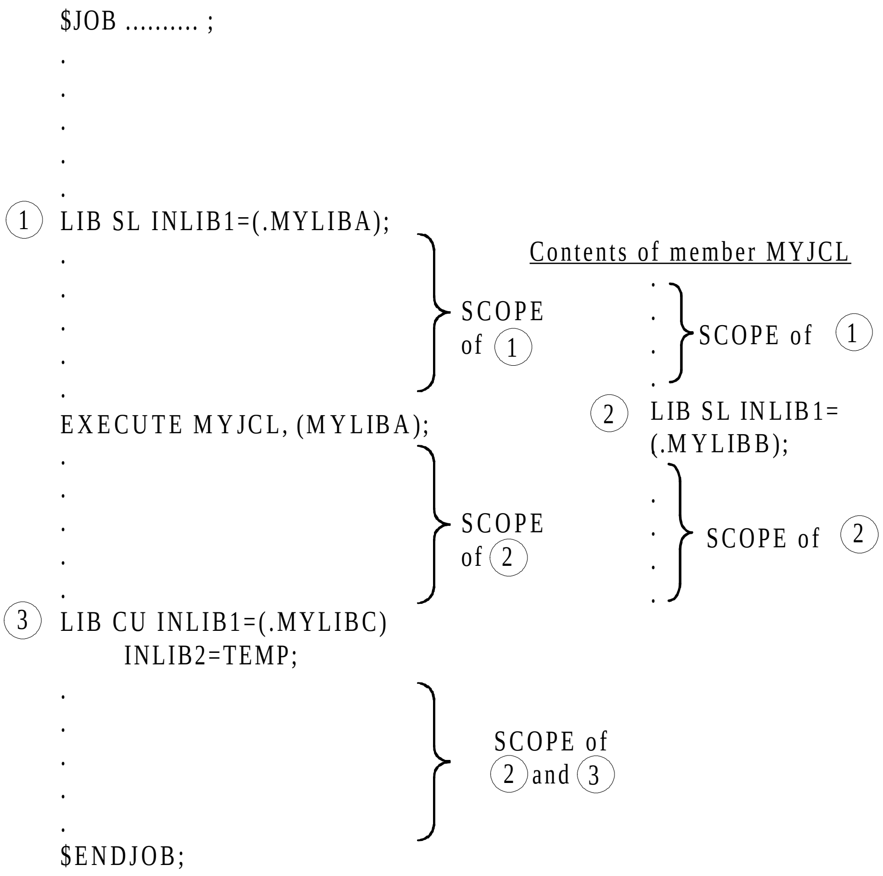
```
    $JOB .......... ;
    .
    .
    .
    .
    .
(1) LIB SL INLIB1=(.MYLIBA);
    .
    .
    .
    .
    .
    INVOKE MYJCL, (MYLIBA);
    .
    .
    .
    .
    .
(3) LIB CU INLIB1=(.MYLIBC)
           INLIB2=TEMP;
    .
    .
    .
    .
    .
    $ENDJOB;
```

Contents of member MYJCL

```
    .
    .
    .
    .
(2) LIB SL INLIB1=(.MYLIBB);
    .
    .
    .
    .
    .
```

The first LIB statement defines .MYLIBA as the input JCL library. The LIB statement of the invoked sequence overrides this, so that the input SL becomes .MYLIBB. However, the scope of this override is limited to the invoked sequence itself, so that the original input SL library .MYLIBA becomes effective again after the INVOKE statement.

The third LIB statement defines a CU library search path. As this LIB statement is for a different type of library (namely CU), it does not override the LIB SL statement and .MYLIBA remains the effective input SL library.

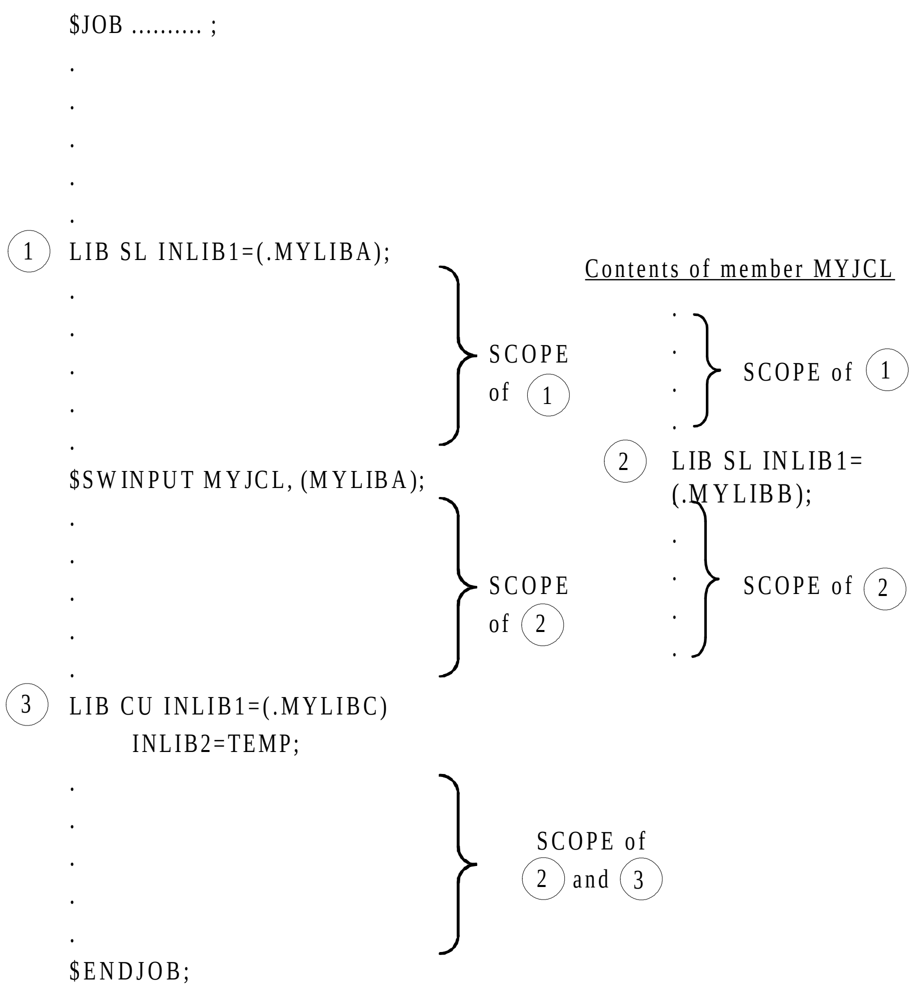


```
     $JOB .......... ;
     .
     .
     .
     .
     .
①   LIB SL INLIB1=(.MYLIBA);
     .
     .
     .
     .
     .
     EXECUTE MYJCL, (MYLIBA);
     .
     .
     .
     .
     .
③   LIB CU INLIB1=(.MYLIBC)
            INLIB2=TEMP;
     .
     .
     .
     .
     .
     $ENDJOB;
```

Contents of member MYJCL

```
     .
     .
     .
     .
②   LIB SL INLIB1=
     (.MYLIBB);
     .
     .
     .
     .
```

This example is similar to the previous example, except that the INVOKE has been replaced by an EXECUTE. Note that the LIB statement of the main sequence is not applicable within the sequence MYJCL. The LIB statement within MYJCL has no effect in the main sequence, so LIB statement no 1 is again effective after the EXECUTE statement.



```
$JOB .......... ;
.
.
.
.
.
(1) LIB SL INLIB1=(.MYLIBA);
.
.
.
.
.
$SWINPUT MYJCL, (MYLIBA);
.
.
.
.
.
(3) LIB CU INLIB1=(.MYLIBC)
         INLIB2=TEMP;
.
.
.
.
.
$ENDJOB;
```

Contents of member MYJCL

```
.
.
.
.
(2) LIB SL INLIB1=
    (.MYLIBB);
.
.
.
.
```

This example is similar to the two previous examples, except that the contents of MYJCL are "inserted" into the main sequence using the $SWINPUT statement. The $SWINPUT statement is processed by the Stream Reader, which logically replaces the $SWINPUT by the contents of MYJCL. The JCL Translator processes the sequence as if it were a single main sequence. Consequently, the LIB statements take effect as they would if they were all in the main sequence, i.e., 1 is overridden by 2, which remains effective until another LIB SL is encountered.

5.4.3 Nested INVOKE Statements

A stored sequence may itself contain INVOKE and/or EXECUTE statements. These statements can in turn 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 be nested only up to a depth of nine levels. Any INVOKE statement in a stored sequence that is referred to by an 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.

5.4.4 Invoking or Executing Input Enclosures

Sequences of JCL statements can be INVOKEd or EXECUTEd from input enclosures, rather than from stored files. You 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;
```

❑

5.4.5 Difference 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 an 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 or 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 so that the translation of other jobs will not be delayed. Therefore, if you reference a non-resident library via an INVOKE statement, you 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 in the note below.

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 the section "*Sequence Modification and Error Processing*" for a discussion of this technique.

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

1. whether the stored JCL statements are on a resident disk, or a non-resident disk which is not mounted;
2. whether the insertion of JCL can be static or must be done dynamically;
3. whether you wish to have all JCL errors detected before any step is started, or at job execution time.

NOTE:

Use of INVOKE with Non-Resident Libraries

As mentioned above, an 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 an EXECUTE statement that refers to the input enclosure or library member containing the INVOKE.

FOR EXAMPLE:

```
LIB SL, INLIB1 = (TOOLS.SLLIB, DEVCLASS = MS/D500, MEDIA = TEAM15)

INLIB2 = (TEST.SLLIB, DEVCLASS = MS/D500, MEDIA = EXP42);

...

EXECUTE *MY – ENC;
```

At step-initiation of the EXECUTE statement, as the libraries TOOLS.SLLIB and TEST.SLLIB are assigned to the step, the operator is requested to mount the volumes TEAM15 and EXP42. Then, if the input enclosure MY-ENC is as follows:

```
$INPUT MY-ENC;

INVOKE MY-WORK, (TOOLS.SLLIB, DEVCLASS = MS/D500, MEDIA = TEAM15);

INVOKE TEST-WORK, (TEST.SLLIB, DEVCLASS = MS/D500, MEDIA = EXP42);


$ENDINPUT;
```

the statements will be translated successfully, because the volumes TEAM15 and EXP42 are already mounted.

❑

5.4.6 $SWINPUT Statement

The $SWINPUT statement can be used to switch the input stream reading from the current stream (or file) to another named file. The file named must be available at Stream Reader time (i.e., must not need volume mounting or other resources). The CONSOLE option of $SWINPUT allows stream input to come from the console (the console of the submitter if the submitter is a station operator, or the IOF console if the submitter is an IOF user - if the console is not logged on, the job aborts). This facility should be used with care, especially in larger installations, as the Stream Reader is halted while it awaits operator input.

In the examples below, INFILE references files cataloged in the SITE.CATALOG.

EXAMPLE 1:

```
$JOB JOBNO-1, USER = XYZ, PROJECT = SW1;

...

...
$SWINPUT INFILE = SW1.FIL1;

$SWINPUT MEMN01, INFILE = SW1.LIB;

ASSIGN INT,

$SWINPUT CONSOLE = 'GIVE PRSONNEL FILE NAME'


ANSWERS = ('SW1.HOME','SW1.OVERSEAS', 'SW1.LOCAL','SW1.OTHER');

CATALOG = 2;

...

...

...

...

$ENDJOB;
```

The contents of the file SW1.FIL1 and of the member MEMN01 of the library SW1.LIB are inserted in the job stream in place of the first two $SWINPUT statements above. The contents of both of these files would be JCL statements needed in the job description. Both files are processed through to end-of-file condition.

The third $SWINPUT statement is embedded in a JCL statement and refers to a console. The message GIVE PERSONNEL FILE NAME will appear on the console and the operator will be given 3 attempts to specify one of the 4 valid replies listed in ANSWERS. Failure to supply a valid reply in 3 attempts will lead to job abortion. If the operator replies SW1.PERS then the ASSIGN statement becomes ASSIGN INT, SW1.PERS, CATALOG = 2;

EXAMPLE 2:

```
$JOB JOB-3, USER = MIKE, PROJECT = SW1;

STEP PROG1, .LIB;

$SWINPUT INFILE = .JCL1;

$INPUT INDATA2 CKSTAT;

XXXXXXXX }
........ }
........ } data A
........ }
XXXXXXXX }

$SWINPUT SUB1, INLIB = .LIB;

XXXXXXXX }
........ }
........ } data B
........ }
XXXXXXXX }

$ENDINPUT ;
$ENDJOB ;
```

Logically, this job is equivalent to the following,

```
JOB JOB-3, USER = MIKE, PROJECT = SW1 ;

STE PROG1, SW1.LIB ;

XXXXXXXX }
XXXXXXXX } contents of file SW1.JCL1
XXXXXXXX }

$INPUT INDATA2 CKSTAT ;

XXXXXXXX }
XXXXXXXX } data A
XXXXXXXX }

XXXXXXXX }
XXXXXXXX } contents of subfile of library SW1.LIB
XXXXXXXX }

XXXXXXXX }
XXXXXXXX } data B
XXXXXXXX }

$ENDINPUT ;
$ENDJOB ;
```

The following points should be noted

(i) The CKSTAT (Check for Stream Reader Statement) parameter must be specified on the $INPUT statement for the input-enclosure INDATA2. This parameter advises the Stream Reader that one or more $SWINPUT statements may appear within the input-enclosure data (i.e., any $SWINPUT statements are recognized and acted on). If CKSTAT is omitted, then $SWINPUT statements in the input enclosure are treated as data (i.e., part of the input enclosure and not interpreted as $SWINPUT statements). Note that for processing efficiency, if there is no $SWINPUT statement in the input enclosure, CKSTAT should be omitted.

(ii) Within the limit of 10 levels of nesting, any number of $SWINPUT statements may appear in a job stream. It is your responsibility to ensure that the final job stream, after all the input switching has taken place, is a valid job.

EXAMPLE 3:

```
$DATA MEMBER-1, SW1.LIB, END = 'FINISH', CKSTAT ;

XXXXXXXX }
........ }
........ } data A
........ }
XXXXXXXX }

$SWINPUT SW1.FILE4 ;

XXXXXXXX }
........ }
........ } data B
........ }
XXXXXXXX }

FINISH
```

The data loaded into MEMBER-1 of the library SW1.LIB will be:

data A
contents of file SW1.FILE4
data B

The files SW1.LIB and SW1.FILE4 must be available at Stream Reader time.

CKSTAT is necessary as a $SWINPUT statement appears in the data to be processed by $DATA.

EXAMPLE 4:

```
$SWINPUT CONSOLE = 'FILE NAME?', ANSWERS = ('A.B', 'A.C', 'A.D');
```

The message FILE NAME? will appear on the console. Valid operator replies are A.B, A.C, or A.D. If none of these replies is given in three attempts, the job aborts.

EXAMPLE 5:

```
$SWINPUT CONSOLE = ('DATA?','NEXT LINE'), END = 'FINISH';
```

The message DATA? is sent to the console to solicit the first line of data. After each reply the message NEXT LINE is sent to request another line of data. This process is repeated until the operator gives the reply FINISH. If the string 'NEXT LINE' is omitted from the $SWINPUT statement then the message DATA? would be used to solicit the second and subsequent lines of data.

❑

5.5 JCL Parameter Setting

5.5.1 Principles of Parameter Setting

There is often a need to use JCL sequences which differ from each other only in the value of certain parameters (ifn, efn, media-name, etc.) with each of these parameters likely to be found at several locations in the same sequence. JCL parameter setting does away with the tedious job of having to duplicate "n" number of times the same JCL, changing only these parameters.

The principle is simple.

- The parameter values are replaced in the JCL by "parameter references" (consisting of the symbol & followed by a maximum of 8 letters or digits).
- The values of these parameters are set using certain JCL statements (VALUES, in most cases).
- At translation time, the JCL Translator substitutes the corresponding parameter value for each reference, and does so each time this reference is found in the JCL being translated.

In this way, depending on the use, only statement(s) that set the parameter values need be modified. The following basic example illustrates this principle:

```
VALUES SAVE-PARIS, SALES-PARIS, K116, 1602;
.
.
.
ASSIGN RESULT, &2, DVC = MS/D500, MEDIA = &3;
.
.
.
VOLSAVE VOLUME = (DVC = MS/D500, MEDIA = &3)

       OUTFILE = (&1, DVC = MT/T9, MEDIA = &4);

Is the equivalent of:

ASSIGN RESULT, SALES-PARIS, DVC = MS/D500, MEDIA = K116;
.
.
.
VOLSAVE VOLUME = (DVC = MS/D500, MEDIA = K116)

        OUTFILE = (SAVE-PARIS, DVC = MT/T9, MEDIA = 1602);
```

The &1 reference was replaced by SAVE-PARIS, which is the first value in the list given by VALUES. Similarly, &2 was replaced by SALES-PARIS, which is the second value in the list. And &3 was replaced by K116, at the two locations where it appears, by the third value in the list.

This subsection, detailing how JCL parameter setting works in GCOS, deals with four main topics.

1. The first topic consists of the precise syntax of the parameter references and how they are entered in the JCL text.
2. The second covers how, through the use of VALUES and MODVL statements, the parameter values are substituted for their references in a direct JCL stream; that is, directly entered without resorting to the RUN, INVOKE or EXECUTE statements or the OCL SJ command.
3. The third covers parameter setting within sequences called up via INVOKE or EXECUTE, as well as jobs initiated by RUN or the OCL SJ command.
4. The last topic is the special cases that arise when setting the Input Enclosures parameters.

5.5.2 JCL Parameter References

There are two types of parameter references: positional references and keyword references.

5.5.2.1 Positional References

A positional reference consists of the & character followed by a 1 or 2 digit number (maximum value 99).

EXAMPLES:

&11, &20, &03, &3

Note that the last two examples have exactly the same value. This reference is called positional because the value of the corresponding parameter is given by a positional parameter in the statement which defines the parameter value; the value of the number following the & character gives the position of this parameter in the definition statement. In the example given at the beginning of this subsection, all the parameter values are positional and the VALUES statement gives the parameter values in the form of a list of positional parameters.

❑

5.5.2.2 Keyword References

A keyword reference consists of the & character followed by a string of up to 8 letters or digits. The first character in the string must be a letter.

EXAMPLES:

&KEYWORD &X1234567 &A1 &A01

Note that the last two examples are not the same.

This reference is called a keyword reference because the value of the corresponding parameter is given by a keyword parameter in the statement that gives parameter value. The parameter whose reference is & < Keyword > is defined in the definition statement by < Keyword > = value.

Going back to the first example, by using keyword references for both MEDIA names we have:

```
VALUES SAVE-PARIS, SALES-PARIS
       DISK = K116 TAPE = 1602;

.
.

ASSIGN RESULT, &2, DVC = MS/D500, MEDIA = &DISK;
.
.
.

VOLSAVE VOLUME = (DVC = MS/D500, MEDIA = &DISK)
        OUTFILE = (&1, DVC = MT/T9, MEDIA = &TAPE);
```

The substitution of parameter values for their references gives the same results as in the first example.

CAUTION:

In the above example, it so happens that the positional parameters are replaced by positional references and that the keyword parameter values are replaced by keyword references. This is not a rule. The opposite could just as well have been done, or only keyword references could have been used:

```
VALUES DISK = K116 TAPE = 1602
       EFN1 = SALES-PARIS EFN2 = SAVE-PARIS;
.
.
.
ASSIGN RESULT, &EFN1, DVC = MS/D500, MEDIA = &DISK;

.
.
.

VOLSAVE VOLUME = (DVC = MS/D500, MEDIA = &DISK)
        OUTFILE = (&EFN2, DVC = MT/T9, MEDIA = &TAPE);
```

In addition, such a rule would be illogical, as will be seen in the following paragraphs where references may replace either part of a parameter or a set of parameters.

❑

5.5.2.3 Location of Parameter References in the JCL

A reference (either positional or keyword) can be found anywhere in a job description except in a Stream Reader Statement; (that is, statements which must be preceded by the $ sign) because they are used by the Stream Reader (therefore before translation and the substitution of parameter values for the references): $JOB, $ENDJOB, $INPUT, $ENDINPUT, $SWINPUT, $DATA and $ENDDATA.

This means that a parameter reference, within a job description may be written in the place of:

- A single syntactic unit: ifn, efn, devclass, media-name, member-name, etc. This is the case of all the references used in the above examples.
- A more or less complex expression corresponding to a set of parameters, or even to one or more full JCL statements or including the end of one statement and the beginning of the next one. The only requirement for this is that it be possible for the "JCL portion" replaced by the reference to be defined by a string of characters not exceeding a length of 128.
- A portion of a single syntactic unit: a name prefix or suffix, for example.

This freedom of use may lead to ambiguities in interpreting the written JCL text: for example, does the expression &12ND represent positional reference &1 followed by 2ND, or positional reference &12 followed by ND? (it cannot be a keyword reference because the character following the & sign is not a letter).

It is therefore necessary to be very familiar with the rules used by the JCL translator in order to interpret texts which follow the & sign and to know where the exact end of the reference is.

The JCL translator recognizes the end of a reference as soon as one of the following three conditions is met:

1. If it encounters a character which is not part of the set authorized by the syntax involved. In other words:

   for a positional reference (beginning with a digit): any non-numeric character (letter, space, ; : ? . etc.)

   for a keyword reference (beginning with a letter): any non-numeric or non-alphabetic character (space, ; : ? . etc.).

   Example: in the expression

   ```
   (&NAME.FILEA, DVC = MT/&1/&2, MEDIA = &3TAPE)
   ```

   The keyword parameter &NAME is delimited by the . and positional parameters &1, &2 and &3 are delimited by the characters /, and T respectively.

2. If it reaches the limit number of characters for the type of reference involved without having encountered condition A above or condition C below (2 digits for a positional reference or 8 letters or digits for a keyword reference).

   Example:

   &1234 is interpreted as being reference &12 followed by 34.

   &KEYWORD12A is interpreted as being reference &KEYWORD1 followed by 2A.

3. If it encounters the symbol ||, two vertical lines, which are deleted when the parameter value is substituted for its reference.

   Example:

   &1||234 is interpreted as being reference &1 followed by 234.

   &KEYWORD||12A is interpreted as being reference &KEYWORD followed by 12A (and not &KEYWORD1 followed by 2A, as in B, above).

   Practically speaking, the following two rules should be observed:

   a. Parameter references can be placed anywhere in the JCL except in Stream Reader statements.
   b. The symbol || (!! in IOF) is used to delimit references (C above) each time the reference is not directly followed by a character authorized in the syntax of a reference (A above) and it has not reached maximum length (B above). (Most of the time, condition A occurs).

NOTE:

For reasons concerning the structure of the JCL translator analyzer, the double vertical line is also required in the special case when a parameter reference ends a job description on the last positions of a record preceding $ENDJOB.

In the remainder of this document, the distinction will no longer be made between positional and keyword references; only parameter references will be referred to without further precision because the procedure involved in substituting parameter values for their references is the same in both cases. The only differences lie in the way in which the values are defined in the definition statements for these values and in the way in which the JCL translator analyzes their references in the text to be translated. All this has been covered above. Thus, in order to reduce lengthy examples, only positional references will be used.

5.5.3 Procedure Involved in Substituting Parameter Values for References

5.5.3.1 How VALUES and MODVL Work in the "DIRECT STREAM"

A "direct stream", as used in our discussion, is a JCL stream entered directly, that is, without resorting to INVOKE, EXECUTE, or RUN statements or to the OCL SJ command.

The parameter setting procedure can be broken down into two steps:

- setting up a table giving the parameter values to substitute for the corresponding references.
- substituting these values in the JCL text.

The first step is more or less complex depending on whether the VALUES statement alone is used or whether both the VALUES and MODVL statements are used, and depending on whether this is done in a direct stream or in a sequence called in or initiated by INVOKE, EXECUTE, RUN, or SJ in combination with the parameter values that these statements may provide themselves.

The second step is the same in both cases.

Going from simple to complex, the discussion will begin with an explanation of VALUES in direct stream. This will also allow us to explain in detail the second step; i.e., substituting the values for references in the JCL. The following paragraphs will cover streams called in or initiated by INVOKE, EXECUTE, RUN or SJ.

5.5.3.2 The VALUES Statement

Value Definitions in the VALUES statement

The VALUES statement is the easiest way to set up a table of values to be substituted for references. It uses positional parameters to define the values to be substituted for positional references and keyword parameters for the values to be substituted for keyword references.

This values table applies to all statements which follow VALUES until a MODVL statement is encountered which modifies it, or another VALUES statement is encountered which cancels it and defines a completely new table, or until the $ENDJOB statement is encountered.

The values as they are found in the VALUES statement can be expressed in one of the following ways:

1. A string of characters observing the syntax of the single syntactic unit (or of the portion of a single syntactic unit corresponding to the parameter referred to: an efn, a devclass, a media-name, etc.

EXAMPLE:

```
CUSTOMER.FILE   T7    1610

in (&1, DVC = MT/&2/D1600, MEDIA = &3)
```

2. A protected string of characters (in quotes) having a maximum length of 128 characters. (If this protected string also contains a protected string, the latter must be placed in double quotes.) This protected string can correspond in the JCL to very different types of referenced units:

   Protected strings: option string, commands for a processor in the parameter COMMAND = 'command [command] ...'

   Single syntactic units that may be protected strings:
   efn, media-name

   More or less complex portions of JCL that may overlap two consecutive statements or even contain a full statement.

❑

These different possibilities result in a substitution procedure, which, in the case of protected strings, may seem complex if unfamiliar, but which is in reality simple once it becomes familiar. It is important to be very familiar with it if mistakes are to be avoided. This is covered in the next paragraph.

Substituting Parameter Values for References

The basic example given in the introduction shows how parameter values are substituted for references, but it is restricted to the case in which references involve a single syntactic unit or in which values defined by VALUES are unprotected strings.

Let us examine the general case. When the JCL translator-analyzer encounters a parameter reference, it expects to find a well-defined type of unit: a label, an efn, a keyword, a member-name, etc. (or a portion of one of these single syntactic units).

1. The value found in the table of values is not in quotes. The JCL translator simply substitutes this value for the reference and checks that the result meets the requirements of the syntax of the expected syntactic unit (see the example at the beginning of this subsection).
2. The value found is a protected string. This again breaks down into two conditions:
    a. The expected single syntactic unit is (or may be) a protected string: option string, processor commands in COMMAND =..., media-name. Here again the JCL translator simply substitutes, but no analysis is made as to the contents of the protected string; it just checks that its length does not exceed the authorized length for the unit involved (6 characters for a media-name, for example).

    b. The expected single syntactic unit can under no circumstances be a protected string (a keyword, a decimal number, a devclass, an SIV, etc). This means that the reference has been placed in the text as a replacement for a more or less complex JCL portion and that the protected string found in the table of parameter values contains this JCL portion. In this case, the JCL translator substitutes this string (except the quotes) for the reference and continues analyzing the text beginning with the string which was substituted.

    This is the substitution algorithm in general. It implies the following rule: when, during a JCL parameter setting operation, an entire JCL portion is to be replaced by a single reference, this JCL portion must not begin with a single syntactic unit which is always or may possibly be a protected string. If this rule is not observed, the JCL translator assumes it is in case b1, whereas case b2 was anticipated: it will take the entire string for the value of the beginning single syntactic unit.

Here are several examples to illustrate this rule.

EXAMPLE 1:

```
VALUES 'A + B', MT/T9, 1601;
.
.
.

ASSIGN IFN, &1, DVC = &2, MEDIA = &3;

Is the equivalent of:

ASSIGN IFN, 'A + B', DVC = MT/T9, MEDIA = 1601;
```

EXAMPLE 2:

```
VALUES CUSTOMER.FILE, 'DVC = MS/D500, MEDIA = K116';
.
.
.
ASSIGN IFN, &1, &2;

Is the equivalent of:

ASSIGN IFN, CUSTOMER-FILE, DVC = MS/D500, MEDIA = K116;

For the parameter referenced by &2, case b2 is involved.
```

EXAMPLE 3:

```
VALUES 'CUSTOMER.FILE, DVC = MS/D500, MEDIA = K116';
.
.
.
ASSIGN IFN, &1;

Is the equivalent of:

ASSIGN IFN, 'CUSTOMER.FILE, DVC = MS/D500, MEDIA = K116';

And not

ASSIGN IFN, CUSTOMER.FILE, DVC = MS/D500, MEDIA = K116;
```

Case b1 and not b2, is involved. The JCL translator took the entire protected string for an efn. And if the second ASSIGN was expected, it was because the above rule was not observed.

EXAMPLE 4:

For the same reason MEDIA = &1 with a &1 value defined by 'K116, DEVCLASS = MSD500' does not result in

```
MEDIA = K116, DEVCLASS = MS/D500
```

The translation aborts with the fatal message MEDIA: TOO LONG STRING. Actually, since a media-name may be a protected string, case b1 is involved and the JCL translator takes the entire string for the media-name, but this is obviously too long (6 characters maximum for a media-name).

For the same reason, a media-list (even in parentheses) cannot be considered as a single parameter. Its value should be defined in quotes because of the commas or blanks separating the various media-names, or the JCL translator would take the list for a single media-name.

EXAMPLE 5:

On the other hand, DVC = &1 can perfectly well be written with the &1 value defined by 'MS/D500, MEDIA = K116'. Case b2 is involved because a devclass cannot be a protected string. The following will be the result of the substitution after the quotes are removed:

```
DVC = MS/D500, MEDIA = K116
```

❑

NOTES:

1. It is possible for the value of a parameter not to be defined in the table of values.

   ```
   VALUES A,, C; or VALUES A # C;
   ```

   The value corresponding to the &2 reference is not defined. This is the same as an empty string (not to be confused with a blank): what precedes &2 is then directly concatenated with what follows.

   ```
   J.&2SMITH results in J.SMITH and not J. SMITH
   ```

2. The presence of a reference as the value of a parameter in a VALUES statement is the equivalent of #. Caution: this is true in a direct stream, but no longer in the JCL called in by INVOKE, EXECUTE, RUN, or SJ.

   ```
   VALUES   A, &2, C; or VALUES   A, &KEY, C;
   ```

   gives the same result as VALUES A,, C;

3. When case b2 is involved and the string which was just substituted for the reference is being analyzed, a reference can perfectly well be found once again; the substitution algorithm is then applied to this reference in the same way it would have been if the reference had been found in the text before any substitution took place. In particular, the string which was just substituted may very well be just a reference:

   ```
   VALUES A, '&3', MS/D500, K116;
   .
   .
   .

   ASSIGN IFN1, &1, DVC = &2, MEDIA = &4;
   ASSIGN IFN2, &2, DVC = &3, MEDIA = &4;

   Is the equivalent of;

   ASSIGN IFN1, A, DVC = MS/D500, MEDIA = K116;
   ASSIGN IFN2, '&3', DVC = MS/D500, MEDIA = K116;
   ```

   For the first ASSIGN, DVC = &2 has first become DVC = &3 then DVC = MS/D500. On the other hand, for the second ASSIGN, case b1 was involved; the &2 reference was changed for an efn which may be a protected string; therefore the value '&3' was taken for the efn value.

   Lastly, still in the same case, a second precaution must be taken when the reference ends the protected string which defines the value to be substituted: it must be followed by the double line || if the length of the reference is less than its permitted maximum length (<2 for a positional reference, <8 for a keyword reference). Here is an example to help explain:

   ```
     VALUES A, 'D&3', 50;
      As applied to DVC = MS/&2 | |0
   ```

   Firstly, the &2 reference (separated from the 2 which follows the 0 to clearly show that the reference is not &||) is replaced by the corresponding value, with the || removed, and as the expected unit cannot be a protected string, the quotes are removed and results in DVC = MS/D&30. The translator again analyzes from D, finds the &30 reference which is not defined, and the final result is DVC = MS/D.

On the other hand, if the following is written:

```
          VALUES A, 'D&3||', 50;
          The first step results in DVC = MS/D&3||0
```

and the second substitutes its value for &3 by removing the ||, which results in DVC = MS/D500.

4. In addition to efn and media-name, a third unit may be a protected string; it is member-name in the SOURCE = member-name of COBOL, FORTRAN, RPG, and PL1 compilation statements. In these statements, the full star-convention with FROM =... TO =... can be used.

EXAMPLE:

```
COBOL SOURCE = 'TOTO* FROM = TOTODD TO = TOTOPP'
      INLIB = INLIB1 CULIB = USER.CULIB;
```

The possibility of having such an expression in SOURCE = excludes the following writing which consists in taking a member-list for a parameter:

```
VALUES '(TOTO, TITI, TATA)';
COBOL SOURCE = &1;
```

After substitution, the SDS will attempt to find the expression of a star-convention, which does not exist. If the parameters of such a member-list are to be set, the same procedure used for a media-list must be used; that is, setting the parameter of each object separately:

```
VALUES TOTO, TITI, TATA;
COBOL SOURCE = (&1, &2, &3); works perfectly well.
```

This remark does not hold true for the other statements in which a member-name may appear. For example, in an INVOKE, the member-name and its library may be grouped together in the same parameter in quotes; case b2 is then involved and not b1 as above:

```
VALUES 'TESTVAL, USER.SLLIB;
INVOKE &1;
```

This is the equivalent of INVOKE TESTVAL, USER.SLLIB; INVOKE calls in the TESTVAL member of the USER.SLLIB library and not the 'TESTVAL, USER.SLLIB' of the SYS.SLLIB library, which would not make sense.

❑

5.6 The MODVL Statement

The MODVL statement is used to modify the table of values previously set up using a VALUES statement or already modified by another MODVL statement. Application of the modified table continues up to the next MODVL, VALUES, or $ENDJOB statement.

MODVL modifies the table of parameter values in 3 ways:

- by replacing certain values by others,
- by adding values of new parameters,
- by deleting others, which consists in replacing them with an empty string by indicating NIL for these values.

MODVL modifies the table of parameter values in 3 ways:

EXAMPLE:

```
VALUES A, B, C, D;
MODVL ,1, NIL,, 3, KW1 = AB;
```

After MODVL:

- the value corresponding to the &1 reference is not modified. It is A.
- the value corresponding to the &2 reference is no longer B. It is 1.
- the value corresponding to the &3 reference is no longer C. It is "empty".
- the value corresponding to the &4 reference is not modified. It is D.
- the value 3 corresponding to the &5 reference is added.
- the value AB corresponding to the &KW1 reference is added.

The final table of values to be substituted after MODVL is therefore

```
&1     &2     &3   &4      &5      &KW1
A      1    empty    D      3       AB
```

❑

NOTE:

One very important remark must be added for an understanding of the MODVL mechanism. The presence of a parameter reference as a value in MODVL, is not, as it is in VALUES, equivalent to no value; it simply means that in order to modify the table, the value of the corresponding reference is taken from the table itself at the time the modification is made relative to the parameter involved. It must be clearly understood that this is a dynamic process. The various table values are processed in the order they are mentioned by MODVL from left to right. If the &3 reference corresponds to a value which is itself modified, calling in this reference in MODVL will not have the same effect depending on whether the value relative to &1 or the one relative to &5 is to be modified.

For example, let us apply the following MODVL statement to the table resulting from the above MODVL statement:

```
MODVL &4,, &1,,, KW2 = &KW1, KW1 = &4, KW3 = &KW1;
```

The resulting table will be as follows:

&1	&2	&3	&4	&5	&KW1	&KW2	&KW3
D	1	D	D	3	D	AB	D

- &1 takes on the value of &4 of the initial table; i.e., D.
- &2, &4 and &5 are not modified,
- &3, being processed after &1, takes on the value of &1 of the already modified table; i.e., D (and not A, the value of the initial table).

&KW2 and &KW3, although both referring to &KW1, take on two different values, AB and D, respectively, since the first is created before the modification of KW1 and therefore takes its initial value AB, whereas the second is created after the modification of KW1 and therefore takes on its final value of D.

Remember that replacing table values by other values applies both to values that are not in quotes as well as to those that are. In addition, this process involves the replacement of values to be later substituted and not "substitution" as meant in the substitution algorithm explained above when speaking about VALUES. The substitution of values in the modified table in the JCL that follows MODVL, obviously takes place according to the algorithm explained for VALUES.

Note that if MODVL has deleted a value (using NIL) a further reference to this parameter has no effect. If such a reference is used in a further MODVL statement the value you want to modify is not modified i.e., NIL is not a value that you can transfer from one MODVL to another.

5.6.1 Parameter Setting in a Sequence Called or Initiated by INVOKE, EXECUTE, RUN, or SJ

5.6.1.1 Setting the Parameters

When setting the parameters for the called-in sequence or the launched job, these JCL statements and the SJ command behave in the same way. In particular, they can give the values of the parameters to be set within the called-in sequence, owing to the VALUES = (...) keyword for JCL statements. Therefore, in the following paragraphs, only sequences called in by INVOKE will be covered, since the same process is involved for EXECUTE, RUN, and SJ, but at slightly different times.

The problem that arises involves setting up the table of values to be substituted for references, since as far as substitution itself is concerned, the process is the same.

First, remember that in the expression VALUES = (...) (by which INVOKE provides the values to be substituted for references in the called-in sequence), parameter references may be found in the place of values. The first step consists of replacing these references with the corresponding values in the calling-in sequence.

```
VALUES A B C D;
.
.
.
INVOKE SEQ, VALUES = (&3, 2, 3, 4, &1);
Is equivalent to INVOKE SEQ, VALUES = (C, 2, 3, 4, A);
```

5.6.1.2 The Principle of Operation

By using the keyword VALUES = (...), INVOKE provides the called-in sequence with the parameter values that must be applied. These values are called external (relative to the sequence). But it can happen that VALUES = (...) does not provide values for every referenced parameter in the called-in sequence. In this case, the values defined by the VALUES and MODVL statements of the called-in sequence are used by default in exactly the same way as if this sequence were used in "direct stream": this is the reason why these values are called default values.

FOR EXAMPLE:

```
INVOKE SEQ, VL = (# B # D);

      With SEQ: VALUES 1, 2, 3, 4;
                  .
                  .
                  .
The external values are empty   B  empty  D

The default values are   1     2    3    4
```

The table of current values that are substituted for the references in the remainder of the sequence is:

1 B 3 D

The following is a detailed explanation of how the table of default values is set up and evolves as a function of the VALUES and MODVL statements encountered in the called-in sequence.

The VALUES statements in the sequence called in by INVOKE may themselves contain parameter references. Contrary to what happens in "direct stream" (and concerning this, the general operating principle described above needs to be explained), these references are not equivalent to "empty" but they are given the corresponding external value (which may be "empty") in order to make up the table of default values which will be valid after the VALUES statement involved.

❑

EXAMPLE:

```
INVOKE SEQ,    VL = (# B # D);
With  SEQ:         VALUES &2, &1,   3, 4;

Gives default values:     B  empty  3  4
      and current values  B    B    3  4
```

These are the values that are applied to the called-in sequence up to the next VALUES or MODVL statement or the end of sequence.

MODVL applies (dynamically as well as in direct stream) to the table of default values (and not to the current values).

Continuing with the above example, and supposing that further on in the called-in sequence the following statement is found:

MODVL	1, , C ;
the table of default values becomes:	1 empty C 4
and the table of current values:	1 B C D

But the MODVL statement itself may contain parameter references for parameter values. The general rule applies, taking into account the dynamic operation of the MODVL statement: this reference is replaced by the corresponding external value if it exists, otherwise by the corresponding default value (as it figures in the table at this time).

❑

FOR EXAMPLE:

```
INVOKE SEQ, VL = (# # # D);

with SEQ:         VALUES &4 2 &1 4;
                   .
                (1).
                    MODVL  A   &4   &1
                 (2).
```

For all the statements in (1)

The default values are: D 2 empty 4

and the current values: D 2 empty D

Then MODVL modifies the table of default values:

- A first replaces D as the value of the first parameter, which gives the following default table:

  A 2 empty 4

- The second MODVL value is actually the &4 reference which is first replaced by the corresponding external value D, which is the value that is substituted for the second value of the default table, which now becomes:

  A D empty 4

- The third MODVL value is the &1 reference for which there is no corresponding external value. Therefore, the default of the table as it exists at this time (i.e., A) is the one that is taken to modify the table itself, which then becomes:

  A D A 4

As the fourth parameter is not modified by MODVL; the final default value table is:

A D A 4

And for all the statements in (2), the table of current values, which will be substituted for the parameter references, is:

A D A D.

There is no special problem if the INVOKE statements are nested. The current values of the sequence called in by the first INVOKE statement are taken to complete the VALUES = (...) of the INVOKE statement internal to this sequence. From this point on, the external values to be used for the sequence called in by this second INVOKE statement are available and the same procedure takes place to get the values to be used in this second sequence.

❑

5.6.2 Input Enclosure Parameter Setting

It is also possible to have parameter references inside an Input enclosure (which may be, for example, the commands for a utility program initiated by a step of the job in progress). The environment in which the parameters are set is characterized as follows:

An Input enclosure can be found only at the job level; it cannot be found inside a sequence called in by INVOKE or EXECUTE (but a job initiated by RUN or SJ is an actual job and can therefore contain Input enclosures).
This Input enclosure can be ASSIGNed several times during the same job, either at the job level or at the level of a sequence called in by INVOKE or EXECUTE.

The sequence of events is as follows:

- At the time the Stream Reader reads a JOB, it stores a version (called version 0) of the Input enclosure in SYS.IN in which the parameter references figure as such.
- Then at translation time, each time the Input enclosure is ASSIGNed, either at the job level or at the level of a sequence called in by INVOKE, the JCL translator makes a new version of the Input Enclosure in SYS.IN. In this version, the appropriate values of the parameters have been substituted for their references.
- At the end of translation, version 0 of the Input enclosure is saved in the SYS.IN for further possible assignment during a sequence called in by an EXECUTE statement contained in the job.

As with the parameter setting of the JCL itself, the input enclosure setting procedure poses two types of problems:

1. The choice of values to be substituted for the references,
2. The actual substitution of these values for the references.

The second point will be covered first, since it is simpler.

5.6.2.1 Substitution Algorithm

The substitution algorithm is exactly the same as for the JCL statements.

EXAMPLE:

```
Consider &1 corresponding to '&2MAIN'
         &2 corresponding to H_TN_EP
and the command PRINT &1; (LIBMAINT command).
```

The first step consists in converting it into

```
PRINT &2MAIN;
```

The second step results in the final version

```
PRINT H_TH_EPMAIN;
```

Two minor restrictions should be noted as compared with operation with the JCL statements:

1. A given reference cannot overlap two consecutive records as it may with two consecutive JCL statements,
2. If a record ends with a reference, this reference must be followed by the symbol || (in JCL this was only required for the last statement in a job, just before $ENDJOB).

❑

5.6.2.2 Determining the VALUES to be Substituted

So that parameter setting operates correctly, one of the two SIVs, JVALUES or CVALUES (i.e., job-level VALUES and current VALUES), must be used in the $INPUT statement. If none of these two SIVs is used, a new version of the Input enclosure is not created; version 0, containing the parameter reference in the initial state, is the one that is ASSIGNed.

Let us now examine Input enclosure parameter setting with JVALUES and CVALUES, depending on whether the Input enclosure ASSIGN is:

- at the job level,
- at the level of a sequence called in by INVOKE,
- or at the level of a sequence called in by EXECUTE.

5.6.2.3 At Job Level

Both SIVs (JVALUES and CVALUES) behave in the same way. The table of values to be used is the one that is valid at the time ASSIGN occurs in the particular job. The table is the one resulting from the set of VALUES and MODVL statements in a "direct stream" and which, for a job called in by RUN or SJ, consists of the external values provided by RUN or SJ, possibly completed by the default values determined by the set of VALUES and MODVL statements at the time the ASSIGN occurs.

5.6.2.4 At INVOKE Level

The JVALUES and CVALUES statements behave differently.

With JVALUES, all the ASSIGNs within the INVOKE statement (or within nested INVOKEs) trigger parameter setting using the table of values which would have been valid for an ASSIGN located in the place of the first INVOKE.

With CVALUES, the table of current values as it is valid at this time within the sequence called in by INVOKE is used.

5.6.2.5 At EXECUTE Level

Contrary to INVOKE, which is processed by the translator, EXECUTE corresponds to a step which, at the time of job execution, calls the translator in again to translate the designated sequence.

JVALUES and CVALUES statements behave in the same way. They both use the table consisting of the external values provided by EXECUTE, possibly completed by the default values at the time the ASSIGN occurs. The sequence of events is the same as at the job level for a sequence called in by RUN.

The following job structure illustrates all the cases that may arise:

```
$JOB...;

VALUES A;
.
.
.
ASSIGN IFNA, *EXAMPLE;
.
.
.
INVOKE SEQ1, VL = (1); with SEQ1: ASSIGN IFN1, *EXAMPLE;
.
.
.
VALUES B;
.
.
.
ASSIGN IFNB, *EXAMPLE;
.
.
.
EXECUTE SEQ2, VL = (2); with SEQ2: ASSIGN IFN2, *EXAMPLE;

VALUES C;
.
.
.
$INPUT EXAMPLE, CVALUES or JVALUES;
.
..

$ENDINPUT;
$ENDJOB;
```

In this example, it is assumed that the set of values passed by the INVOKE or EXECUTE statement is complete and that therefore the possible default values in the called-in sequences are irrelevant. If this were not the case, it would be easy to correct. Accordingly, the following conclusions can be made:

- The ASSIGN IFNA concerns an Input enclosure whose parameters have been set using the set of values A whatever the SIV (JVALUES or CVALUES) in the $INPUT may be. Similarly, the ASSIGN IFNB concerns an Input enclosure whose parameters have been set using the set of values B.
- The ASSIGN IFN1 concerns an Input enclosure whose parameters have been set using:
  - the set of values 1 for $INPUT...., CVALUES. (The current values are those given by the INVOKE statement.)
  - the set of values A for $INPUT..., JVALUES. (If at the JOB level, there were an ASSIGN IFN1 in the place of the INVOKE, the set of values A would be the ones used.)
- The ASSIGN IFN2 concerns an Input enclosure whose parameters have been set using the set of values 2, whether there is a CVALUES or JVALUES in the $INPUT.

5.6.2.6 Parameter Setting of Part of an Input Enclosure

Parameter setting may be used for only a portion of an input enclosure. The portion(s) of the Input enclosure for which the parameters should not be set (i.e., for which the & symbol must be left as is), must be preceded by // BOD and followed by // EOD which serve as protection. This is especially useful in Input enclosures using parameter setting and containing LIBMAINT commands including EDIT with requests (substitution, concatenation) in which the & symbol has a completely different meaning than that of introducing a parameter reference. The portions of the Input enclosure containing such requests must be preceded by // BOD and followed by //EOD.

EXAMPLE:

```
$JOB   TEST  USER = X  PROJECT = Y ;
 VALUES  TN_EPMAIN ;
 LIBMAINT SL LIB = SL.LIB COMFILE = *IN ;
$INPUT   IN  JVALUES ;
 PRINT &1 ;
 EDIT ;
 R&1
 //BOD
 ^,$ GP/TN/
 ^,$ S/TN/&_EP/
 ^,$ GP/TN/
 //EOD
 Q
$ENDINPUT ;
$ENDJOB ;
```

is equivalent to:

```
$JOB   TEST  USER = X  PROJECT = Y ;
 LIBMAINT SL LIB = SL.LIB COMFILE = *IN ;
$INPUT   IN ;
 PRINT TN_EPMAIN ;
 EDIT ;
 R TN_EPMAIN
 ^,$ GP/TN/
 ^,$ S/TN/&_EP/
 ^,$ GP/TN/
 Q
$ENDINPUT ;
$ENDJOB ;
```

Input enclosures may be used in the place of a library-member to contain the JCL called in by INVOKE or EXECUTE and the Input enclosure which may be referenced by the INVOKE UPDATE parameter and contains the mini-editor commands.

None of those Input enclosures can have their parameters set as discussed in this subsection. That is, the introductory $INPUT must not contain a SIV JVALUES or a SIV CVALUES keyword.

❑

5.7 Job Stream Creation

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 sequential disk or tape file
- a source library member

Furthermore, the operator can ask the Stream Reader to select only certain jobs of the job stream.

A job stream can be stored on a sequential disk or tape using the CREATE statement as in the following example:

```
$JOB...;
 CREATE  INFILE = (STREAM, MEDIA =          OUTFILE = (STREAM,
MEDIA = DD345, DEVCLASS =
                     MT/T7);
$ENDJOB;
```

The same function is also provided via LIBMAINT:

```
$JOB...;
   LIBMAINT SL,
     INFILE = (STREAM, MEDIA = ,
     OUTFILE = (STREAM, MEDIA = DD345, DEVCLASS = MT/T7),
     COMMAND = ‘MOVE INFILE: KARDS, 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 CDOi MOUNT KARDS FOR Xj.
```

where CDOi 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: DD345: MT/T7
```

5.8 The Mini-Editor

It is sometimes desirable to INVOKE a sequence of JCL that is slightly different from an existing stored sequence (for example, to test modifications). The stored sequence may be modified using the editor and then tested in a subsequent job (using INVOKE). Alternatively, the sequence can be modified at JCL translation time (i.e., at the moment that the sequence is inserted in the job stream) without modifying the stored sequence. This has the advantage that the stored sequence is unchanged, which is preferable if the modified sequence is to be run just once or is found to be in error.

By using the UPDATE = *input-enclosure-name parameter of INVOKE, a sequence can be modified at JCL translation time without changing the sequence as stored. The input enclosure contains editor commands to effect the required modifications.

The editing commands available to the Mini-Editor user is the following subset of the LIBMAINT editing commands:

ad1	A - Creates a new line after the line number specified.
ad1 [,ad2]	C - Changes the whole of the specified lines for the text following the C.
ad1 [,ad2]	D - Deletes the lines specified.
ad1	I - Inserts new lines of text before the line number specified.
ad1 [,ad2]	S - Substitutes one character string for another on the specified line.

Use of the Mini-Editor commands requires the specification of the line number of each statement. This information can be obtained using the PRINT command of LIBMAINT. These line numbers are indicated by ad1 and ad2 in the commands above.

Note the following:

- only numerical line number addressing can be used
- ¬ and $ are not acceptable as line numbers
- ❷ is treated normally, but *,.,¬, and $ are not treated as special characters
- the commands must be submitted in increasing order by ad1.

For more details see the *Library Maintenance Reference Manual*.

6. Sequence Modification and Error Processing

6.1 Introduction

GCOS contains system components and utilities that are designed to minimize the effects of serious errors that 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. These facilities are 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).

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).

6.2 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 Directory* along with probable causes and remedial action where appropriate.

6.2.1 JCL Errors

Errors found by the JCL Translator result in error messages being printed in the JOR (Job Occurrence Report). These error messages are self-explanatory and usually appear just after the statement to which they apply. There are two types of message, WARNING and FATAL. WARNING messages usually supply information (e.g., standard action or default value taken) rather than signal errors and the statement concerned can subsequently be executed successfully. FATAL messages indicate serious errors that will prevent successful execution of the statement concerned and consequently the job is not executed. Some types of JCL error can lead to a message or sequence of messages that require further analysis to trace the error. Examples of these are given below.

EXAMPLE 1:

Missing closing quote mark in a protected string.

```
$JOB...;
...
...
COMM 'RECREATE MACHREF FROM BACKUP,;

STEP PROGA, LOD,LIB;
ASSIGN M5242, EMPLOYE.MST; ASSIGN M5243, FILE.A;
ASSIGN M5244, FILWORK, DEVCLASS=MS, MEDIA=RDISK2;
ALLOCATE M5244, SIZE=1, INCRSIZ=1, UNIT=CYL;
SEND 'LINE UP MACH PREF ERROR. ABORT.';

FATAL 211   INVALID KEYWORD: MACH
FATAL 211   PREF
FATAL 211   ERROR. ABORT.ENDSTEP;
FATAL 101   NO MATCHING BETWEEN STEP STATEMENT AND
            ENDSTEP STATEMENT
```

The error is a comma instead of a quote mark following BACKUP in the COMM statement. This results in all the statements (up to the next quote mark) following the COMM statement being treated as part of the protected string (i.e., part of the comment). The first quote mark of the SEND statement is taken as closing the string.

The result is several error messages which apply to the SEND and ENDSTEP statements even through there is in fact no error in either of these. The error is several statements earlier.

EXAMPLE 2:

Missing semicolon at the end of a statement

```
      $JOB ........;
       ....
       ....
       ASSIGN MY123, FILE.X

       ASSIGN MY124, FILE.Y;
FATAL  KEYWORD UNKNOWN.
       ASSIGN

FATAL  KEYWORD UNKNOWN.
MY124

FATAL  KEYWORD UNKNOWN.
       FILEY
       ....
       ....
```

The missing semicolon on the first ASSIGN statement causes the second ASSIGN statement to be treated as parameters of the first.

The error messages appear after the second ASSIGN statement even though there is in fact no error in this statement.

EXAMPLE 3:

Missing closing parenthesis.

```
           $JOB ........;
            ...
            ...
            ...
            ...
            STEP MYPROG, (MY.LIB, CATALOG=3, CPTIME=10,
            DUMP=DATA, REPEAT;

   FATAL    211 INVALID KEYWORD: CPTIME

   FATAL    211 INVALID KEYWORD: DUMP

   FATAL    211 INVALID KEYWORD: REPEAT

FATAL ... MISSING PARENTHESEIS
      ...
      ...
      ...
```

The missing closing parenthesis causes the last three parameters to be treated as part of the library file description. Even though the three parameters are valid parameters of the STEP statement, they are not valid parameters for a library file description. In this case the error messages immediately follow the statement in error; however, they are misleading at first sight.

EXAMPLE 4:

```
           Error in INVOKE library
           ...
           INVOKE MYSUB, (MY.LIB);

FATAL      WRONG LIBRARY SPECIFICATION
           XXXXXXXX
           ...
```

This error message can be misleading in IOF or in batch if LIST=ALL is not specified in the $JOB statement. The problem is in the sequence of statements inserted (and which are not printed unless LIST=ALL is specified), rather than in the INVOKE statement. The job should be re-run in batch with LIST=ALL on the $JOB statement.

❑

6.2.2 Labeling a JCL Statement

A label can be associated with any JCL statement simply by preceding the statement with the label name and a colon.

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.
- A JUMP statement cannot refer to an INVOKE (or other translator) statement. Consequently such translator statements cannot be labeled (however EXECUTE which is not a translator statement can be labeled).

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 LMI...;
                              ASSIGN...;
Step 1                     JUMP B...; (forward)
                              ASSIGN...;
                           B : ASSIGN...;
                        ENDSTEP;

Step 2            A :   STEP LM...;

                        ENDSTEP;

                  JUMP A...; (backward)

                  $ENDJOB;
```

Labels in an invoked JCL sequence 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.

6.3 Switches

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, SW0 = 1;

$ENDJOB;
```

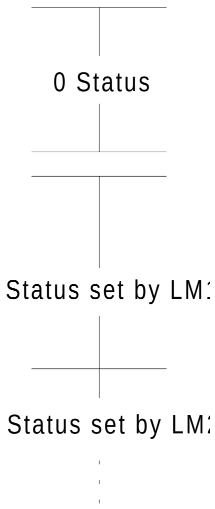
causes only LM2 to be executed. Note that the value of SW0 in the RUN statement has no effect on the switches of the father job. The transfer of information from one job to another via switches in the RUN statement can also be performed by means of switches in a RELEASE statement (see *RELEASE* above, and in *Section 1*).

6.4 Status

The status is a decimal number that is associated with each executing job step. Its purpose is to enable you to program, via JCL, the action to be taken in the event of execution time errors. The default action of the system is to abort if STATUS > 10000, and to carry on execution otherwise. The user with appropriate JUMP JCL statements may override this system default action. The use of status 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;
```

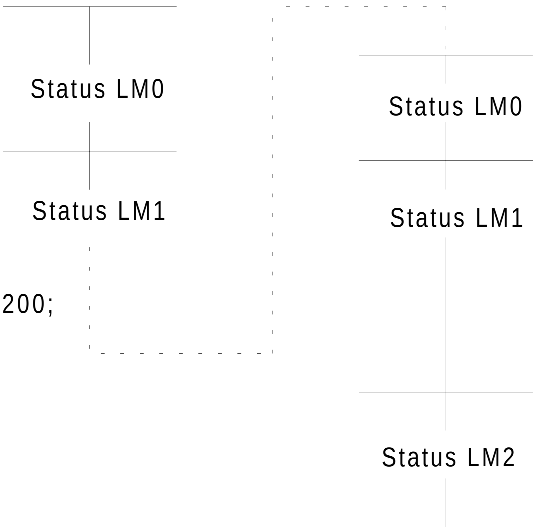


The status set by LM1 may be tested between the ENDSTEP statement labeled FIRST (which corresponds to LM1 execution) and the ENDSTEP statement labeled SECOND (which corresponds to a new load module execution).

❑

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 you can direct the flow of control of a job according to the execution of a particular load module.

You can set the status, also referred to as the step completion code, to any value 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 OLDFILE 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 OLDFILE HAS NOT BEEN SAVED will be sent to the operator; if the file save is unsuccessful (SEV0) 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). You can set the value of SEV by means of the LET statement.

Table 6-1 shows the relationship between a particular SEV grouping (with its corresponding status value range) and its significance to the system (for all SEV groups). Note that the interpretation of status values set under 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.

Table 6-1. Step Termination Conditions

STATUS		Meaning (when produced by system)	Job Occurrence Report message	Consequences
Group	**Value**			
SEV0	0-99	Normal termination	TASK TERMINATED STEP TERMINATED STATUS=SEV1 (or SEV2) or STATUS=numeric code (SEV1 or SEV2 is printed only if numeric code equals 100 or 1000 respectively)	Execution terminated normally. Job processing continues
SEV1	100-999	Normal termination + WARNING		
SEV2	1000-9999	Normal termination + WARNING		
SEV3	10000 19999	Work not performed due to user error. Step is not repeatable.	STEP ABORTED STATUS=SEV3 (or SEV4) or STATUS=numeric code (SEV3 or SEV4 is printed only if numeric code equals 10000 or 20000 respectively)	REPEAT option (Checkpoint/Restart): If operator enters YES, step repeated from last checkpoint. No REPEAT option (no Checkpoint/Restart): Scan JCL statement following ENDSTEP; - If no label, no JUMP CONTINUE, no JUMP with SEV test, or a SEV test does not match, abort the job; - If JUMP with SEV test that matches, jump and continue, but keep the SEV level; - If JUMP CONTINUE, jump and reset the SEV level, then continue; - If a label, scan next statement and begin same process again.
SEV4	20000 32767	Work not performed due to GCOS problem or to external events (I/O Error). Step is repeatable.		
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	

6.4.1 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 reads the next JCL statement (following ENDSTEP or the equivalent), and the following process takes place:

- if the statement is neither a label, a JUMP CONTINUE, nor a JUMP testing the severity level, the job is aborted;
- if it is a JUMP testing the severity level and the condition tested does not match, the job is aborted;
- if it is a JUMP testing the severity level and the condition tested matches, the jump takes place, the severity level is kept, but the JCL translation continues;
- if it is a JUMP CONTINUE, the severity level is reset, the jump takes place, and the JCL translation continues;
- if it is a label, the next JCL statement is read and the same procedure is repeated.

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 terminates 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 execute the jump to ABNORM (i.e., the SEND statement is skipped) 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.

6.4.2 Setting Severity Value

The LET statement with SEV parameter can be used to simulate an error condition thus allowing the processing sequence to be altered. 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 on whether the first step terminates normally (severity less than 3) or not. In other words, the success or failure of the first step determines which of the other 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 *Figure 6-1*.

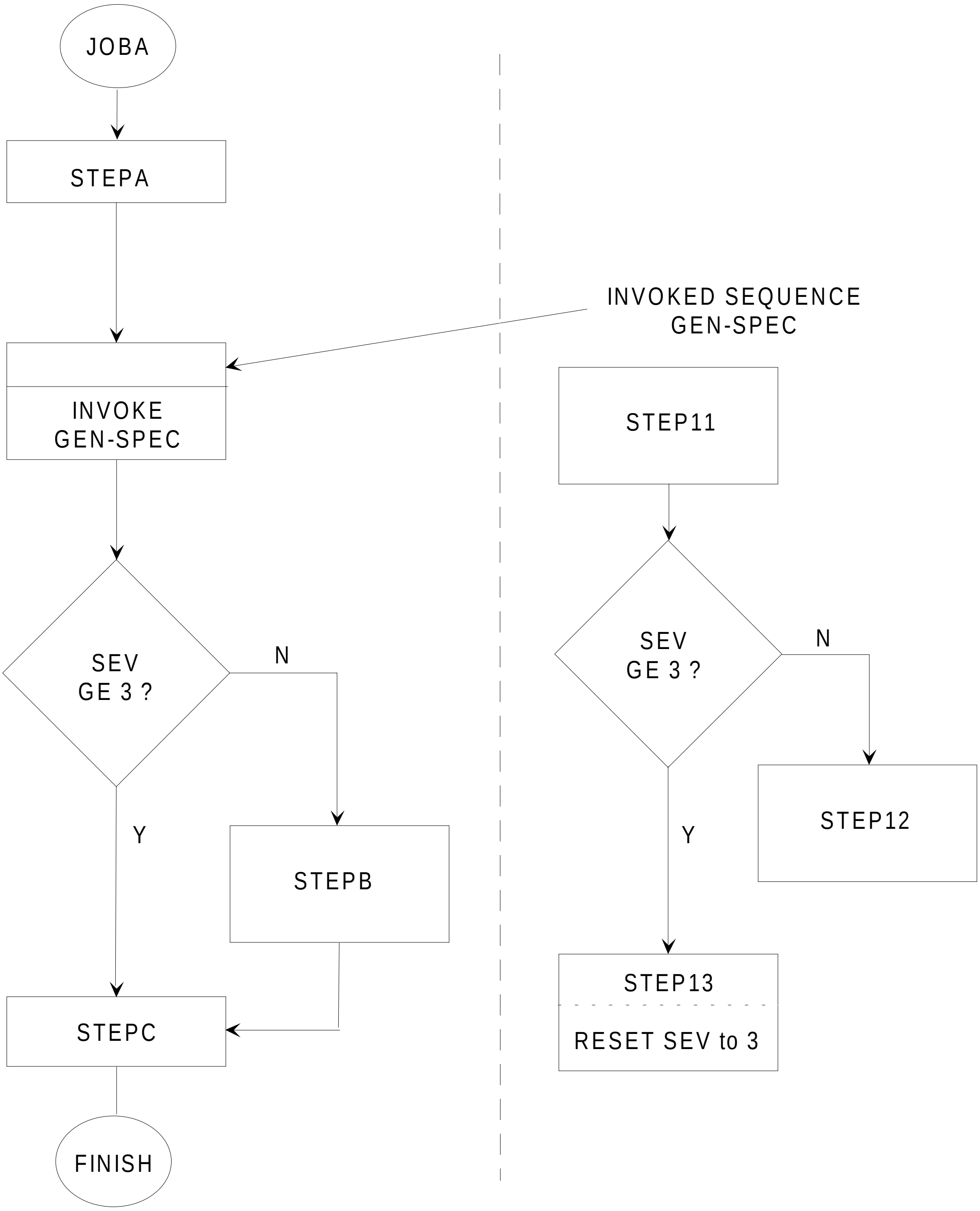


Figure 6-1. Use of LET SAVE

Figure 6-2 contains the JCL statements that correspond to the situation shown in *Figure 6-1*. If STEP11 terminates normally, STEP12 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, STEP11 aborts STEP13 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 STEP13. In this case only STEPC is executed next.

```
                $JOB JOBA,.....;
                  STEP STEPA,.....;           Contents of GEN.SPEC
                     .
                     .                           STEP STEP11......;
                     .                             .
                  ENDSTEP;                         .
                  INVOKE GEN.SPEC......;           .
                                                   .
                  JUMP LAST,SEV,GE,3;
                                                 ENDSTEP;
                  STEP STEPB,......;
                    .                              JUMP ERR,SEV,GE,3;
                    .
                    .                            STEP STEP12.....;
                    .
                    .
                                                 .
                  ENDSTEP.                       .
                  LAST: STEP STEPC.....;
                                                 ENDSTEP;
                    .
                    .                              JUMP END;
                                            ERR: STEP STEP13.....;
                      ENDSTEP;
$ENDJOB;
                                                   .
                                                   .
                                                   .
                                                 ENDSTEP;
                                               LET SEV,3;
                                                 END: JUMP CONTINUE;
```

Figure 6-2. Example of the Use of LET STATUS Group Value

7. Job Occurrence Report

7.1 Introduction

For each job run on the system and for each data enclosure entered, a system report is produced called the Job Occurrence Report (JOR).

The JOR has the following roles:

- To describe the work which has been submitted to the system and to list the JCL of that job.
- To describe the operations that take place in the order that they take place. Whenever a step is executed, messages are produced in the JOR that fully identify the step and give the result of the step execution. Each time an external event influences the execution of the job (i.e., operator intervention) a message is produced in the JOR to record this event.
- To record any abnormal situations by producing a warning or an error message in the JOR. Each message is coupled with a code that allows easy identification of error explanation in the documentation.
- To record the amount of resources (CPU time, I/Os) that have been used by the job during its execution. Most of the accounting information that is produced in the accounting file is also produced in the JOR.

The JOR is generally organized into separate sub-reports:

- The Job/Data Introduction and Translation Report that is produced when job descriptions or data enclosures are entered into the system.
- The Job Execution Report that is produced when the job is executed. In the case of data entry the Job Execution Report is not produced because the $DATA JCL statement does not imply that any job has been run.

The JOR messages usually begin in column 11. The left margin (columns 1.10) is reserved so that important messages can be emphasized (i.e., start and end step messages and error messages).

The Output Writer normally prints the JOR when the job terminates. After being printed it is automatically deleted. There are, however some cases when the JOR is not printed:

- For a user job, the JOR parameter of the $JOB JCL statement indicates whether or not the JOR is to be printed. The available options are:

NORMAL.	The JOR is automatically printed at job termination. This is the default.
NO.	The JOR is not printed.
ABORT.	The JOR is only printed when the job terminates abnormally.

- For a data entry, a JOR is not printed except in the following circumstances:
- An error has been detected during the data entry and an error message printed in the JOR.
- The PRINT option was present in the $DATA JCL statement.
- For an IOF session a JOR is optionally produced for each IOF user. By default this is not created, but it can be if requested by the user, for instance at IOF start-up. The JOR is then printed when the user logs off, but only the Job Execution Report that gives the interactively executed steps is provided.
- For service jobs the JOR is normally not printed unless the job aborts. However, the JOR of BTNS service jobs is always printed. This provides statistics on line usage. The JOR of the Magnetic Writers is also printed. It gives the contents of the file (rons/output).

7.2 Job Occurrence Report Description

This section describes how the Job Occurrence Report (JOR) is organized. Every message that appears during normal execution of the job is explained. Error messages will be detailed in the next section.

The JOR is divided into two separate sub-reports:

- The Job/Data Entry Introduction Report
- The Job Execution Report

The JOR is generally embedded with other outputs of the job between banners. These banners contain information printed in large characters to allow easy selection and routing of reports by the operator.

7.2.1 Output Writer Banners

These are in two forms: Output Writer Start Banner and Output Writer End Banner.

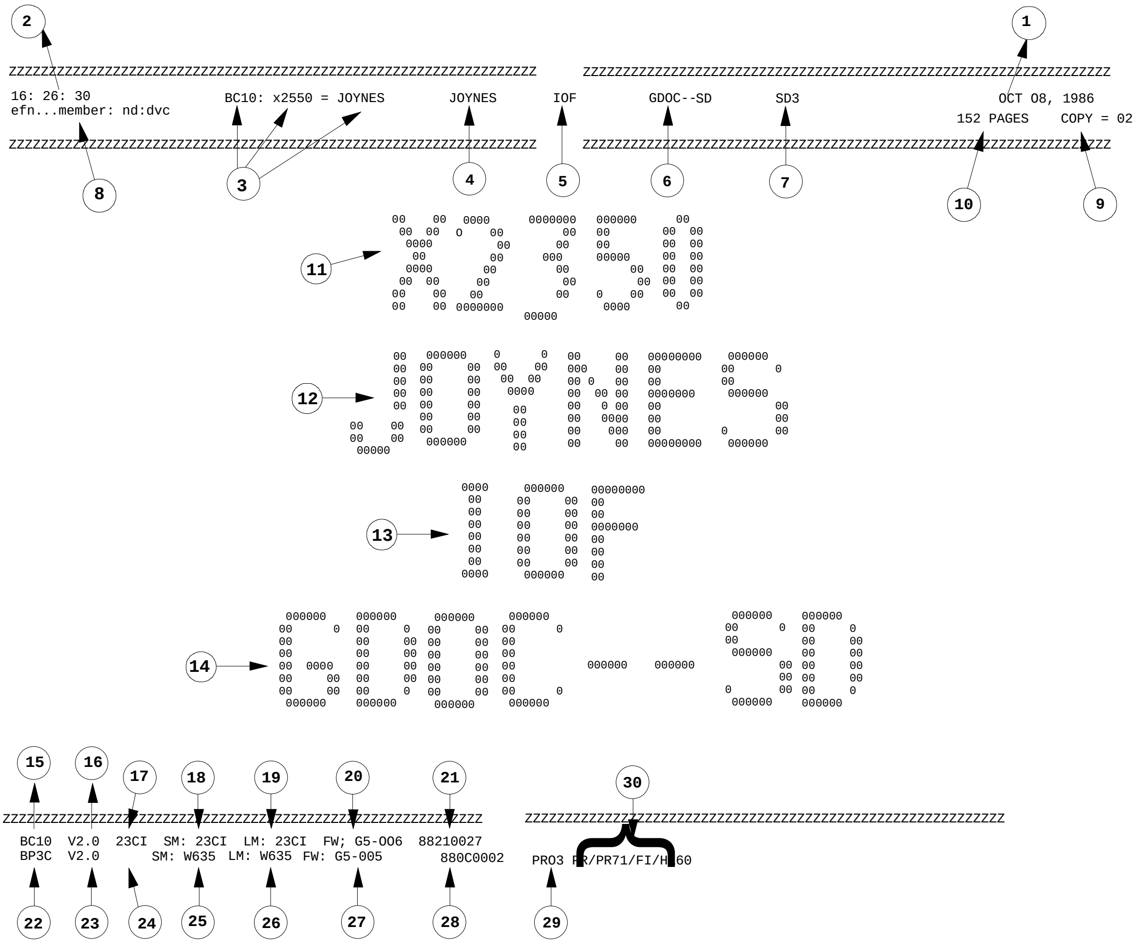
OUTPUT WRITER START BANNER

The first thing printed is the Output Writer Start Banner (Example in Figure 7-1). The information produced is:

1. Start print date
2. Start print time
3. Executing HOST, Run Occurrence Number of the Job, Output index or output name (if applicable)
4. User name
5. Job name
6. Billing
7. Project. An extra line may appear giving one or more of the following values, if applicable
8. Permanent file identifier with subfile name
9. Number of copies
10. Size of output in number of pages. Then four lines of text are output in large letters which give respectively
11. Run Occurrence Number
12. User name
13. Job name
14. Billing

NOTE:

The above values may be overridden by values provided by the user via the BANINF parameter of the JCL statement OUTVAL. Refer to Subsection 2.4 of this manual, and Section 4 of the JCL Reference Manual.



```
16: 26: 30                 BC10: x2550 = JOYNES       JOYNES       IOF          GDOC--SD        SD3                          OCT 08, 1986
efn...member: nd:dvc                                                                                                  152 PAGES     COPY = 02
```

```
 BC10  V2.0  23CI  SM: 23CI  LM: 23CI  FW; G5-OO6  88210027
 BP3C  V2.0       SM: W635  LM: W635  FW: G5-005       880C0002    PRO3 PR/PR71/FI/H160
```

Figure 7-1. Output Writer Start Banner

BOTTOM Lines:

When an output is printed that has been created on another site, or if the status of the site has changed since the output was saved on tape, an extra line is printed before the system status. This line contains:

15. Name of the system on which the output was created (host_id).

16. External name of the Software Release of the host system.

17. Version of the host system.

18. Shared modules of the host system used when producing the job.

19. Load modules of the host system used when producing the job.

20. Version of the firmware of the host system.

21. Identifier of the CPU of the host system.

    The last line describes the system which is printing the output and contains items 22 to 28 which are similar to items 15 to 21 above, plus items:

29. Printer name.

30. Printer attributes.

Output Writer End Banner

When all the outputs related to the same job have been printed, the Output Writer prints an End Banner. (Example in Figure 7-2.) The same information as in the Start Banner is printed, except that:

- only the Run Occurrence Number and User-name are printed in large letters,
- if the output could not be successfully printed, the reason for failure is given in place of the software and firmware descriptions. The reasons for this may be one of the following:

OUTPUTS HELD — The output has been retained on SYS.OUT by the system due to a fault during printing, or by an IOF user HO command.

OUTPUT HELD BY THE OPERATOR

The output has been retained on SYS.OUT by the operator (HO command) or HOLD answer has been given to the RESTART FROM? question.

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), or CAN answer has been given to the RESTART FROM? question.

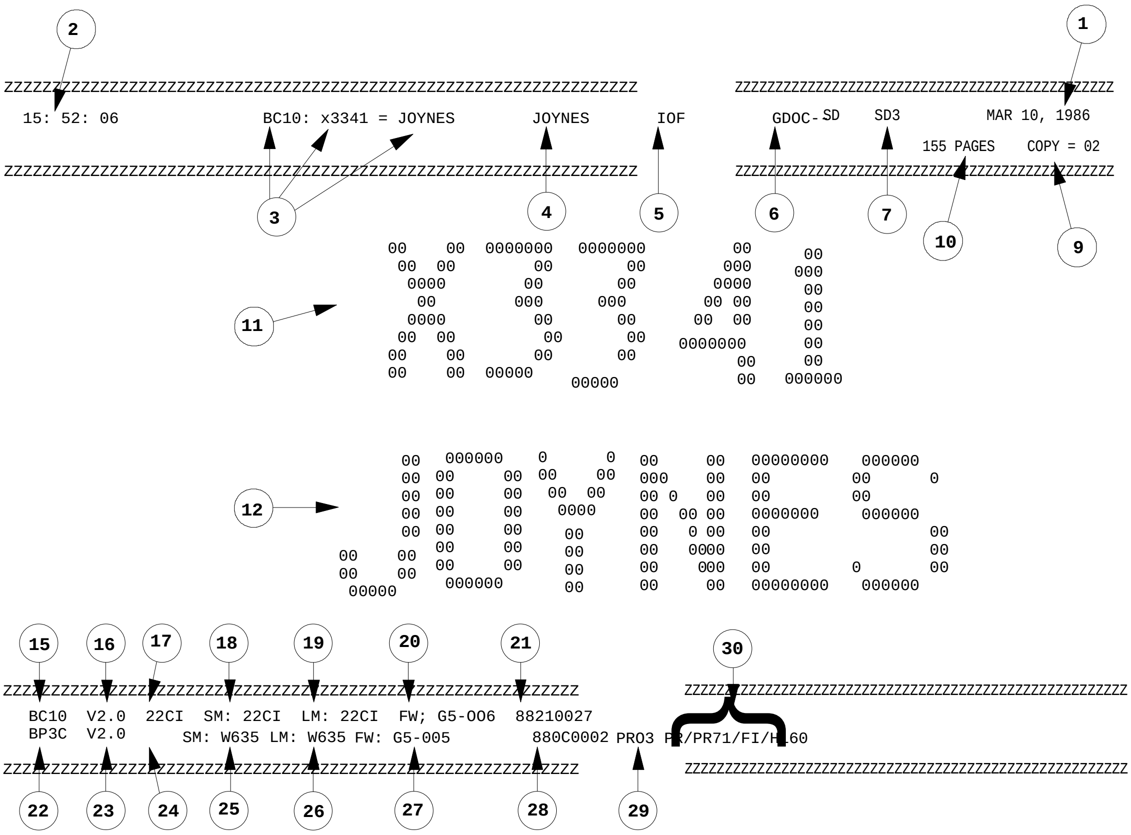
WRITER TERMINATED BY THE OPERATOR

All Output Writer activities have been terminated by the operator (TO command).

The Output Writer end banner is not produced if a severe or irrecoverable error occurs or if the STRONG option is included when stopping the Output Writer.

NOTE:

The printing of banners may be temporarily suppressed with the NBANNER (OUTVAL, SYSOUT, and WRITER statements). In the same way, the frequency of banner output may be controlled by the BANLEVEL parameter for the same statements.



```
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ     ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
  15: 52: 06              BC10: x3341 = JOYNES      JOYNES      IOF       GDOC-SD    SD3        MAR 10, 1986
                                                                                   155 PAGES    COPY = 02
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ     ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ

ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ     ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
  BC10  V2.0  22CI  SM: 22CI  LM: 22CI  FW; G5-OO6  88210027
  BP3C  V2.0     SM: W635 LM: W635 FW: G5-005        880C0002 PRO3 PR/PR71/FI/H160
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ     ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ
```

Figure 7-2. Output Writer End Banner

7.2.2 Job/Data Introduction and Translation

An example of the Job Introduction and Translation Report is given in Figure 7-3. The most important information is marked with a number, which refers to the explanations given below. The listing is organized in the following way:

A. Report Heading

14. The full identification of the job or data introduced: job-name, user-name, project, billing, Run Occurrence Number.

15. The date when the Job/data was introduced.

16. The input medium from which the Job/data was entered. In the example, the job was entered from a library member by a Start Job (SJ) command; the member name, library name, volume name and device class are given.

17. In the case of a job, the time at which job translation started is given.

B. Parameter Values

When the job has been introduced by a Start Job command, or a RUN JCL statement with parameters passed, the values of these parameters are printed.

C. Source JCL.

In the case of a job, the JCL is printed according to the LIST parameter of the $JOB JCL statement; the available options are:

SOURCE: Only the JCL at first level is printed: if some stored JCL is activated through the INVOKE or $SWINPUT statements, it is not printed.

ALL: All the JCL statements are printed including stored JCL activated by INVOKE and $SWINPUT JCL statements.

NO: Only the error messages are printed.

NOTE:

In addition to the rules stated above, if Access Rights have been implemented, stored JCL is printed only if the user has the READ access right on the files concerned. The JCL executed as a result of an EXECUTE statement is printed in a separate report since the translation takes place during the job execution.

The following information may also appear in the JCL list:

18. Error messages detected during the introduction or translation. 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 has been aborted at the end of translation.

19. Stored JCL sequences that are activated are flagged with one or several "*" characters in the margin; the number of asterisks indicates the level of nesting.

20. By default, data contained in input enclosures is not printed; only the number of records found in the input enclosure is given. However, when the PRINT option is present in the $INPUT JCL statement the data cards are printed, up to a limit of 200.

21. The number of records read during JCL introduction is given. Note that invoked JCL is counted as 2 records.

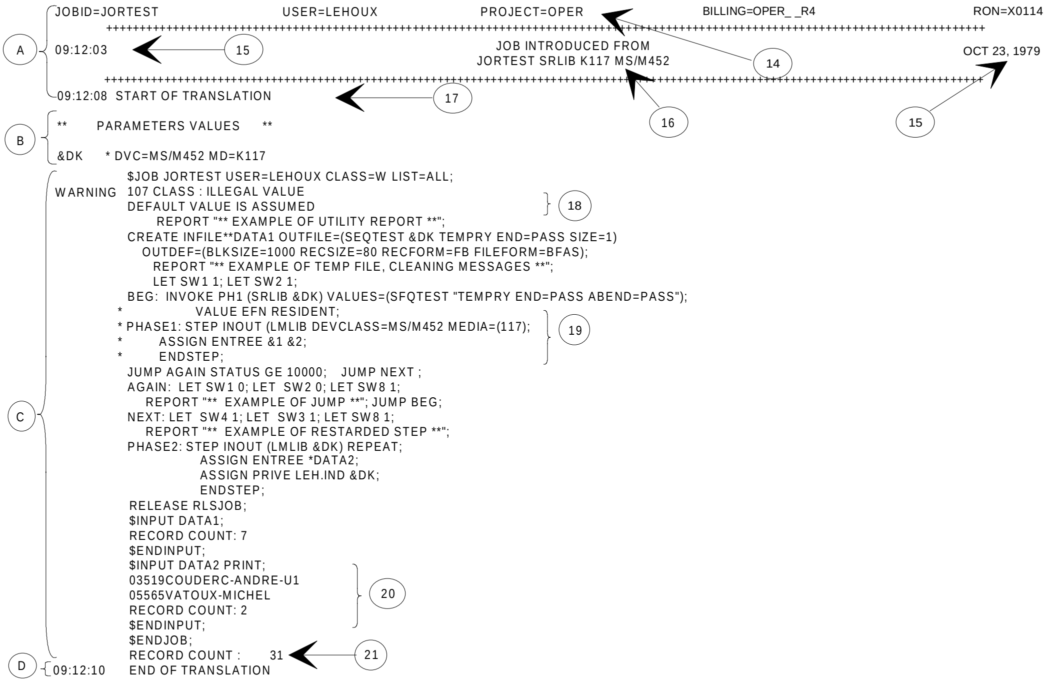
D. Report Footing

   This gives:

   a. The time when the translation finished (job)

   b. The result of job/data introduction;

```
               JOB ABORTED
   RESULT:     DATA INTRODUCTION COMPLETED
               DATA INTRODUCTION ABORTED
               END OF TRANSLATION
```



```
JOBID=JORTEST              USER=LEHOUX            PROJECT=OPER              BILLING=OPER_ _R4              RON=X0114
          ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
09:12:03                                                    JOB INTRODUCED FROM                                 OCT 23, 1979
                                                          JORTEST SRLIB K117 MS/M452
          ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
09:12:08  START OF TRANSLATION

**       PARAMETERS VALUES      **

&DK      * DVC=MS/M452 MD=K117
              $JOB JORTEST USER=LEHOUX CLASS=W LIST=ALL;
WARNING       107 CLASS : ILLEGAL VALUE
              DEFAULT VALUE IS ASSUMED
                    REPORT "** EXAMPLE OF UTILITY REPORT **";
              CREATE INFILE**DATA1 OUTFILE=(SEQTEST &DK TEMPRY END=PASS SIZE=1)
                 OUTDEF=(BLKSIZE=1000 RECSIZE=80 RECFORM=FB FILEFORM=BFAS);
                   REPORT "** EXAMPLE OF TEMP FILE, CLEANING MESSAGES **";
                   LET SW1 1; LET SW2 1;
              BEG:  INVOKE PH1 (SRLIB &DK) VALUES=(SFQTEST "TEMPRY END=PASS ABEND=PASS");
            *                VALUE EFN RESIDENT;
            * PHASE1: STEP INOUT (LMLIB DEVCLASS=MS/M452 MEDIA=(117);
            *          ASSIGN ENTREE &1 &2;
            *          ENDSTEP;
              JUMP AGAIN STATUS GE 10000;    JUMP NEXT ;
              AGAIN:  LET SW1 0; LET  SW2 0; LET SW8 1;
                  REPORT "**  EXAMPLE OF JUMP **"; JUMP BEG;
              NEXT: LET  SW4 1; LET  SW3 1; LET SW8 1;
                  REPORT "**  EXAMPLE OF RESTARDED STEP **";
              PHASE2: STEP INOUT (LMLIB &DK) REPEAT;
                              ASSIGN ENTREE *DATA2;
                              ASSIGN PRIVE LEH.IND &DK;
                              ENDSTEP;
              RELEASE RLSJOB;
              $INPUT DATA1;
              RECORD COUNT: 7
              $ENDINPUT;
              $INPUT DATA2 PRINT;
              03519COUDERC-ANDRE-U1
              05565VATOUX-MICHEL
              RECORD COUNT: 2
              $ENDINPUT;
              $ENDJOB;
              RECORD COUNT :       31
09:12:10      END OF TRANSLATION
```

Figure 7-3. Job Introduction and Translation

7.2.3 Job Execution Report Structure

Some messages may be printed in the Job Execution Report during each phase of job processing. The organization of the report is as follows:

E. Job Initiation Messages

F. Job Termination Messages

G. Step Initiation Messages

H. Step Execution Messages

I. Task Termination Messages

J. Step Clearing Messages

K. File Usage Messages

L. System Resources Statistics Messages

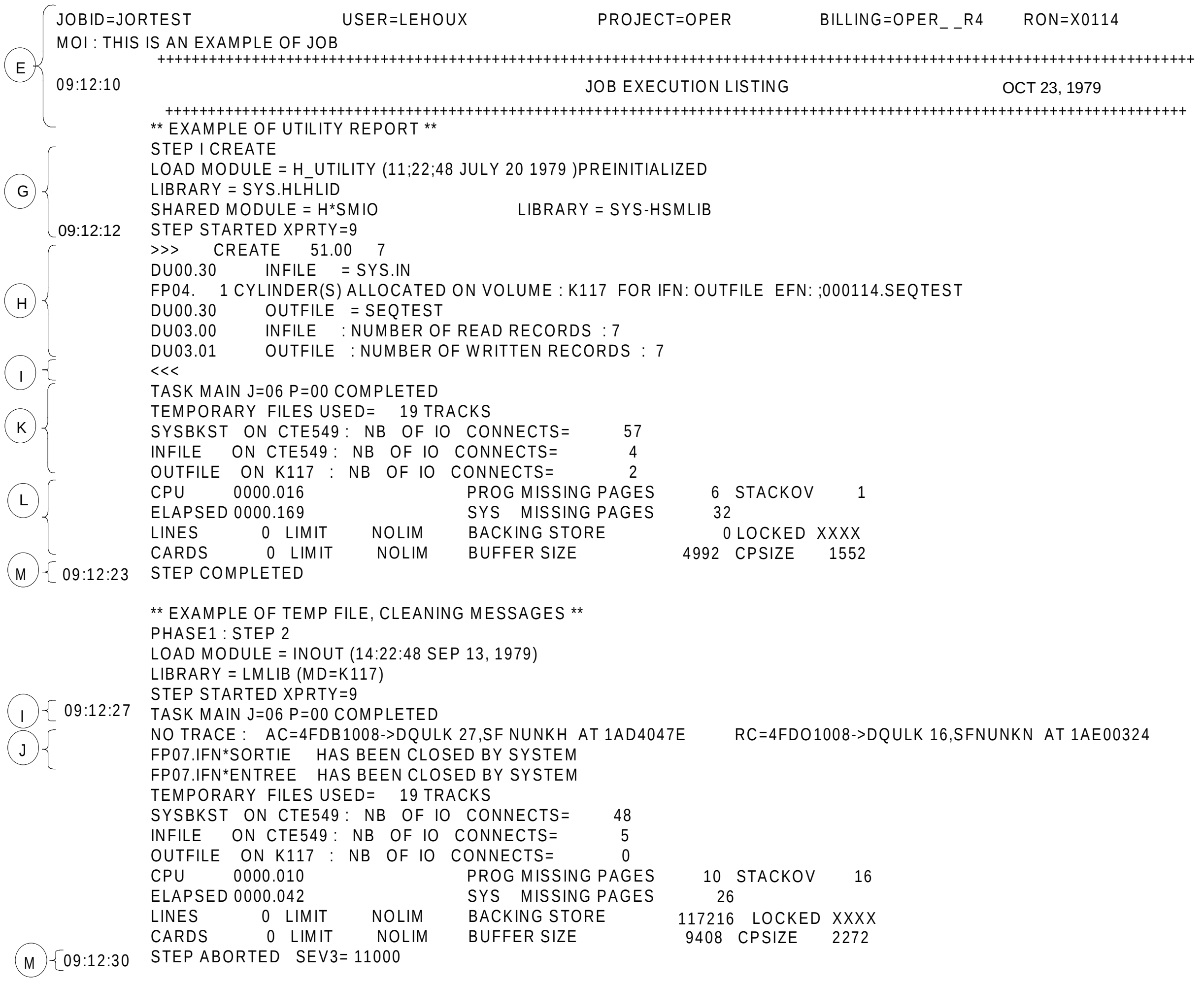
M. Step Result Messages

In addition to the above messages, the following may appear anywhere in the Job Execution Report.

N. Messages at restart time that appear in the event of a system crash

O. Job execution trace messages

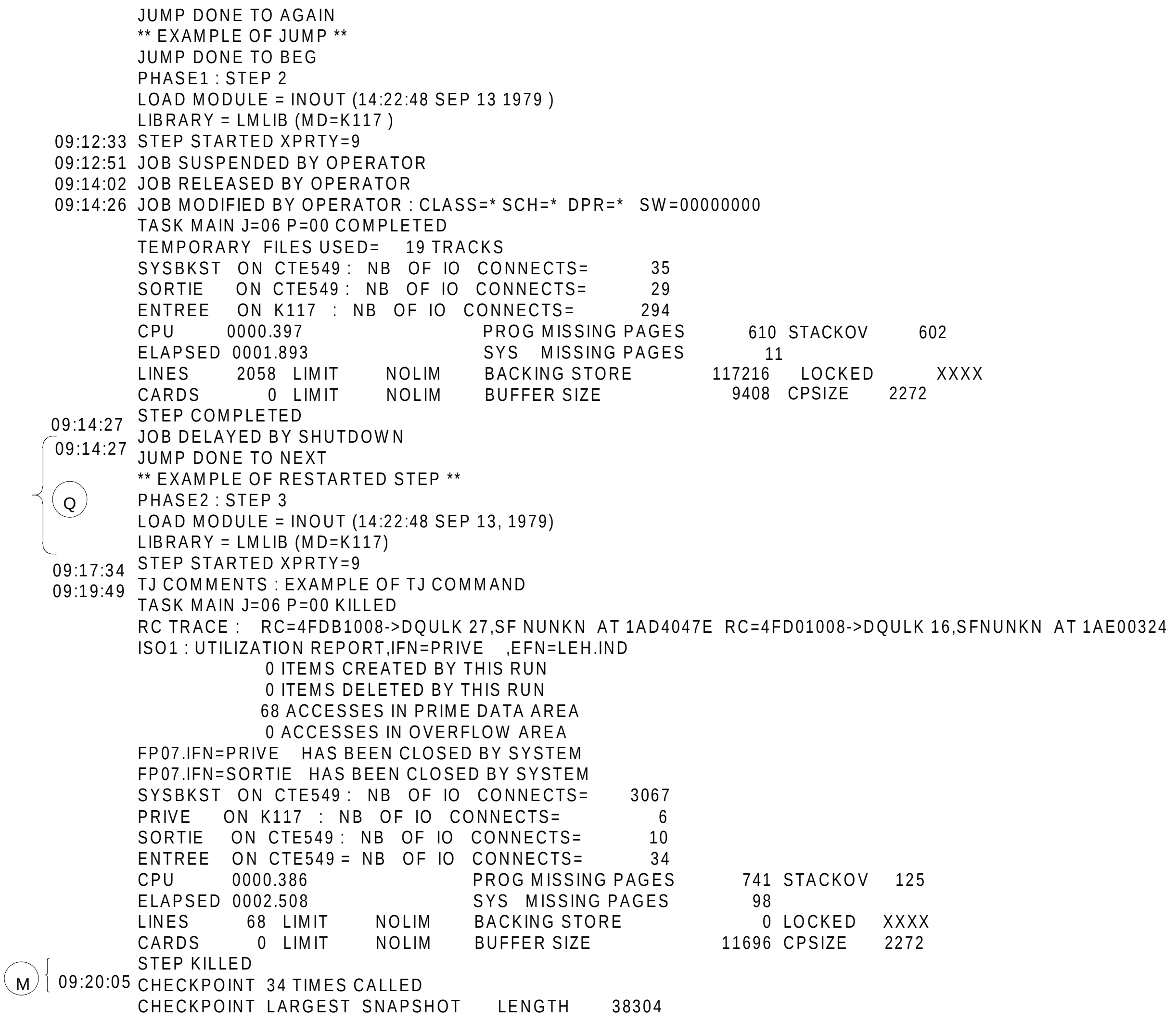
The above letters are references to the text below where a detailed explanation is given along with examples.



E
G
H
I
K
L
M
I
J
M

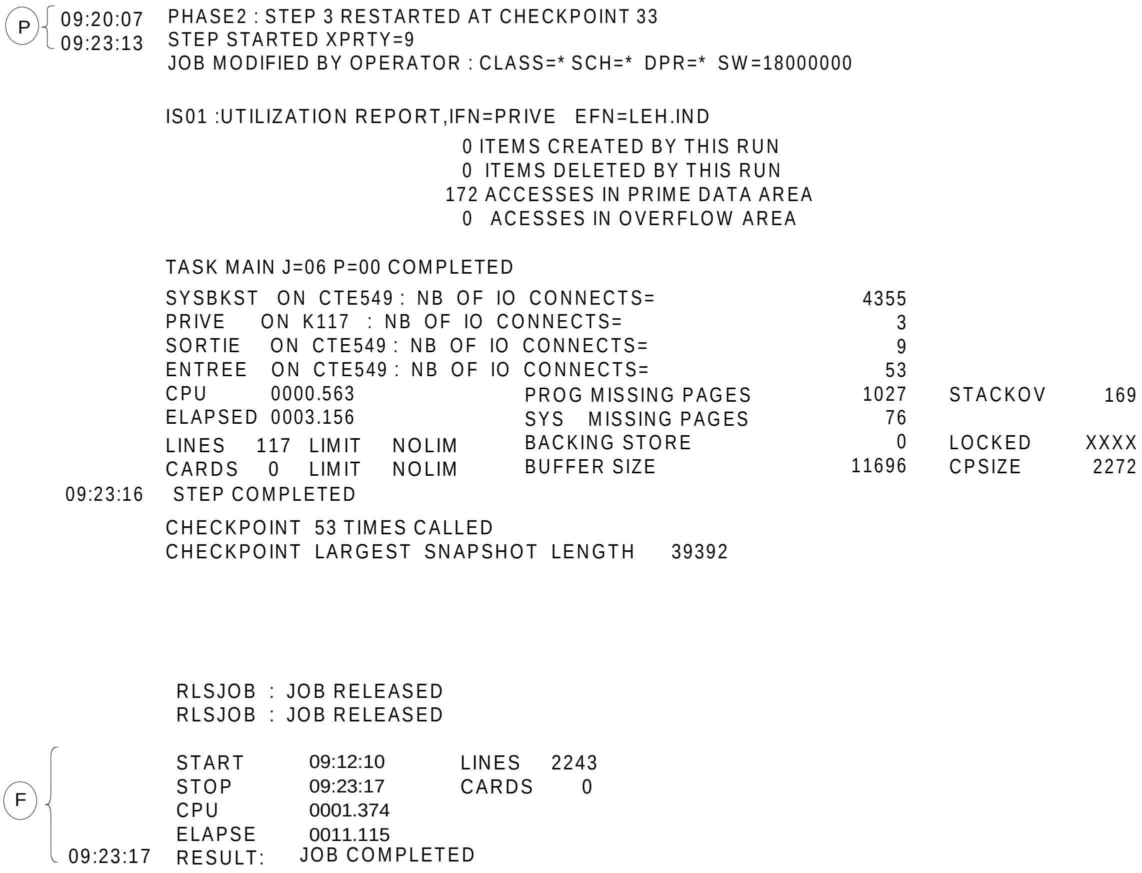
```
JOBID=JORTEST                USER=LEHOUX               PROJECT=OPER          BILLING=OPER_ _R4     RON=X0114
MOI : THIS IS AN EXAMPLE OF JOB
               +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
09:12:10                                                  JOB EXECUTION LISTING                         OCT 23, 1979
                +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
              ** EXAMPLE OF UTILITY REPORT **
              STEP I CREATE
              LOAD MODULE = H_UTILITY (11;22;48 JULY 20 1979 )PREINITIALIZED
              LIBRARY = SYS.HLHLID
              SHARED MODULE = H*SMIO                    LIBRARY = SYS-HSMLIB
09:12:12      STEP STARTED XPRTY=9
              >>>     CREATE     51.00    7
              DU00.30        INFILE    = SYS.IN
              FP04.    1 CYLINDER(S) ALLOCATED ON VOLUME : K117  FOR IFN: OUTFILE  EFN: ;000114.SEQTEST
              DU00.30        OUTFILE   = SEQTEST
              DU03.00        INFILE    : NUMBER OF READ RECORDS  : 7
              DU03.01        OUTFILE   : NUMBER OF WRITTEN RECORDS  :  7
              <<<
              TASK MAIN J=06 P=00 COMPLETED
              TEMPORARY  FILES USED=    19 TRACKS
              SYSBKST   ON  CTE549 :  NB   OF  IO   CONNECTS=        57
              INFILE     ON  CTE549 :  NB   OF  IO   CONNECTS=         4
              OUTFILE   ON  K117   :  NB   OF  IO   CONNECTS=         2
              CPU        0000.016                  PROG MISSING PAGES        6   STACKOV      1
              ELAPSED 0000.169                     SYS    MISSING PAGES       32
              LINES          0   LIMIT       NOLIM       BACKING STORE              0 LOCKED  XXXX
              CARDS          0   LIMIT       NOLIM       BUFFER SIZE            4992   CPSIZE     1552
09:12:23      STEP COMPLETED

              ** EXAMPLE OF TEMP FILE, CLEANING MESSAGES **
              PHASE1 : STEP 2
              LOAD MODULE = INOUT (14:22:48 SEP 13, 1979)
              LIBRARY = LMLIB (MD=K117)
              STEP STARTED XPRTY=9
09:12:27      TASK MAIN J=06 P=00 COMPLETED
              NO TRACE :    AC=4FDB1008->DQULK 27,SF NUNKH  AT 1AD4047E        RC=4FDO1008->DQULK 16,SFNUNKN  AT 1AE00324
              FP07.IFN*SORTIE     HAS BEEN CLOSED BY SYSTEM
              FP07.IFN*ENTREE    HAS BEEN CLOSED BY SYSTEM
              TEMPORARY  FILES USED=    19 TRACKS
              SYSBKST   ON  CTE549 :  NB   OF  IO   CONNECTS=       48
              INFILE     ON  CTE549 :  NB   OF  IO   CONNECTS=         5
              OUTFILE   ON  K117   :  NB   OF  IO   CONNECTS=         0
              CPU        0000.010                  PROG MISSING PAGES       10   STACKOV      16
              ELAPSED 0000.042                     SYS    MISSING PAGES        26
              LINES          0   LIMIT       NOLIM       BACKING STORE        117216   LOCKED  XXXX
              CARDS          0   LIMIT       NOLIM       BUFFER SIZE            9408   CPSIZE     2272
09:12:30      STEP ABORTED   SEV3= 11000
```

Figure 7-4.   Example of Job Execution Report (1/3)



```
          JUMP DONE TO AGAIN
          ** EXAMPLE OF JUMP **
          JUMP DONE TO BEG
          PHASE1 : STEP 2
          LOAD MODULE = INOUT (14:22:48 SEP 13 1979 )
          LIBRARY = LMLIB (MD=K117 )
09:12:33  STEP STARTED XPRTY=9
09:12:51  JOB SUSPENDED BY OPERATOR
09:14:02  JOB RELEASED BY OPERATOR
09:14:26  JOB MODIFIED BY OPERATOR : CLASS=* SCH=*  DPR=*   SW=00000000
          TASK MAIN J=06 P=00 COMPLETED
          TEMPORARY  FILES USED=     19 TRACKS
          SYSBKST   ON  CTE549 :  NB   OF  IO   CONNECTS=          35
          SORTIE    ON  CTE549 :  NB   OF  IO   CONNECTS=          29
          ENTREE    ON  K117   :  NB   OF  IO   CONNECTS=         294
          CPU        0000.397                   PROG MISSING PAGES        610  STACKOV       602
          ELAPSED  0001.893                     SYS    MISSING PAGES       11
          LINES       2058   LIMIT       NOLIM      BACKING STORE          117216    LOCKED         XXXX
          CARDS          0   LIMIT       NOLIM      BUFFER SIZE              9408   CPSIZE      2272
09:14:27  STEP COMPLETED
09:14:27  JOB DELAYED BY SHUTDOWN
          JUMP DONE TO NEXT
          ** EXAMPLE OF RESTARTED STEP **
   Q      PHASE2 : STEP 3
          LOAD MODULE = INOUT (14:22:48 SEP 13, 1979)
          LIBRARY = LMLIB (MD=K117)
09:17:34  STEP STARTED XPRTY=9
09:19:49  TJ COMMENTS : EXAMPLE OF TJ COMMAND
          TASK MAIN J=06 P=00 KILLED
          RC TRACE :    RC=4FDB1008->DQULK 27,SF NUNKN  AT 1AD4047E  RC=4FD01008->DQULK 16,SFNUNKN  AT 1AE00324
          ISO1 : UTILIZATION REPORT,IFN=PRIVE    ,EFN=LEH.IND
                          0 ITEMS CREATED BY THIS RUN
                          0 ITEMS DELETED BY THIS RUN
                         68 ACCESSES IN PRIME DATA AREA
                          0 ACCESSES IN OVERFLOW AREA
          FP07.IFN=PRIVE    HAS BEEN CLOSED BY SYSTEM
          FP07.IFN=SORTIE   HAS BEEN CLOSED BY SYSTEM
          SYSBKST   ON  CTE549 :  NB   OF  IO   CONNECTS=       3067
          PRIVE     ON  K117   :  NB   OF  IO   CONNECTS=          6
          SORTIE    ON  CTE549 :  NB   OF  IO   CONNECTS=         10
          ENTREE    ON  CTE549 =  NB   OF  IO   CONNECTS=         34
          CPU         0000.386                  PROG MISSING PAGES         741  STACKOV    125
          ELAPSED  0002.508                     SYS   MISSING PAGES         98
          LINES         68   LIMIT       NOLIM      BACKING STORE                0  LOCKED     XXXX
          CARDS          0   LIMIT       NOLIM      BUFFER SIZE              11696  CPSIZE     2272
          STEP KILLED
M  09:20:05 CHECKPOINT  34 TIMES CALLED
          CHECKPOINT  LARGEST  SNAPSHOT      LENGTH      38304
```

Figure 7-4. Example of Job Execution Report (2/3)



```
P { 09:20:07  PHASE2 : STEP 3 RESTARTED AT CHECKPOINT 33
    09:23:13  STEP STARTED XPRTY=9
              JOB MODIFIED BY OPERATOR : CLASS=* SCH=*  DPR=*  SW=18000000

              IS01 :UTILIZATION REPORT,IFN=PRIVE   EFN=LEH.IND
                                              0 ITEMS CREATED BY THIS RUN
                                              0  ITEMS DELETED BY THIS RUN
                                            172 ACCESSES IN PRIME DATA AREA
                                              0   ACESSES IN OVERFLOW AREA

              TASK MAIN J=06 P=00 COMPLETED
              SYSBKST   ON  CTE549 :  NB  OF  IO  CONNECTS=                   4355
              PRIVE     ON  K117   :  NB  OF  IO  CONNECTS=                      3
              SORTIE    ON  CTE549 :  NB  OF  IO  CONNECTS=                      9
              ENTREE    ON  CTE549 :  NB  OF  IO  CONNECTS=                     53
              CPU        0000.563                PROG MISSING PAGES        1027    STACKOV      169
              ELAPSED  0003.156                  SYS    MISSING PAGES        76
              LINES     117   LIMIT     NOLIM    BACKING STORE                0    LOCKED      XXXX
              CARDS     0     LIMIT     NOLIM    BUFFER SIZE              11696    CPSIZE      2272
      09:23:16  STEP COMPLETED
              CHECKPOINT  53 TIMES CALLED
              CHECKPOINT  LARGEST  SNAPSHOT  LENGTH     39392



               RLSJOB  :  JOB RELEASED
               RLSJOB  :  JOB RELEASED

               START         09:12:10        LINES     2243
               STOP          09:23:17        CARDS        0
F {            CPU           0001.374
               ELAPSE        0011.115
      09:23:17 RESULT:     JOB COMPLETED
```

Figure 7-4. Example of Job Execution Report (3/3)

7.2.4 Job Initiation and Termination Messages

JOB INITIATION MESSAGES (E)

The following information is given:

- The full identification of the job: job name, user name, project name, run occurrence number.
- The Message of Today (MOT) which has been created by the operator and normally gives general information about the installation (e.g., shut down time).
- The date and time at which execution started.

JOB TERMINATION MESSAGES (F)

The following information is given:

START: time at which execution started

STOP: time at which execution ended

LINES: number of lines registered in the SYSOUTs

CARDS: number of punched cards registered in the SYSOUTs

CPU: total CPU time used by the steps, expressed in minutes (excluding System Functions)

ELAPSE: job elapsed time. This is normally greater than the sum of the elapsed time of all steps. The difference is the time spent waiting for resources and time used by system tasks.

RESULT: indicates the time at which the job terminated and the result of the job: COMPLETED, ABORTED or KILLED.

STEP INITIATION MESSAGES

When a step execution begins, messages are given in the JOR which allow the user to identify the step being executed. The messages given depend on the type and result of the step initiation.

Normal Step Initiation. (G)

```
   [label:] STEP ssn [utility-name]

   LOAD MODULE = lm-name (lm-creation-date)    [PREINITIALIZED]

   LIBRARY = lm-lib-name [(MD = volume-name)]

   [SHARED MODULE = sm-name LIBRARY = sm-lib-name]

hh.mm.ss STEP STARTED XPRTY = priority
```

Description of Parameters:

Label: is the label that immediately precedes the STEP or utility statements in JCL.

ssn: is the static step number. A number is allocated to each step at JCL translation time when a STEP statement or utility statement appears.

Utility-name: When the step is a utility, the utility name is also given.

lm-name: the name of the load module being executed as found in the STEP JCL statement.

lm-creation-date: the time and date that the load module was created. This is presented in the format:

```
hh.mm.ss month, year
```

PREINITIALIZED: The load module was already present in backing store: it had been loaded previously by a PLM OCL command.

lm-lib-name: The name of the library where the load module was found.

Volume-name: The name of the volume where the load module resides; it is given only if the load module is not preinitialized or if the load module library is not resident.

sm-name: Identifies the shared modules used by the step; (i.e., in a TDS step); identifies the name of the library which contains the shared modules.

hh.mm.ss: Gives the time at which step execution started.

Priority: Gives the execution priority of the step at start time.

Restart of a Step: (P)

```
[label:] STEP ssn [utility-name] {REPEATED                 }
                                 {RESTARTED AT CHECKPOINT n }
hh.mm.ss STEP STARTED XPRTY = priority
```

Description of Parameters:

Label: has the same meaning as for normal execution

ssn: has the same meaning as for normal execution

Utility-name: has the same meaning as for normal execution

REPEATED: the step has been restarted from the beginning

RESTARTED FROM: the step has been restarted from checkpoint n

hh.mm.ss: indicates the time the step execution restarted

Priority: indicates the execution priority of the step at restart time.

Abnormal Step Initiation:

```
[label.] STEP ssn [utility-name]
[error-message]

                 {DELAYED BY SHUTDOWN                    }
STEP INITIATION  {KILLED SEV 5                           }
                 {ABORTED SEVi [RC = edited-return-code] }
```

Description of Parameters:

Label: has the same meaning as for normal execution

ssn: has the same meaning as for normal execution

Utility-name: has the same meaning as for normal execution.

Error-message: gives code for the abnormal step initiation; Refer to the System Error Messages and Return Codes Manual for a description of the code.

DELAYED BY SHUTDOWN:

indicates that the operator used an END SYSTEM SESSION command whilst the step was in the step initiation phase possibly waiting for a resource. The step initiation is delayed and will be restarted at the next session.

KILLED: indicates that the operator has issued a TJ command without the STRONG option during the step initiation. This may be because one of the resources requested by the job was not available.

ABORTED: indicates that the step initiation could not be completed; the reason is indicated by the return code. In most cases, a preceding explanation of the error message gives a more explicit explanation of the error that has arisen. SEV3 indicates that the step initiation has failed because of a user error whilst SEV4 means that the failure was due to an irrecoverable I/O error.

STEP EXECUTION MESSAGES (H)

During step execution some messages may be logged in the JOR either by the program itself or by system procedures called by the program. The following types of messages may appear:

- Information and Error Messages:
- These messages always start with a message key. They are documented in the System Error Messages and Return Codes Manual.

FOR EXAMPLE:

```
CBL13. ACCEPT program-id entered-data
```

❑

- These messages may be:

Informative messages: CBL13 makes a trace of text entered from the console as a result of an ACCEPT instruction.

Error messages: which indicate an abnormal situation has been detected. This may result in step abortion.

Mini-reports Some utilities, especially Data Management utilities, may produce a report in the JOR that summarizes the operations performed. This report is embedded between a Start Banner and an End Banner and has the following format:

```
>>> utility-name utility-version      start banner
.
.
.
.                                     utility report
.
.
.
.                                     end banner
>>>
```

Other steps such as BTNS and TDS may produce statistical results in the JOR. The reports produced by these utilities are documented in the appropriate manuals.

STEP TERMINATION MESSAGES

When a user program terminates, the following operations take place:

- Termination of the task(s). Each step is mapped on one or several tasks; the task is the level of management from the system point of view. Whenever a task terminates, the result of the task is printed in the JOR.
- Clearing of the step environment, for example, files still open at this time are closed by the system.
- Creation of accounting information relative to
  - file usage
  - system resources such as CPU time, memory use.

After this has been done the step automatically terminates and the result is printed in the JOR.

During these phases messages are printed in the JOR and are described below.

TASK TERMINATION (I)

For each task the termination result is printed. It should be noted that this result is only from the system point of view and has no relative significance to the result of the step; for example, when compiling a COBOL program containing FATAL errors, the TASK is considered COMPLETED but the STEP is considered ABORTED because FATAL errors have been encountered. The message has the following format.

```
TASK task name J = j number P = p number
               {COMPLETED [RC = edited-return-code ]          }
               {ABORTED BY SYSTEM [RC = edited-return-code ]   }
               {ABORTED BY USER. TERMINATOR CODE = status     }
               {KILLED                                        }

[RC TRACE : RC = edited-return-code AT ADDRESS address]
```

Description of Parameters:

task name: name of the task. In a mono task step this name is MAIN

j-number: the process group number (step number) as known by the hardware

p-number: the process number (task number) as known by the hardware

COMPLETED: the task has reached a normal end from the system point of view

ABORTED BY SYSTEM: the task has been aborted by the system because a program exception has been detected (refer to messages EX.xx in the System Error Messages and Return Codes Manual) or because a line limit has been reached.

ABORTED BY USER: a task abort has been requested. The termination code gives the status value in decimal. This may happen when the system detects invalid structures during step execution

KILLED: a Terminate Job (TJ) command has been issued during step execution

RC TRACE: a trace of the last four abnormal return codes set by functions called by the program. This information is useful to the Service Center in case of system malfunction.

STEP CLEARING MESSAGES (J)

When messages are produced in the JOR during the step-clearing phrase, they begin with a message key. These messages are documented in the System Error Messages and Return Codes Manual.

FILE USAGE (K)

The following information is given:

When temporary files have been used during the step execution, the sum of allocated space for all temporary files used is given under the heading "TEMPORARY FILES USED".

For each file that has been opened during the step execution the following is given:

ifn: internal file name

ON volume-name: the name of the volume on which the file resides. In the case of a multi-volume disk file, if the file is assigned with MOUNT=1, the names of all volumes supporting the files are printed. Without MOUNT=1, only the name of the first volume is printed.

NB OF I/O CONNECTS: the number of physical I/Os issued by the program for that file on the specified volume (1 per block transfer + file OPEN, CLOSE etc).

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. When no I/O retries have been performed this value is not given. A high value may indicate that the volume or drive is damaged.

In addition to the user defined files, the following conventional files may appear in the list:

SYSBKST: "system backing store". This gives the number of I/Os performed on the system disk, including swapping of segments of the step

PRMBKST: "permanent backing store". This gives the number of I/Os performed on this file, when the file does not reside on the system disk. This file contains the preinitialized load modules and the checkpoint images.

TMPBKST: "temporary backing store". This gives the number of I/Os performed for the step on this file that has been allocated to the step, provided that the file does not reside on the system disk.

TMPBKST*: Gives the number of I/Os performed for swapping out segments belonging to other steps (on volumes different from the system disk) to swap in a segment of the current step.

JOURNAL: Gives the number of I/Os performed on the journal files for the step. If Before Journal was used, "BEFORE" appears instead of volname; if After Journal was used, "AFTER" appears in lieu of volname.

NOTE: As all steps using the After Journal at the same time share the same buffer, the I/O is account to the step that caused the physical writing of the buffer.

ROLLBACK: Appears when files are rolled back after a step abort or system crash. Gives the number of I/Os performed to read the Before Journal in order to rollback the files.

H_DPPR: Appears when the step aborted and a dump has been taken. Gives the number of I/Os performed while storing the dump in the SYSOUT.

NOTE: A user who wishes to save a dump on a file (rather than printing it) may do so by assigning this efn to a file.

SYSTEM RESOURCES STATISTICS (L)

This information gives the amount of resources used by the step. The following descriptions are given:

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. The time is expressed in minutes. In the case of a multitask step, the value given is the sum of the CPU time used by each task. Details about CPU time used per task can be found in the accounting file.

All the CPU time output within the user process-group is accounted for, including the following:

- user processing (COBOL, RPG, FORTRAN...)
- run-time packages (if any)
- data management including all functions used; open, get, put, close, editing in the case of the Sysout access method, etc.
- virtual memory activity (mainly missing segment handling...)
- physical I/O initiation
- part of step initiation and step termination.

All "centralized" system functions are left out (They are executed within the System process group). These functions include:

- physical I/O termination
- scheduling and sequencing of job and steps
- system availability management (management of retries and error logging).
- management of SYSIN and SYSOUT queues
- management of operator dialog.

Very little time is spent in the execution of these centralized system functions: they usually account for well under 10 % (or even 5 %) of total CPU usage.

ELAPSED: The elapsed time between the end of step initiation and the end of step termination. The time is expressed in minutes.

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. This value can be used to evaluate the working set of the program specified in the SIZE JCL statement.

BACKING STORE: Total amount of temporary backing store that has been allocated to the step. The size is expressed in bytes although backing store is allocated in units of 39K bytes. In the case of preinitialized load modules and repeatable steps with or without checkpoints, the size occupied by the original copy of the segments or by the checkpoint snapshot is not included in this value and only the modified segments which have been swapped out are counted.

LOCKED: The size of the non-relocatable resident segments.

LINES: The number of lines in the SYSOUT files relating to this step. In the case of step abort with dump, the lines created for the dump are included in this value.

CARDS: The number of card images in the SYSOUT files relating to this step.

LIMIT: The maximum number of LINES or CARDS allowed for this job as specified in the STEP JCL statement. NOLIM is taken as default if no limit is specified.

STACKOV: Number of stack overflows. Stack segment size is dynamically adapted (increased or decreased) according to the user needs. The value given indicates the increase in the number of lines. A high value here may indicate that a procedure activation in a loop caused increasing stack size adaptation.

STEP RESULT (M)

The following message is displayed:

```
              {CDOMPLETED  [SEV. [= status-value]]   }
hh.mm.ss STEP {ABORTED     [SEV. [= status-value]]   }
              {KILLED                                }
```

Parameter Description:

COMPLETED: The step execution is considered as correct; that is, the status-value is less than 10000. If the status-value is not 0 then it is displayed in the message.

ABORTED: The step execution is considered as incorrect; that is, the status-value is greater than or equal to 10000. The next step of the job will not be executed unless a JUMP JCL statement follows the step description in the JCL.

KILLED: The operator issued a Terminate Job (TJ) command.

If the step used the checkpoint/restart facility the following messages are also displayed.

```
CHECKPOINT nn TIMES CALLED
CHECKPOINT LARGEST SNAPSHOT LENGTH snapshot-size
```

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.

MESSAGES AT RESTART TIME (N)

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, a warm restart performs the necessary operations to allow the inter-step statement to be executed. In this case the following message is written on the JOR:

```
JOB RESTART AFTER A SYSTEM CRASH
```

If a warm restart aborts the job that was being executed or was suspended at the time of the crash because of irrecoverable inconsistencies found in its structure, the following message appears in 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 is required
FILREST:	A file restore should be performed
VOLREST:	A volume restore should be performed
VOLCHECK:	Volume checking is required
DEALLOC:	Deallocate the file
PREALLOC:	Preallocate the file
VOLPREP:	A volume preparation should be performed
UNKNOWN:	Damage has been done but the system is unable to establish the type of recovery action necessary.

JOB EXECUTION TRACE (O)

As one of the purposes of the Job Execution report is to trace every event that arises during all aspects of job execution, the following messages may also appear as the result of a specific event. These can appear at the execution of a JUMP JCL statement or where an operator command has influenced job execution or where the execution of a JCL statement does not imply any step execution.

```
COMMAND ABORTED: command text. RC=edited-return-code
```

This is where an OCL command has been found in the JCL. This is normally treated as if it was entered at the operator console, but in this case the command has not been accepted by the system. The return-code gives the reason for the problem. If this code is CDUNKN, this indicates that the command submitted is not OCL. Other codes normally indicate a system error.

```
Job-name: HOLD COUNT DECREMENTED TO n
```

This message is the result of a RELEASE JCL statement being encountered. The HOLD=n parameter in the $JOB card indicates the number of times a RELEASE JCL statement must be read before the job can be put into the IN SCHEDULING state. The message indicates that the old count has been decremented but is not yet equal to zero for that job. Therefore the job stays in the HOLD state.

If several jobs with the same name are in the HOLD state a message appears for each one.

ron IN job-name user-name class SPR = n submitter-ron station-name

This message is the result of a RUN JCL statement being encountered. It provides the identification of the submitted job: run occurrence number, job-name, user-name, job class, scheduling priority, run of the submitter, name of the RBF station to which the job is attached.

```
@hh.mm.ss JOB DELAYED BY SHUTDOWN
```

This message is the result of an END SYSTEM session command entered by the operator. The execution of the job continues until the current step terminates. The job will restart automatically at the next step when the new session begins.

```
hh.mm.ss JOB HELD BY OPERATOR
```

This message is the result of a Hold Job (HJ) command. Depending on the job state at the time the command is entered, the job will not be selected for execution if the job execution is suspended or if the job was in the IN SCHEDULING state. The job is restarted when the operator issues a Release Job (RJ) command.

```
hh.mm.ss JOB FORCED BY OPERATOR
```

This message is the result of a Force Job (FJ) command. This command allows the job to bypass the scheduling mechanisms.

```
hh.mm.ss JOB MODIFIED BY OPERATOR.  CLASS ={class} SCH {spr}
                                          { *   }     { * }
DPR ={dpr}    SW ={switches-value}
     {*  }        {      *       }
```

This message is the result of a Modify Job (MJ) command. It indicates which job parameters have been modified, c indicates that the parameter has not been modified by the command. The operator can modify the job class, the scheduling priority (spr), the execution priority (dpr) and the switches. If the switches have been modified the new state is given in hexadecimal.

```
hh.mm.ss JOB REACTIVATED BY SYSTEM
```

This message appears when the system automatically resumes execution of a job that had been previously suspended because the system was overloaded (TDAC).

```
job-name: JOB RELEASED
```

This message is the result of a RELEASE JCL statement. It indicates that the job specified in the message has been released. When several jobs with the same name have been released by the execution of the statement, one message appears for each of them.

```
hh.mm.ss JOB RELEASED BY OPERATOR
```

This message is the result of a Release Job (RJ) command. The job that was previously in the HOLD or SUSP state is put into the In Scheduling or Executing state.

```
hh.mm.ss JOB SUSPENDED BY SYSTEM FOR TDAC
```

This message indicates that the job execution has been automatically suspended by TDAC because the system was overloaded. The job will be automatically released later.

```
JUMP {CONTINUE      }
     {DONE TO label }
```

This message traces the Jump JCL statements which have been actually executed. That is, for conditional Jumps, those for which the condition was verified.

```
hh.mm.ss TJ COMMENTS: Operator-entered-text
```

This message is the result of a Terminate Job (TJ) command. It displays on the JOR the text that has been entered by the operator to the TJ command. This text normally indicates the reason why the operator interrupted the execution.

A. RESIDENT Volumes

A.1 Introduction

The list of RESIDENT volumes is specified at ISL time. During a Warm Restart, this list can be modified temporarily, that is, for the duration of the current session. While a volume is RESIDENT, it should be processed as such and not via DEVCLASS and MEDIA. Before removing a disk from the list of RESIDENT volumes, ensure that it contains no extents of files that also have extents on other RESIDENT volumes. If these precautions are not taken, problems may subsequently arise in the processing of such multi-volume files. An example of such a problem is given below.

EXAMPLE:

1. The volume MYDISK is declared RESIDENT.
2. The library file MYLIB is allocated on MYDISK using LIBALLOC with DEVCLASS and MEDIA.
3. LIBMAINT with RESIDENT is used to load MYLIB. Extension (if necessary) of MYLIB may take place on other RESIDENT volumes (up to the maximum size specified by LIBALLOC). Suppose that MYLIB is extended onto the RESIDENT volume RESDSK. The situation is as shown in *Figure A-1*.

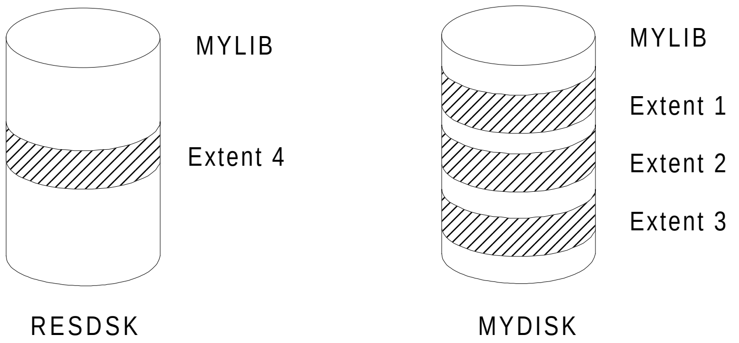


Figure A-1. MYLIB Extended onto RESDSK

4. The file MYLIB can be processed normally as long as it is treated as RESIDENT. In particular, the utilities FILDESC and FILDUPLI can be used to describe and copy the file, respectively.

5. If MYLIB is processed as a non-RESIDENT file (i.e., specifying DEVCLASS and MEDIA), the result depends on the MEDIA value given.

   a. MEDIA = (MYDISK,RESDSK)

      This will lead to "normal" processing of all the extents of MYLIB. From this viewpoint, it is equivalent to 4 above. However, it is undesirable as it relies on the user specifying an up-to-date list of volume names. Thus, if the file were subsequently extended onto a third disk volume, then this volume name would have to be added to the list.

   b. MEDIA = MYDISK

      This will lead to partial processing of the file, that is, of the extents on MYDISK.
      In particular, FILDESC indicates,

      ```
        FREE LOGICAL TRACKS USED negative
      % USED greater than 100
      ```

      both of which warn the user of the existence of extents external to MYDISK.

      FILDUPLI duplicates only those extents found on MYDISK i.e., it obeys the request to process only MYDISK.

   c. MEDIA = RESDSK

      This will lead to an attempt to process just Extent 4, which resides on RESDSK. In fact, FILDESC, FILDUPLI, and LIBMAINT will all abort as it is not possible to process Extent 4 on its own.

6. MYLIB is "partially" saved. A VOLSAVE (of MYDISK) or a FILSAVE (of MYLIB with MEDIA=MYDISK) results in a partial save of the file (not deliberately, of course!).

7. Other users continue to process MYLIB as a RESIDENT file i.e., they process the extents on MYDISK and on RESDSK. This processing includes the addition of new members and the modification/deletion of existing members.

8. MYDISK or MYLIB is restored from the save made at phase 6.

   MYLIB is now in an inconsistent state. The directory as restored (i.e., as saved at 6) is not consistent with the file. This is due to the processing done in phase 7. For example, members deleted from Extent 4 (during phase 7) are in the restored directory, while new members added to Extent 4 do not appear in the directory.

❑

NOTE:

RESIDENT disks can be considered as a pool of on-line storage and declaring a volume RESIDENT adds it to this pool. Files on a RESIDENT volume may be extended onto other RESIDENT volumes without specific user request (phase 3 above). This should be borne in mind before removing a volume from the RESIDENT pool. While RESIDENT, a volume should be processed as RESIDENT, and not by using DEVCLASS and MEDIA (phase 5 above). Violating this principle can lead to partial processing of files, and in certain circumstances lead to inconsistent files.

A.2 Recommendations

The following subsections give recommendations about making disks RESIDENT or non-RESIDENT.

A.2.1 Making a Volume RESIDENT

Ensure that the volume will in future be processed as RESIDENT only. This may mean changing the JCL of steps that process it. Processing the volume sometimes as RESIDENT and sometimes as non-RESIDENT (i.e., DEVCLASS, MEDIA) may lead to inconsistent results. This can occur when users "forget" to change their JCL.

A.2.2 Making a Volume Non-RESIDENT

Before withdrawing a volume from the list of RESIDENT disks, ensure that:

- It contains no extents of files from other RESIDENT disks
- None of its files have extents on other RESIDENT disks.

If necessary, such files can be copied using FILDUPLI (but with RESIDENT).

Ensure that the JCL of the steps concerned is changed, so that in future the volume is processed as a non-RESIDENT volume.

B. Parameter Substitution

B.1 General Concepts

The user can define a parameter value within any of the JCL statements VALUES, MODVL, INVOKE, EXECUTE or RUN (or within the operator command SJ). Each defined value is a character string, which may be protected (enclosed in single quotes). The character string replaces each corresponding parameter value reference that comes within the scope of the defining statement (e.g., each parameter value reference in a job stream introduced by RUN statement).

The substitution of a parameter value can occur anywhere within a job description (including within an input enclosure) except within a Stream Reader statement (i.e., $JOB, $ENDJOB, $INPUT, $ENDINPUT, $DATA, $ENDDATA, $SWINPUT). The system replaces parameter value references with parameter values during job translation.

B.2 Parameter Value References

There are two types of parameter value references: positional parameter value references and keyword parameter value references; they refer respectively to positional parameters and keyword parameters in the defining statement. Their functions are identical, but they differ in definition and in use.

B.2.1 Positional Parameter Value References

A positional parameter value reference consists of an ampersand character (&) followed by a one-digit or a two-digit number (maximum value 99); for example:

```
&11 &20 &03 &3
```

Note that the last two examples are exactly equivalent in meaning.

The number after the & character specifies the position of the appropriate parameter value definition within the VALUE statement, or within the VALUES parameter of the INVOKE, EXECUTE, or RUN statement; consider the following parameter value definition:

```
VALUE ONE, TWO, XYZ, Q43;
```

If the parameter value reference &2 appears after the above VALUES statement in a job description, it will be replaced at JCL translation time by the character string TWO; similarly, the string Q43 will replace each occurrence of &4.

B.2.2 Keyword Parameter Value Reference

A keyword parameter value reference consists of an ampersand (&) followed by a keyword of up to eight alphanumeric characters, the first of which must be a letter (A to Z); for example:

```
&KEYWORD &X1234567 &A1 &A01
```

Note that the last two examples are not equivalent.

For a particular keyword parameter value reference, the system substitutes the value assigned to the corresponding keyword parameter value definition within the VALUES statement, or within the VALUES parameter of the INVOKE, EXECUTE or RUN statement; consider the following parameter value definition:

```
VALUES VALIO = A, VAL56 = 14;
```

The character A will replace each occurrence of &VALIO, and the character string 14 will replace each occurrence of &VAL56.

B.2.3 General Rules for Parameter Value References

A parameter value reference can occur anywhere within a job description (except within $JOB, $ENDJOB, $INPUT, $SWINPUT); this includes, for example, within parameter names (e.g., if the defined value for &1 is E then &1ND represents END; if the defined value is ABE then &1ND represents ABEND). This situation could lead to confusion; for example, do the characters &11ND represent the value of &11 followed by ND or the value of &1 followed by 1ND?

To avoid ambiguity, the following rules apply to the way in which the system interprets the text that follows an ampersand character.

1. If the first character is a digit, there is a positional parameter value reference; in this case if the next character is a digit, it is included as the second digit of the value reference; for example:

   ```
   &1A2 - value of &1 followed by A2
   &12A - value of &12 followed by A
   &123 - value of &12 followed by 3
   ```

2. If the first character is a letter, there is a keyword value parameter reference; in this case the keyword includes every character up to the first non-alphanumeric character; if more than eight successive alphanumeric characters follow the ampersand, only the first eight are taken as the keyword.

   For example, consider the definition:

   ```
   VALUES KEYWORD = DFILE, KEYWORD1 = SFIL1;
   MODVL  KEYWORD = OFILE, KEYWORD1 = SFIL1;
   ```

   the reference &KEYWORD, becomes DFILE,
   the reference &KEYWORD1, becomes SFIL1,
   the reference &KEYWORD12, becomes SFIL12,
   the reference &KEYWORD2, becomes an empty string, (since KEYWORD2 has not been defined; see rules for Parameter Substitution below).

3. If the first character is neither a letter nor a digit, an error condition will result.

4. Two consecutive vertical bars (||, each of which corresponds to the internal EBCDIC hexadecimal value 4F and H36 card code 12-8-7) always end a parameter value reference and are omitted when the string value is substituted; the use of these special symbols avoids the addition of unwanted characters at the end of a parameter value reference. For example:

   &22AB represents the value of &22 followed by AB
   &2||2AB represents the value of &2 followed by 2AB
   &END, represents the value of &END followed by,
   &EN||D, represents the value of &EN followed by D,

NOTE:

On an IOF (Interactive Operation Facility) terminal, the character ? is used instead of each vertical bar.

B.3 Rules for Parameter Substitution

1. When the JCL Translator finds a parameter value reference, it analyzes the corresponding replacement value (i.e., the parameter string to be substituted) in terms of the required syntax of the current JCL entity (i.e., the label, keyword, statement name etc. that the translator is currently analyzing). For example, consider the definition:

   ```
   FDESC = 'DEVCLASS = MS/M400, MEDIA = C053'
   ```

   (Note that a protected string is necessary because there are characters other than letters, digits and hyphens).

   If there is a statement of the following form;

   ```
   ASSIGN INFL, MY.FILE, &FDESC;
   ```

   The Translator will remove the enclosing quotes, ensure that the DEVCLASS and MEDIA parameters are syntactically correct, and produce the following statement:

   ```
   ASSIGN INFL, MY.FILE, DEVCLASS = MS/M400, MEDIA = C053;
   ```

   If for example, MODIA had appeared instead of MEDIA, the JCL Translator would have detected an error at the time of the replacement.

IMPORTANT:
If a protected string is permitted within a particular JCL entity (e.g., for a nonstandard file name as an external-file-name), a replacement value that is a protected string is not analyzed.

Consider the definition:

```
FDESC = 'MY.FILE, DEVCLASS = MS/M400,
MEDIA = C053'
```

If there is a statement of the following form:

```
ASSIGN INFL, &FDESC;
```

The Translator will simply replace the value reference with the defined value, to produce:

```
ASSIGN INFL, 'MY.FILE, DEVCLASS = MS/M400,
MEDIA = C053';
```

The character string MY.FILE, DEVCLASS = MS/M400, MEDIA = C053 as nonstandard external-file-name.

2. An "empty string" value is associated with any reference whose value has not been defined. The JCL Translator ignores an empty string. For example, if the keyword value parameter FDESC has not been defined, the statement:

```
Becomes: ASSIGN INFL, MY.FILE, &FDESC;

ASSIGN INFL, MY.FILE;
```

B.4 Parameter Substitution in Input Enclosures

An input enclosure can contain parameter value references. The appropriate values to be substituted can be taken either from values that apply at the level of the original job or from those that apply at the level of the current invoked JCL sequence, depending on whether the appropriate $INPUT statement contains the JVALUES parameter or the CVALUES parameter.

B.5 Definition of Parameter Values

B.5.1 Types of Defining Statements

The VALUES and MODVL statements allow the user to define and redefine default values within a JCL entity (i.e., within a job description or a JCL sequence). The user can also define external values to JCL entities from the RUN, INVOKE and EXECUTE statements that introduce the entities; these external values override any values that are defined within the introduced JCL entity (by VALUES or MODVL).

Table B-2 at the end of this Appendix illustrates the effect of internal and external value definition.

B.5.2 Mixing of Positional Parameters and Keyword Parameters

Both positional parameters and keyword parameters can appear in a value definition. However, all positional parameters must be specified before any keyword parameters.

EXAMPLE:

```
VALUES A, B, C, D, KEY = E, CHECK = F, SUB = G;
```

In the above example, there are four positional parameters and three keyword parameters. The characters A, B, C, D, E, F and G will replace respectively &1, &2, &3, &4, &KEY, &CHECK and &SUB.

NOTE:

If a particular value is to be identical to the specified value reference (e.g. CHECK = CHECK), the keyword parameter does not need to appear in full but can be specified as a single self identifying value (provided that, as such, it is not recognized as a positional parameter); for example:

```
VALUES A, B, C, D, KEY = KEY, CHECK = CHECK, SUB = SUB;
```

can be rewritten as:

```
VALUES A, B, C, D, KEY = KEY, CHECK, SUB;
```

However, if KEY appeared in place of KEY = KEY, it would be considered as the fifth positional parameter, replacing &5, and not as a self-identifying parameter. These remarks also apply to the MODVL statement.

❑

B.5.3 The VALUES Statement

The VALUES statement sets default values within the current JCL entity. These values are valid until the next VALUES or MODVL statement or, if none is present, to the end of the JCL entity. The values apply to parameter references in all JCL statements except in the seven Stream Reader statements ($JOB, $ENDJOB, $INPUT, $ENDINPUT, $DATA, $ENDDATA, $SWINPUT) and in the next VALUES or MODVL statement. Each VALUES statement resets to "not defined" all default values set by a previous VALUES or MODVL, before it sets any new default values. This means that if there is no definition for a particular value reference in a VALUES or MODVL statement (and if no externally defined value applies) an "empty string" is substituted, irrespective of the value set by earlier VALUES or MODVL statements. Consider the following sequence of statements, assuming that no external values have been set.

```
VALUES RES, DOC1, C035;
.                           &1      &2    &3
.      this sets the value: RES    DOC1  C035
.

VALUES &1, DOC2,
.                             &1        &2      &3

.      this sets the values: ''empty''  DOC2  ''empty''.
```

Note that the value &1 for the first parameter is not defined, since the previous VALUES statement is not valid for this VALUES statement; the statement VALUES, DOC2; would have an identical effect.

```
VALUES'&2', NDOC, C053;

.                                  &1     &2    &3
       this sets the values:       &2    NDOC  C053
```

NOTE:

Where a value reference &1 occurs in a following JCL statement, the value &2 is substituted. Provided that a protected string is not permitted at this position (see "Rules for Parameter Substitution" above), the JCL Translator will substitute the value NDOC. If the value &2 for the first parameter had not been protected, the JCL Translator would have made the substitution immediately, to give an empty string, as in the preceding VALUES statement above.

B.5.4 The MODVL Statement

The MODVL statement is similar to the VALUES statement except that MODVL does not reset to "not defined" the default values set by preceding MODVL or VALUES statements. MODVL affects the values of those parameters that are explicitly given a new value by it. MODVL does not affect the values of parameters for which it does not supply a new value; these parameters retain the values (which may be "not defined") that they had before the MODVL statement. Consider a sequence of statements similar in effect to that examined under the VALUES statement above:

```
VALUES RES, DOC1, C035;
.                              &1     &2     &3
.     this sets the values     RES   DOC1   C035
.
MODVL  &1, DOC2;
.
.                              &1    &2     &3
.     this sets the values    RES   DOC2    C035

MODVL ' &2', NDOC, C053;
.
.                              &1    &2     &3
.     this sets the values     &2   NDOC   C053
```

NOTE:
If &1 is referenced in a subsequent JCL statement, the value &2 is substituted. If a protected string is not permitted at this position, the JCL translator will substitute the value NDOC. If &2 had been used in the MODVL statement, then the effect would have been to substitute DOC2.

```
MODVL   NIL, DOC4;
.                              &1      &2       &3
.    this sets the values   ''empty''  DOC4     C053
```

NOTE:
As MODVL does not supply a new value for &3, it retains its previous value C053. DOC4 replaces the old value of &2. The parameter &1 is explicitly set to "empty". The statement MODVL, DOC4; would have left &1 unchanged (i.e., '&2').

If the last MODVL statement above were replaced by

```
VALUES, DOC4;
                                 &1       &2      &3
then the values set would be  ''empty''   DOC4  ''empty''
```

B.6 Examples of Parameter Substitution with Values

B.6.1 VALUES Statements and Substitution

The following examples are structured as a Table (Table B-1), giving a defining VALUES statement, a statement (or part of statement) containing a parameter reference, and the version with the substituted value or values.

Table B-1. Value Definition and Substitution

Defining Statement	Value Reference(s)	New Version
VALUES MS/M400;	DEVCLASS = &1	DEVCLASS = MS/M400
VALUES RESIDENT;	&1	RESIDENT
VALUES MS, M400;	DEVCLASS = &1/&2	DEVCLASS = MS/M400
VALUES MY, FILE;	&1&2 &10&2 &1||0&2 &010&2	MYFILE FILE MYOFILE MYOFILE
VALUES 'USED A.B';	REPORT &1;	REPORT 'USED A.B';
VALUES JAN;	REPORT 'MST.'&1' -AC';	REPORT 'MST.JAN-AC';

VALUES A,B,F =MY,
FILE = AFILE;
	&10
	&1||0
	&F
	&FILE
	&F||ILE
	&FILEX
	&FILE||X
		"empty string"
		AO
		MY
		AFILE
		MYILE
		"empty string"
		AFILEX

B.6.2 The INVOKE Statement

The following paragraphs, which describe the definition and substitution of externally defined values for an invoked JCL sequence, apply equally to JCL sequences handled by EXECUTE and to job descriptions within job streams handled by RUN (or by the SJ operator command).

The INVOKE statement can define external values for a JCL sequence. These values override the default values that are defined in all VALUES statements that appear within the JCL sequence. Each time the JCL Translator finds a parameter value reference, it does the following:

1. If the corresponding value is defined by the INVOKE statement, the character string replaces the value reference.
2. If the value is not defined by the INVOKE statement, the JCL Translator examines the previous VALUES statement within the sequence; if this statement has defined a default value it replaces the value reference; if there is no defined value, an empty string is substituted.

NOTE:

1. If the corresponding value in the INVOKE statement is NIL, an empty string replaces the value reference even if a default exists within the JCL sequence.
2. If any VALUES statement within the JCL sequence contains a value reference, the corresponding value defined by the INVOKE is substituted.

Consider the following INVOKE statement:

```
INVOKE MEMBER, SYS, VALUES = (, FILE, NIL);
```

Suppose MEMBER contains the following VALUES statements:

```
VALUES ABC, XFILE,C035,MS/M400;
.
VALUES, YFILE,C035,MS/M350, &2;
```

After the first VALUES, the parameter values will be as follows:

```
&1     &2      &3         &4         &5
ABC   FILE   "empty"    MS/M400    "empty"
```

After the second VALUES, the following values apply:

```
   &1     &2       &3        &4          &5
"empty"  FILE    "empty"   MS/M350      FILE
```

B.7 Example Using External Values

Table B-2, shows the effect of the definition of external values by a RUN statement and by an INVOKE statement. The relevant statements of the original job description, those of the job introduced by RUN, and those of the JCL sequence invoked by INVOKE appear on the left of the table. The corresponding values that apply within the current JCL entity after the translation of each statement appear on the right alongside the respective statement. Where a substitution is made, the new version is shown below the relevant statement or part of a statement. Note that, where external values have been defined, the "new version" of a VALUES statement that contains a value reference does not necessarily give the values that apply currently (for example, see the contents of subfile MEM1 in *Table B-2*).

Table B-2. Substitution of External and Internal Values

Job JOBA	**&1 &2 &3 &VI**
`$JOB JOBA, ... ;`	
`VALUES A, B, C ;`	A B C -
`RUN ... , JOBS = JOBB,`	
`VALUES = (D,E,F) ;`	A B C -
`INVOKE MEM1, ... ,`	
`VALUES = (&3,,G) ;`	A B C -
`[becomes VALUES = (C,,G) ;]`	
`STEP &2, ... ;`	A B C -
`becomes STEP B, ... ;`	
`...........`	
`$ENDJOB ;`	
Job JOBB	
`$JOB JOBB, ... ;`	
`STEP &3, ... ;`	D E F -
`[becomes STEP F, ... ;]`	
`...........`	
`VALUES ,,Z,VI=X;`	D E F X
`STEP &1, &2, &3, &VI ;`	D E F X
`[becomes STEP D,E,F,X ;]`	
`...........`	
`$ENDJOB ;`	
Contents of subfile MEM1	**&1 &2 &3 &VI**
`STEP &1, &2 ... ;`	C - G -
`[becomes STEP C,, ... ;]`	
`...........`	
`VALUES M,N,O VI=P`	C N G P
`...........`	
`VALUES &2, Q,R, VI =&VI ;`	C Q G -
`[becomes VALUES , Q,R ;]`	
`...........`	
`VALUES S, &1, T ;`	C C G -
`[becomes VALUES S, C, T ;]`	
`STEP &1, &3, ... ;`	C C G -
`[becomes STEP C, G, ... ;]`	

Index

Technical publication remarks form

Title : DPS7000/XTA NOVASCALE 7000 JCL User's Guide Job Control and IOF

Reference N° : 47 A2 12UJ 03

Date : **September 1999**

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please include your complete mailing address below.

NAME : ______________________________ Date : __________

COMPANY : ______________________________

ADDRESS : ______________________________

____

Please give this technical publication remarks form to your BULL representative or mail to:

Bull - Documentation D$^{ept.}$
1 Rue de Provence
BP 208
38432 ECHIROLLES CEDEX
FRANCE
info@frec.bull.fr

Technical publications ordering form

To order additional publications, please fill in a copy of this form and send it via mail to:

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

Phone: +33 (0) 2 41 73 72 66
FAX: +33 (0) 2 41 73 70 66
E-Mail: srv.Duplicopy@bull.net

CEDOC Reference #	Designation	Qty
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
_ _ _ _ _ _ _ _ _ [_ _]		
[_ _] : The latest revision will be provided if no revision number is given.		

NAME: ______________________________ Date:__________

COMPANY:______________________________

ADDRESS: ______________________________

____

PHONE: ______________________ FAX: ______________

E-MAIL: ______________________________

For Bull Subsidiaries:

Identification: ______________________________

For Bull Affiliated Customers:

Customer Code: ______________________________

For Bull Internal Customers:

Budgetary Section: ______________________________

For Others: Please ask your Bull representative.

BULL CEDOC
357 AVENUE PATTON
B.P.20845
49008 ANGERS CEDEX 01
FRANCE

REFERENCE
47 A2 12UJ 03