UNISYS

System 80 OS/3 Job Control **Programming Guide**

OS/3 Release 14

November 1991

Priced Item

Printed in U S America
7004 4623-010

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special, or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded to Unisys Corporation either by using the User Reply Form at the back of this manual or by addressing remarks directly to Unisys Corporation, SPG East Coast Systems Documentation Development, Tredyffrin Plant, 2476 Swedesford Road, P.O. Box 203, Paoli, PA, 19301-0203, U.S.A.

UNISYS

Product Information Announcement

o New Release o Revision • Update o New Mail Code

Title

System 80 OS/3 Job Control Programming Guide

This announces the release of an update to this document.

This guide is intended for the novice programmer with a basic knowledge of data processing but with little programming experience, and the programmer whose experience is not on Unisys systems.

This update includes mode characteristic information for the 5073 magnetic tape. It also describes the SYSM64 parameter which has been added to the ASM jproc, and the BUFMODE parameter which has been added to the DD job control statement.

All other changes are corrections, deletions, or expanded descriptions applicable to items present in the software prior to this release. References to other Unisys manuals have been updated to reflect the changeover to the new 11-digit document numbering system.

You can order the update only or the complete manual with all updates. To receive only the update, order 7004 4623-010. To receive the complete manual, order 7004 4623-000.

To order additional copies of this document:

- United States customers should call Unisys Direct at 1-800-448-1424.
- All other customers should contact their Unisys Subsidiary Librarian.
- Unisys personnel should use the Electronic Literature Ordering (ELO) system.

See important notice on the back of this sheet.

Announcement only:
SAB, SAE, MB00

Announcement and attachments:
ECC3, MB01, MBB1, MBB3

System: System 80
Release 14
Date: November 1991
Part Number: 7004 4623-010

NOTICE

Please note that, in Release 14, the UP numbers of certain documents were changed to the new Unisys 11-digit document numbering system:

Old Number	New Number	Old Number	New Number
UP-8044	7004 4482-000	UP-9748	7004 4565-000
UP-8076	7004 5190-000	UP-9975	7004 4581-000
UP-8613	7004 4490-000	UP-9976	7004 4599-000
UP-8811	7004 4508-000	UP-9979	7004 4607-000
UP-8834	7004 4516-000	UP-9982	7004 4615-000
UP-8839	7004 5505-000	UP-9985	7004 5224-000
UP-8859	7004 5208-000	UP-9986	7004 4623-000
UP-8870	7004 4524-000	UP-10003	7004 5232-000
UP-8913	7004 4532-000	UP-12443	7004 5240-000
UP-8986	7004 5919-000	UP-12649	7004 4631-000
UP-9744	7004 4540-000	UP-14207	7005 3434-000
UP-9745	7004 4557-000	UP-14208	7005 3442-000

In this update, some old UP numbers are still used in references to other documents; they will be changed in the next revision of this document.

PAGE STATUS SUMMARY
ISSUE: 7004 4623-010

Part/Section	Page Number	Update Level
Cover		000
Title Page/Disclaimer		010
PSS	iii	010
About This Guide	v thru ix	000
Contents	xi thru xix	000
1	1 thru 13	000
2	1 thru 13	000
3	1 thru 23	000
4	1 thru 21 22 23 thru 50	000 010 000
5	1 thru 41	000
6	1 thru 58 59, 60 61 thru 78	000 010 000
7	1 thru 17	000
8	1 thru 15	000
9	1 thru 11	000
Appendix A	1 thru 9	000
Appendix B	1, 2	000
Appendix C	1 thru 9 10 11 thru 20	000 010 000
Index	1 thru 13	000
User Reply Form		
Back Cover		000

Part/Section	Page Number	Update Level

Part/Section	Page Number	Update Level

Unisys uses an 11-digit document numbering system. The suffix of the document number (1234 5678-xyz) indicates the document level. The first digit of the suffix (x) designates a revision level; the second digit (y) designates an update level. For example, the first release of a document has a suffix of -000. A suffix of -130 designates the third update to revision 1. The third digit (z) is used to indicate errata for a particular level and is not reflected in the page status summary.

UNISYS

System 80 OS/3 Job Control **Programming Guide**

OS/3 Release 14

April 1991

Priced Item

Printed in U S America
7004 4623-000

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special, or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded to Unisys Corporation either by using the User Reply Form at the back of this manual or by addressing remarks directly to Unisys Corporation, SPG East Coast Systems Documentation Development, Tredyffrin Plant, 2476 Swedesford Road, P.O. Box 203, Paoli, PA, 19301-0203, U.S.A.

PAGE STATUS SUMMARY
ISSUE: 7004 4623-000

Part/Section	Page Number	Update Level
Cover		
Title Page/Disclaimer		
PSS	iii	
About This Guide	v thru ix	
Contents	xi thru xix	
1	1 thru 13	
2	1 thru 13	
3	1 thru 23	
4	1 thru 50	
5	1 thru 41	
6	1 thru 78	
7	1 thru 17	
8	1 thru 15	
9	1 thru 11	
Appendix A	1 thru 9	
Appendix B	1, 2	
Appendix C	1 thru 20	
Index	1 thru 13	
User Reply Form		
Back Cover		

Part/Section	Page Number	Update Level

Part/Section	Page Number	Update Level

Unisys uses an 11-digit document numbering system. The suffix of the document number (1234 5678-xyz) indicates the document level. The first digit of the suffix (x) designates a revision level; the second digit (y) designates an update level. For example, the first release of a document has a suffix of -000. A suffix of -130 designates the third update to revision 1. The third digit (z) is used to indicate an errata for a particular level and is not reflected in the page status summary.

About This Guide

Purpose

This guide is one of a series of manuals designed to help the programmer use the Unisys Operating System/3 (OS/3).

Scope

This guide specifically describes job control and explains how to use it.

Audience

The intended audience is the novice programmer with a basic knowledge of data processing but with little programming experience and the programmer whose experience is not on Unisys systems.

Prerequisites

Anyone using this guide should understand basic structured programming techniques.

How to Use This Guide

Read the entire guide to familiarize yourself with the basic concepts it presents; then use it for reference as needed.

Organization

This guide is structured to proceed from the basic to the complex, addressing topics in this sequence:

Job Control Overview

This topic is covered in Sections 1 and 2. It tells you what job control is and how it is used by the operating system. You learn the basic concepts of a control stream and the general program logic.

Basic Job Control Programming

This topic is covered in Sections 3, 4, and 5. In these sections, you become familiar with the basic job control statements used to run your programs. You also learn about job control procedure call statements (JPROCS) that can save you coding time and reduce control stream coding errors.

Advanced Job Control Programming

This topic is covered in Sections 6, 7, 8, and 9, building on what you learned in Sections 3, 4, and 5. You learn how you can get better performance and response from the computer by using advanced job control statements that perform functions that cannot be done with the basic set. You also learn how to write JPROC definitions that you can store in the system and how you can call them when needed.

Appendixes

- Appendix A discusses and illustrates the rules used in describing job control statement formats. You also learn how you should code these job control statements.
- Appendix B contains supplementary information that increases your understanding of job control.
- Appendix C contains an alphabetical listing of all the job control statements and JPROCS and their parameters. You can use this as a quick-reference chart.

Results

After reading this guide, programmers will be able to use job control statements and JPROCS to specify to the operating system what specific work it must do.

Notation Conventions

The general conventions that apply to the coding formats presented in this guide are described in Appendix A.

Related Product Information

The following Unisys documents may be useful in understanding and implementing job control.

Note: *Use the version that applies to the software level in use at your site.*

***Integrated Communications Access Method (ICAM) Utilities Programming Guide*, 7004 4565**

This guide describes how programmers can use the utility routines provided by ICAM.

***Job Control Programming Reference Manual*, UP-9984**

This manual is a quick-reference document for programmers familiar with OS/3. It describes the job control statements and job control procedures used in a System 80 environment to communicate with job control. It also presents the procedure definition statements that allow expansion and conditional modification of the job stream when you start the job.

***System Service Programs (SSP) Operating Guide*, UP-8841**

This guide describes the system service programs. They are utility programs that support the operation and organization of the operating system. They include the SAT and MIRAM librarians, the linkage editor, the disk, tape, and diskette prep routines, and various copy routines.

***Consolidated Data Management Macroinstructions Programming Guide*, 7004 4607**

This guide describes consolidated data management (CDM), a collection of program modules that handles the movement of data between input and output devices on your system. It also describes the consolidated data management macroinstructions, which let you obtain information about the characteristics of your file or request that consolidated data management process the files you defined for your program.

***Models 8-20 Operations Guide*, 7004 5208**

This guide describes the hardware configuration of the System 80 models 8-20 and presents procedures for initializing the system. It also covers all commands and procedures used in the OS/3 environment.

***Model 7E Operations Guide*, 7002 3866**

This guide describes the hardware configuration of the System 80 model 7E and presents procedures for initializing the system. It also covers all commands and procedures used in the OS/3 environment.

***Model 50 Detailed Operations Guide,* 7004 1942**

This guide describes the hardware configuration of the System 80 model 50 and presents procedures for initializing the system. It also covers all commands and procedures used in the OS/3 environment.

***Supervisor Technical Overview,* UP-8831**

This manual presents an overview of the OS/3 supervisor and its functions for OS/3 high-level language programmers and site administrators.

***Supervisor Macroinstructions Programming Reference Manual,* UP-8832**

This manual describes, for the assembler programmer, the OS/3 supervisor macroinstructions used for program management, file space management, file access, multitasking, and spooling. It also provides formats and coding conventions for the macroinstructions, diagnostic and debugging information, and examples of macroinstruction use.

***Models 8-20 Installation Guide,* 7004 5505**

This guide provides the system administrator with the information and procedures needed to install, tailor, and maintain OS/3 software in a System 80 models 8-20 environment.

***Model 7E Installation Guide,* 7002 3858**

This guide provides the system administrator with the information and procedures needed to install, tailor, and maintain OS/3 software in a System 80 model 7E environment.

***Model 50 Installation Guide,* 7004 1892**

This guide provides the system administrator with the information and procedures needed to install, tailor, and maintain OS/3 software in a System 80 model 50 environment.

***Interactive Services Operating Guide,* UP-9972**

This guide describes procedures used to communicate with the operating system interactively through a local workstation or remote terminal. It also describes the procedures for logging on and off the system and performing various interactive commands.

***File Cataloging Technical Overview,* 7004 4615**

This overview describes the OS/3 file cataloging facility in a System 80 environment for the system administrator or programmers who are authorized to control use of the system catalog file.

***Spooling and Job Accounting Operating Guide,* 7004 4581**

This guide describes basic spooling and job accounting concepts.

***Screen Format Services Technical Overview,* UP-9977**

This overview describes how programmers can use screen format services to create and maintain formatted screen displays to be used with their application programs.

***Menu Services Technical Overview,* UP-9317**

This overview describes the procedures for creating and using menus. It also describes how menus, displayed on the workstation screen, can be used with assembler, COBOL, RPG II, and FORTRAN IV™ programs.

***Dialog Processor Programming Guide,* UP-8858**

This guide provides the experienced programmer with information on the dialog processor, which is the interface between the dialog (written in dialog specification language) and the application program using the dialog.

***Distributed Data Processing Programming Guide,* 7004 4508**

This guide describes OS/3 distributed data processing and the various distributed data processing program products.

***General Editor (EDT) Operating Guide,* 7004 4599**

This guide describes the commands and procedures needed to use the OS/3 general editor to copy files, concatenate files, and create and modify library modules and data files interactively from a workstation.

***Consolidated Data Management Programming Guide,* UP-9978**

This guide describes consolidated data management and how it moves data between peripheral devices and programs.

***Assembler Programming Guide,* 7004 4532**

This guide describes the OS/3 basic assembly language (BAL) and its use. Included are general language concepts, assembler instructions, and programming techniques.

***Data Utilities Operating Guide,* 7004 4516**

This guide provides the information needed to use the data utilities. Included are instructions on executing data utilities interactively and for batch jobs.

FORTRAN IV is a trademark of SuperSoft Associations.

Contents

Section 3. Minimum Control Stream Requirements

Section 4. Getting the Most Out of the Basic Job Control Statements

Section 5. Doing It the Easy Way - with Procedure Calls

Section 6. Making Job Control Work for You

Section 7. Run-Time Conditional and Set Symbol Job Control Statements

Section 8. How to Write and Call a Job Control Procedure Definition

Section 9. Using the Interactive Job Control Dialog

Appendix A. Statement Conventions

Appendix B. Operation Considerations

Appendix C. Job Control Statement Formats

Index

User Reply Form

Figures

Tables

Section 1
Overview

Why You Need Job Control

To process any program, the operating system must have some necessary instructions and information. Should the system compile, link edit, or execute a program? Does it know what files a program uses, which devices to reserve, and how much main storage a program needs? Should it allocate space for a file? For the operating system to know what specific work - what job - you want it to do and how, you must supply this type of information to that part of OS/3 called *job control.*

To communicate with job control, you use OS/3 *job control language* (JCL), which consists of job control statements and job control procedures (JPROCS). The statements and JPROCS you code make up a job control stream.

Job Control Statements and Job Control Streams

Each of the many job control statements has a different function but they are combined in a control stream to do a singular job. OS/3 requires that every job have a control stream. Using three statements, // JOB, // EXEC, and /&, we can show you the following *outline* job control stream required for executing a program:

Job control stream for executing a program

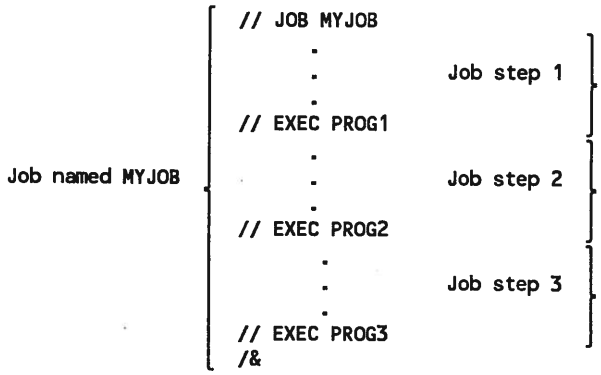
```
// JOB MYJOB          Identifies your job and indicates
      .                the beginning of the control stream.
      .
      .
// EXEC PROG1         Specifies execution of the program PROG1.
      .
      .
      .
/&                    Indicates the end of the control stream.
```

These three statements illustrate the idea of a job control stream, but you'll see in later sections that you must also include statements identifying files and devices. Additional statements are used, depending on the specific function needed to accomplish your job. You can also include program data in the control stream.

In this guide, we'll explain the function of each job control statement and its parameters so you can build simple as well as complex job control streams.

Job Steps

Any job can have one or more steps. If, for example, you want to execute three programs, one after the other, you can construct one job control stream with three (job) steps like this:



A job can have up to 254 job steps. The steps are processed serially and the EXEC job control statement normally marks the end of each one.

Job Control Procedures (JPROCS)

Besides using individual job control statements in your control stream, you can use *job control procedures* (JPROCS).

A JPROC is a series of job control statements that performs a certain function or routine. JPROCS are supplied as part of the system and you can also write your own. They are filed in a library and each JPROC has its own name. (See "Building and Storing Job Control Streams and JPROCS" in this section.) When referenced by that name in a job control stream, the statements that make up the JPROC are generated and incorporated into the control stream.

You may frequently need some function that a specific group of job control statements performs. Instead of coding the same group of statements in every job control stream requiring that function, you can simply define the statements as a JPROC, then code the JPROC name.

Compiling a source program, for example, is something that's done often. If you include a certain system supplied JPROC name in your job control stream, all the statements necessary for the language processor to compile your source program are generated. The following simplified control stream specifies the COBOL language processor JPROC.

```
// JOB MYJOB
      .
      .          Causes the generation of job control
      .          statements that identify files and
// COBOL           devices needed by the COBOL language
      .          processor.  Executes the language
      .          processor so that a source program can
      .          be compiled.
/&
```

System-supplied and user-written JPROCS are explained in Sections 2 and 3.

Job Control and the Operating System

To better understand what job control does, it helps to know where job control fits into the operating system.

Unisys Operating System/3 (OS/3) is divided into two parts: the *executive* and the *system support software components*. Job control is part of the executive portion of OS/3. Together, the supervisor and job control manage job processing for OS/3. Figure 1-1 shows the executive and system support software components of OS/3.

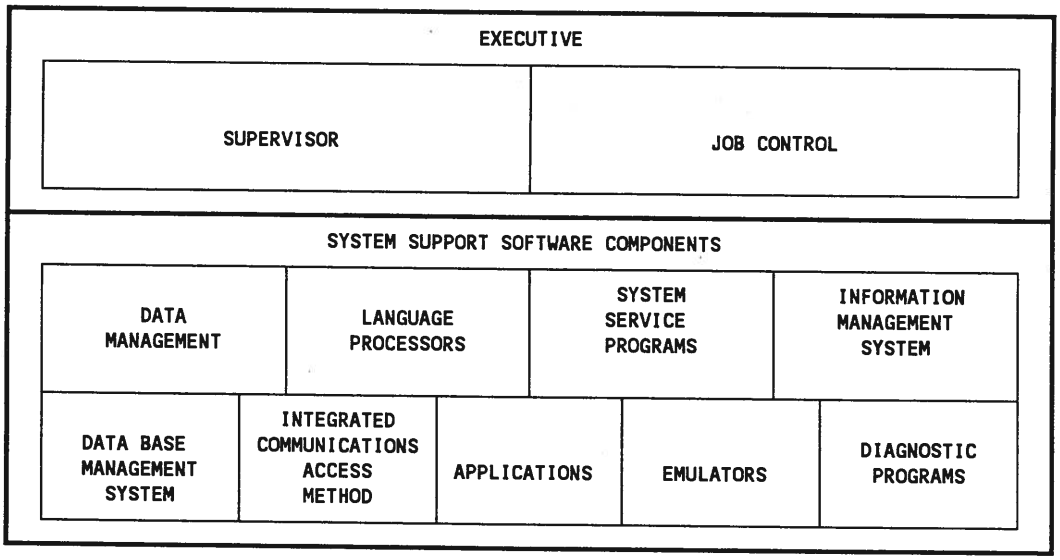


Figure 1-1. Operating System/3 (OS/3)

The supervisor controls the sequence and position of your programs and system programs in main storage. For more information on supervisor facilities, see the *Supervisor Technical Overview*, UP-8831.

Job control manages system facilities and prepares the system for job execution. In general, it does the following:

- Assigns a job number to every active job and symbiont
- Analyzes the job control stream
- Checks the order and syntax of control statements
- Expands job control procedures (JPROCS)
- Schedules jobs and queues them according to priority
- Allocates peripheral devices and main storage

These and some of the other functions that job control is responsible for are handled by (system) programs called *symbionts*. Symbionts are normally executed in response to a user request that may be in the form of a system console command, a workstation command, or certain job control statements. Symbionts compete for main storage and CPU time along with your jobs. The run processor, which begins processing your job control streams, is a symbiont. We'll be discussing the run processor in the next section.

Processing a Job Control Stream

You can build job control streams on disk, data-set-label diskette, or on cards. Looking at Figure 1-2, you can see that job processing involves several steps.

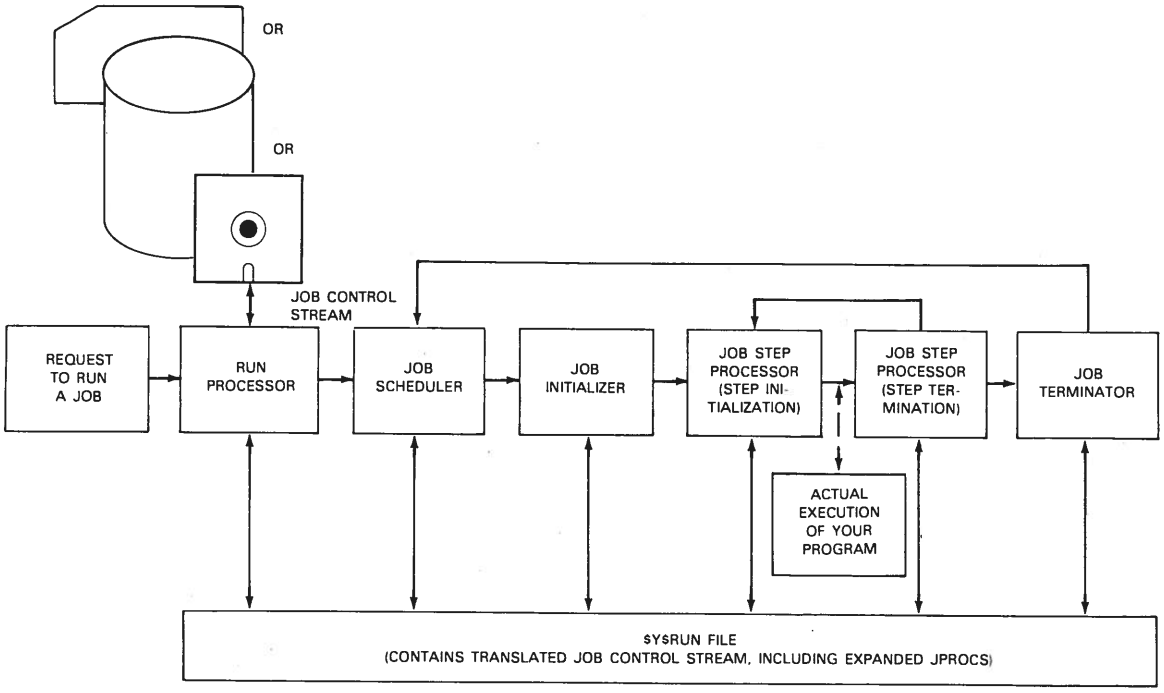


Figure 1-2. Job Processing Flow

A brief discussion of each step in the job processing flow should give you a general idea of what happens after job control accepts a request to process a job.

Beginning Job Processing - the Run Processor

The run processor begins job processing by scanning the control stream, translating the job control statements into tables on disk, and expanding JPROCS. At this point, it also checks the stream for order and syntax errors. If there are errors, no further preparation of the job is made and job control error messages are generated.

Once the control stream is translated, the run processor places it in the YRUN system file (a YRUN file is created for every job being processed). The name of the job (obtained from the // JOB statement) is entered in a table called the job queue table. The job queue table contains the names of all jobs waiting to be executed. The jobs are ordered by a priority specified on the JOB statement (or, as you'll see later, on other job control statements or workstation/console commands). Within a particular priority, the jobs are ordered on a first-in, first-out basis.

RUN PROCESSOR

- Translates job control statements
- Expands JPROCS
- Checks order and syntax of control stream
- Builds control blocks
- Enters job name in job queue table
- Creates YRUN file

Considering Jobs for Execution - the Job Scheduler

After the run processor prepares your job control stream, processing control passes to the job scheduler, which checks the job queue table. If there are jobs in the queue table, the scheduler determines which jobs will be executed next. The job priority and the availability of system resources (peripheral devices and main storage) are the basis for this determination.

A job can have one of four priorities: preemptive, high, normal, or low. At any one time, the job queue table can contain the names of up to 15 preemptive priority jobs, 39 high priority jobs, 71 normal priority jobs, and 15 low priority jobs. The job scheduler considers preemptive jobs for execution first, followed by high, normal priority, and low jobs (in that order). Jobs are considered within each priority level on a first-in, first-fit basis. Lower priority jobs are not considered until there are no other higher priority jobs in the job queue table. Jobs in HOLD status are not considered at all.

Before job execution can start, sufficient main storage and the necessary peripheral devices must be available. The job scheduler checks for both; and if both are not available, the job is left in the job queue table. A slightly different situation exists if roll-out is configured with the system. (See "Job Roll-Out/Roll-In" in Section 2.)

In addition to checking priority and the availability of main storage and peripheral devices, the job scheduler maintains the shared code directory, reserves volumes, maintains a volume use table for all jobs, deletes your job name from the job queue table, and displays your job name at the system console.

JOB SCHEDULER FUNCTION

- Considers your job for execution by priority
- Reserves devices and main storage for your job so that job execution can begin
- Deletes the job name from the job queue table
- Displays the job name on the system console

Beginning Job Execution - the Job Initializer

Processing control passes to the job initializer when job execution is ready to begin. Up to 47 jobs can be executed concurrently.

The job initializer also loads shared code modules, activates job accounting, and updates job log status.

JOB INITIALIZER FUNCTION

- Builds job preamble
- Loads shared code modules
- Activates job accounting
- Updates job log status

Initializing a Job Step - the Job Step Processor

The job step processor performs the functions necessary for initializing and completing a job step. At this point in job processing, the program specified on the EXEC statement is loaded and executed.

JOB STEP PROCESSOR FUNCTION (STEP INITIALIZATION)

- Reviews volume requirements
- Reviews device allocation
- Updates system volume use table
- Allocates devices and disk space
- Locates and updates file control blocks
- Locates and posts address of embedded data
- Stores logging data
- Performs utility functions (rewinding tapes, scratching files, etc.)

Ending the Job Step - the Job Step Processor

The job step processor also performs the end-of-job-step housekeeping duties. If this is the last step in the job, the job step processor passes processing control to the job terminator; if not, it retains processing control for initialization of the next job step.

JOB STEP PROCESSOR FUNCTION (STEP TERMINATION)

- Updates job preamble
- Initiates burst mode printing of spool files
- Records logging data
- Scratches job step (temporary) work files

Ending the Job - the Job Terminator

When the last step in the job has been processed, the job terminator receives control to perform end-of-job housekeeping duties.

JOB TERMINATOR

- Deletes job name from system console
- Scratches job temporary files
- Scratches job's YRUN file
- Requests printing of log and spool files
- Displays job termination message
- Frees memory and releases devices
- Clears job entries from system volume use table

Building and Storing Job Control Streams and JPROCS

Here are some ways you can build and store job control streams:

- If you have UDS-200 data entry equipment, you can use it offline to place job control statements directly onto data-set-label diskettes.
- You can use the general editor (EDT) to build control streams at a workstation. Depending on the instructions you give the editor, the control stream can then be placed on data-set-label diskette, in the spool file, in a permanent job control stream library on disk or format-label diskette, or on cards. You can specify a permanent SAT library of your own as the stream's destination, or you can use YJCS, the system job control stream library. The *General Editor (EDT) Operating Guide*, 7004 4599, explains the use of the general editor.
- You can use the job control dialog to build control streams. The dialog stores the completed stream in YJCS. Section 9 explains the interactive job control dialog.
- If the control stream is already on data-set-label diskette, in the spool file, or on cards, you can use a FILE system console command or the FILE workstation command to place the stream in a permanent SAT library. The FILE system console command is explained in your operations handbook and the FILE workstation command is discussed in the *Interactive Services Operating Guide, UP-9972.*

For JPROCS to function as intended, you must store them in YJCS or your own SAT library. So whether you use EDT or the job control dialog, the eventual destination of the JPROC is a permanent library. See "How JPROC Definitions Are Stored" in Section 8 for more information on storing JPROCS.

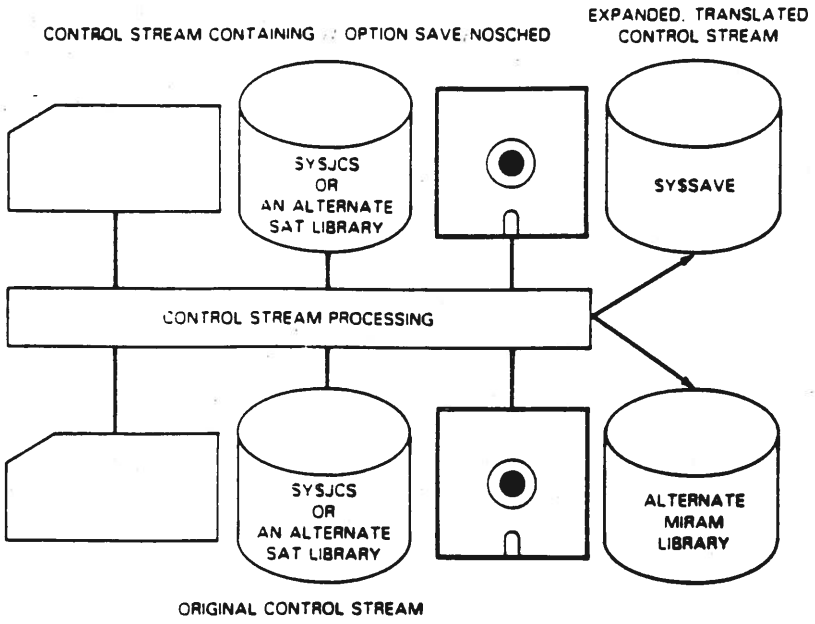
Saving Translated, Expanded Job Control Streams (Save/Restore Facility)

Before a job can be executed, no matter how often its been executed already, it must be translated and have any JPROCS expanded first. This is done by the run processor, and for some jobs (especially those with many JPROCS) this takes a long time. You can reduce this time by saving the control stream in its translated, expanded state. Because the run processor can skip the step of translating and expanding this type of control stream when it is restored and job processing starts, the job's execution begins sooner.

To save a job control stream in its translated, expanded state, you simply include the // OPTION SAVE or // OPTION NOSCHED statement in the control stream. (See "Selecting Optional Features" in Section 6.) When job processing is initiated and the run processor finishes expanding and translating the control stream, a copy of the stream (as it appears in YRUN) is placed in a permanent MIRAM library. You can specify your own library or you can use the system library YSAVE.

Depending on which OPTION statement you used, processing then proceeds through execution (OPTION SAVE) or stops as soon as the expanded, translated stream is placed in the specified library (OPTION NOSCHED). In either case, you'll have a copy of the expanded stream in a permanent library.

When a translated stream is processed, the OPTION SAVE/NOSCHED statement is ignored. If you intend to process the untranslated stream, you should remove the OPTION SAVE/NOSCHED statement. A command different from the one used to initiate processing of the untranslated stream is used for the translated one. See "Running Job Control Streams" later in this section for more information.



When deciding whether or not to save expanded, translated control streams, keep the following in mind: these streams take up more disk space than untranslated ones, you can't use them to update a file catalog (see "File Cataloging" in Section 6), and you can't change parameters on any of the job control statements. Replacing embedded data sets is the most extensive change you can make to these streams (see "Replacing Embedded Data Sets in Expanded Control Streams" in Section 6). Remember, you cannot use a hyphenated job name if you intend to save your translated control stream; the save processor does not recognize the hyphen.

Running Job Control Streams

Running a job control stream is a term commonly used in place of *processing* a control stream. In OS/3 there are several ways you can initiate the running of a control stream. These include the RV/RUN system console and workstation commands, the // RV/RUN job control statements, the SC/SI system console and workstation commands, and the // CC SC/SI job control statements. The differences between these commands and statements are summarized as follows:

- RV system console or workstation command

  This command initiates a stored control stream from a workstation or system console that does not need an input device.

- RUN system console or workstation command

  This command initiates a job control stream from a workstation or system console that needs an input device. This may mean the control stream to be run is on a data-set-label diskette, in the spool file, or on cards. It may mean the control stream is stored in YJCS or an alternate SAT library file but contains a CR job control statement and, therefore, will need an input device to complete processing. (See "Adding Cards to a Stored Control Stream" in Section 6.)

- // RV job control statement

  This statement, when encountered in an executing job control stream, initiates the running of another control stream.

- // RUN job control statement

  This statement, when encountered in an executing job control stream, initiates the running of another control stream. You can use // RUN if the control stream is on cards or is stored in a library but contains a // CR statement because card input is needed to complete job processing.

- SC system console or workstation command

  This command initiates an expanded, translated control stream (stored in YSAVE or an alternate MIRAM library) that does not require replacement of embedded data and, therefore, does not need an input device.

- SI system console or workstation command

  This command initiates an expanded, translated control stream from YSAVE or an alternate MIRAM library that needs an input device for the replacement of embedded data.

- // CC SC job control statement

  This job control statement, when encountered in an executing control stream, initiates an expanded, translated control stream from YSAVE or an alternate MIRAM library that does not require replacement of embedded data and, therefore, does not need an input device.

- // CC SI job control statement

  This job control statement is used the same as // CC SC except that it initiates an expanded, translated control stream from YSAVE or an alternate MIRAM library requiring an input device for the replacement of embedded data.

For information about system console commands, see the appropriate operations guide. For information about workstation commands, see the *Interactive Services Operating Guide*, UP-9972. For information about the // CC SC/SI and // RV/RUN job control statements, see, respectively, the "Using the RV/RUN Job Control Statements to Call Control Streams" and "Using CC SC/SI to Call Saved Translated Control Streams" in Section 6.

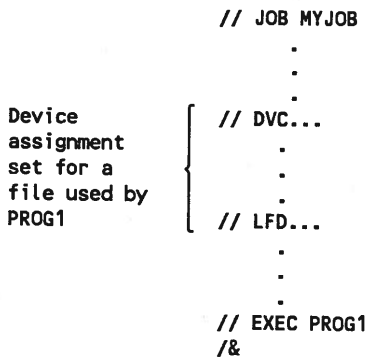
Section 2
Basic Concepts

Assigning Devices and Files

An important part of writing a job control stream is identifying devices and files and establishing a logical connection between the files and the program using them. The following job control statements help you do this:

DD	EXT	LFD	UID	VOL
DST	LBL	ROUTE	USE	
DVC	LCB	SPL	VFB	

The DVC and LFD statements (in that order) are required for every type of file and device you use. The other statements (when used) must appear between the DVC and LFD statements. They're necessary depending on the kind of file, or function you want performed in relation to that file. As a group, these statements are called a *device assignment set.*



The CAT, DECAT, EQU, FREE, REN, and SCR job control statements are not coded between the DVC and LFD statements; so, technically, they're not part of a device assignment set, but their function is related. We'll talk about these in later sections. In this section, we'll talk about the DVC, VOL, LBL, EXT, and LFD job control statements to help you become familiar with the overall function of a device assignment set.

Peripheral Devices and Logical Unit Numbers (DVC Statement)

A peripheral device is any unit of equipment, distinct from the central processor and main storage, that allows the system to send or receive data. Some devices, such as card readers, only handle incoming data (input); some, such as printers and card punches, can only handle outgoing data (ouput); while others, such as disks, format-label diskettes, tapes, and workstations, can handle both (input and output).

In OS/3, each type of peripheral device is assigned a specific number called a *logical unit number.* You specify logical unit numbers in the DVC job control statement. This tells job control (the job scheduler) which peripheral devices you need for your job.

Suppose you need a printer because your program produces printed output. The following information (taken from Table A-1 of the *Job Control Programming Reference Manual*, UP-9984) shows some logical unit numbers for printers.

Device Type Code	Logical Unit No.	Device Type and Features
04040000	14, 15	0791 correspondence quality printer
04010000	16, 17	0798 printer, no optional features
04020000	18, 19	0789 printer
04FF0000	20, 21	Any printer, no features specified
04400000	22, 23	9246 printer, no features specified
04100000	24, 25	0776 printer, no optional features
04200000	26, 27	AP9215 printer, no features specified
04800000	28, 29	0770 printer, no optional features

If you need a Unisys 0776 printer, specify either 24 or 25 on the DVC statement. If any printer will do, specify 20 or 21.

```
Device assignment      // JOB MYJOB
for the 0776          { // DVC 24 }
printer               { // LFD... }
                        // EXEC PROG1

Device                // JOB MYJOB
assignment            { // DVC 20 }
for any               { // LFD... }
available               // EXEC PROG1
printer
```

Each logical unit number you use corresponds to a device requirement for your job. So, if you specify logical unit number 20 in one job step and logical unit number 21 in a following step, two printers must be available in order for your job's execution to begin, even if one is sufficient.

```
// JOB MYJOB
   .
   .
   .
// DVC 20
// LFD...
      .
      .
      .
// EXEC PROG1
      .
      .
      .
// DVC 21
// LFD...
      .
      .
      .
// EXEC PROG2
/&
```

Two printers must be available for this job to run.

Besides using logical unit numbers, disk devices can be assigned by specifying RES or RUN. These and other functions of the DVC statement are further discussed in Sections 3 and 4.

Note: *For 0776 printers, the CLASS=parameter should be used if a unique logical unit number is required.*

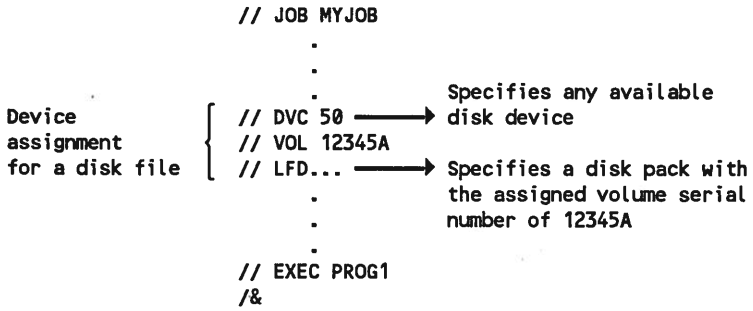
Volume Serial Numbers for Disk, Diskette, and Tape (VOL Statement)

Volume serial numbers are used to uniquely identify disk packs, diskettes (format and data-set-label), and tape reels to the operating system. This number is written externally (generally on a gummed label) and internally (on the actual recording surface). Both numbers should match for identification purposes.

The assignment of volume serial numbers takes place when the prep routines associated with disk, diskette, and tape are performed. See the *System Service Programs (SSP) Operating Guide*, UP-8841, for information about prep routines.

When you specify a volume serial number in a VOL statement, job control checks to make sure that a tape reel, diskette, or disk pack with the matching volume serial number is mounted. If the wrong volume is mounted, the system notifies the operator.

In this example



the disk volume whose serial number is 12345A must be mounted for job processing to continue.

We'll discuss other functions of the VOL statement in Sections 3 and 4.

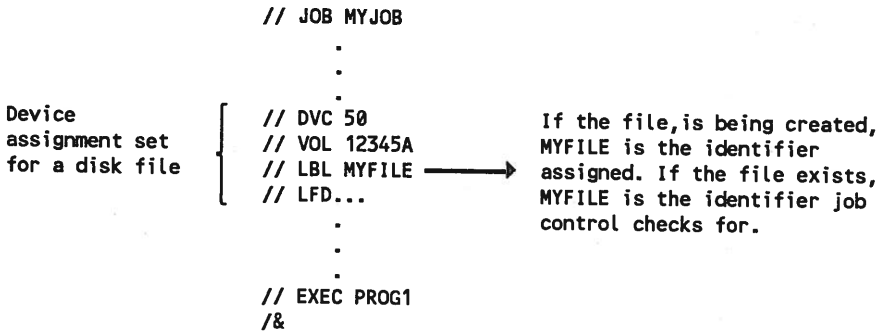
Notes:

1. *OS/3 assumes that all volume serial numbers are unique. The mounting of two volumes with the same volume serial number at the same time yields unpredictable results.*

2. *OS/3 allows a maximum of 151 volumes to be in use by all active jobs. (The maximum number of volumes allowed for a single job is also 151.)*

File Identifiers (LBL Statement)

While a volume serial number identifies one tape, disk, or diskette volume, a file identifier names (or identifies) a particular file on that volume. The file identifier is an alphanumeric name physically written on the recording surface of the tape, disk, or diskette (format and data-set-label). You specify a file identifier on the LBL job control statement. If you're creating the file, the identifier you specify is assigned. If the file already exists, job control checks to see that the file identifier specified with the LBL statement matches one already recorded for a file on a particular volume. This ensures correct file use.



A file identifier specified on an LBL statement is required for any file on disk, diskette, or multifile tape volume. If a tape volume holds only one file, a file identifier may be specified but isn't required. As you'll see in a later section on spooling card input, it is sometimes useful to specify an LBL statement (with a file identifier) in the device assignment set for a card file that's been spooled.

The LBL statement has other functions that are covered in Sections 3 and 4.

Note: *The prep routine for data-set-label diskette automatically assigns a file identifier of DATA unless you specify otherwise during the prep.*

Disk and Format-Label Diskette File Area (EXT Statement)

Whenever you're creating a disk or format-label diskette file, you allocate space for that file in contiguous areas (on the recording surface) called extents. The amount of space as well as other characteristics of the file's extent are specified using the EXT job control statement. The device assignment set for every disk or format-label diskette file you are creating must include an EXT statement. It is also required if you want to change certain extent specifications for a file that already exists.

Using the EXT statement, space on disk or format-label diskette is allocated in terms of one of the following:

- Number of cylinders

  You specify the number of cylinders needed for the file.

- Absolute cylinder address

  You specify the number of cylinders needed for the file and you also specify the starting address of the file as an absolute cylinder address.

- Number of tracks

  You specify the number of tracks needed for the file.

- Absolute track address

  You specify the number of tracks needed for the file and you also specify the starting address of the file as an absolute track address.

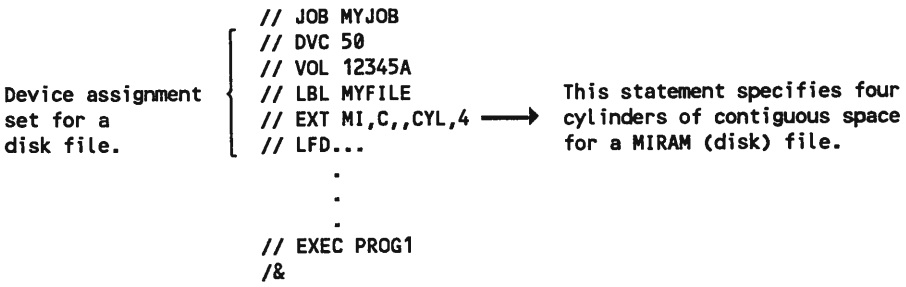
- Number of blocks (by cylinder)

  You specify the number and average length of the blocks needed for the file. Job control converts this specification to the number of cylinders so the actual allocation is by cylinder.

- Number of blocks (by track)

  You specify the number and average length of the blocks needed for the file. Job control converts this specification to the number of tracks so the actual allocation is by track.

You'll learn more about file space allocation when we discuss the EXT statement in Sections 3 and 4. For now, it is enough to know that an EXT statement must be included in the device assignment set when you're allocating space or making certain allocation changes for a disk or format-label diskette file.



Data-Set-Label Diskette File Area (EXT Statement)

The prep routine for a data-set-label diskette automatically allocates the entire diskette for one file and assigns a file identifier of DATA unless you specify otherwise. If the space was already allocated by the prep routine, there is no need for you to include an EXT statement in your device assignment set. If, however, the space was not previously allocated, you must use the EXT statement to allocate the space yourself. Allocating the space yourself allows you to have control over how many files the diskette can contain.

Space on data-set-label diskette must always be allocated by block and it must be contiguous. Data-set-label diskette files are always one-extent files. For information about the EXT statement for data-set-label diskette, see "Information about Data-Set-Label Diskette File Allocation" in Section 4.

Logical File Names (LFD Statement)

We've already talked about how you specify a file identifier (a name that's physically recorded on the surface of a disk, tape, or diskette) on the LBL job control statement. There is another name, however, that is required for every file (not just disk, tape, and diskette) and must be included in every device assignment set. It is the logical file name: the name your program references the file by.

You specify it on the LFD (logical file definition) job control statement, which is always the last statement in any device assignment set. The name you specify logically (LFD) links the file (name) you reference in your program with the physical file (LBL) defined in your job stream's device assignment set. The names that you use are:

- In BAL

  The name from the label field of the file definition macroinstruction.

```
If:                          Then:

1        10      16
-------------------
FILE1    CDIB                // DVC 50          ]
                             // VOL 12345A      }  Device assignment
                             // LBL MYFILE      }  set for a newly
                             // EXT MI,C,,CYL,4 }  allocated file
                             // LFD FILE1       ]  referenced by the
                                                   program as FILE1
```

- In COBOL

  The LFD field of the implementor name from the SELECT clause.

```
If:                                        Then:

12
-----------------------------------------
SELECT CDS ASSIGN TO CARDREADER-INFIL-F     // DVC 30    ]  Device assignment set
                                            // LFD INFIL ]  for the card file
```

(In basic and extended COBOL, the LFD name corresponds to the first eight characters of the file name from the SELECT clause. If an external name is specified, however, then use the external name instead.)

- In FORTRAN

  The device number from the READ or WRITE statement, prefixed by FORT.

  If:

  ```
  1         7  10
  ------------------
            READ(6,10)
  ```

  Then:

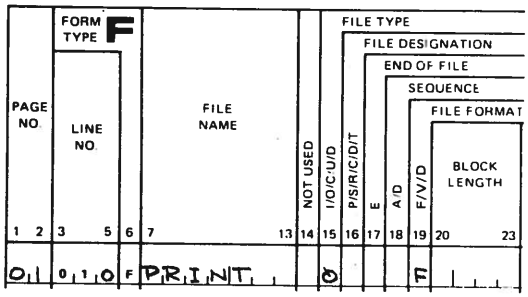
  ```
  // DVC 90
  // VOL TAPE01
  // LBL PAYFIL
  // LFD FORT6
  ```

  Device assignment set for a tape file

- In RPG II

  The file name from the file description specification.

  If:

  Then:

  ```
  // DVC 20
  // LFD PRINT
  ```

  Device assignment set for a print file

The file name used for a printer file in programs supplied by Unisys (such as the compilers and the linkage editor) is a standard system file name, PRNTR. So, if you want the printed output from a compilation, for example, the LFD statement for the print file device assignment set is // LFD PRNTR. This logical file name applies only to programs supplied by Unisys. In a job or job step that executes a user program, you must supply your own logical file names (for the printer plus any other files) on the LFD job control statement.

When using any other Unisys routines (such as the data utility routines), specify the standard system file names shown in the coding examples in the corresponding publication.

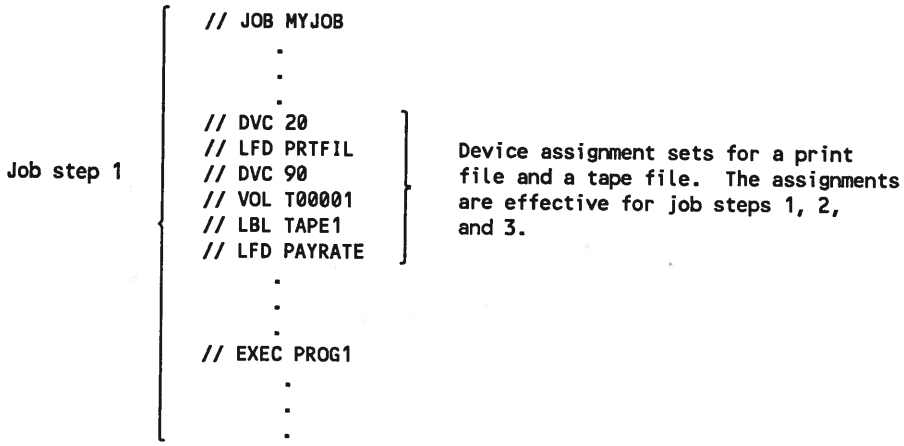
These and other applications of the LFD statement are discussed in Sections 3 and 4.

Device Assignment Set Placement and Duration

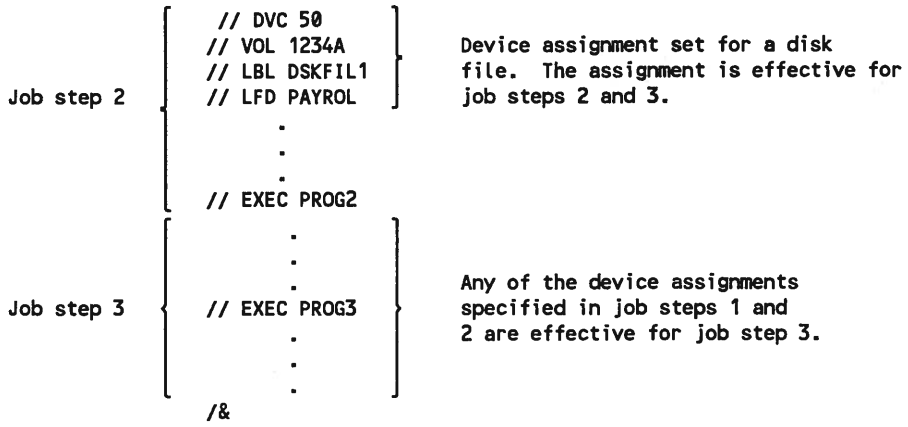
There is no strict rule for the placement of a device assignment set in a job control stream: simply place the device assignment set somewhere between the JOB statement and the EXEC statement.

```
// JOB MYJOB
      .
      . <---------- other job control statements
      .
// DVC 50
// VOL 12345
// LFD DSKFIL1
// LFD PAYROLL
      .
      . <---------- other job control statements
      .
// EXEC PROG1
/&
```

Where a multiple step job is concerned, just remember that a device assignment set specified in one job step is normally effective for that step as well as any that follow. Consider this example.



continued



In the preceding example, PROG1 can reference only PRTFIL and PAYRATE. It cannot reference PAYROL. PROG2 and PROG3 can reference PRTFIL, PAYRATE, and PAYROL.

Job Termination

There are two ways in which a job can terminate: normally or abnormally.

1. Normal Termination

   This is initiated by the control stream, the program, or the workstation or system console operator. Generally, it occurs after the last job step, but it can also be caused by the operator using the CANCEL or STOP operator command, or by the program issuing a cancel instruction. If terminated by the CANCEL system command or program instruction, the entire job terminates immediately. This includes the currently executing job step plus all subsequent job steps (if any) in the job. The STOP operator command terminates a job when the job step currently executing is finished.

2. Abnormal Termination

   This is caused by program errors or by control stream errors (syntax order). If caused by program errors, you can get a main storage printout (dump), which can be used to debug your program, provided that you have placed an OPTION DUMP statement in the control stream prior to the job step that caused termination. The OPTION job control statement is covered in "Selecting Optional Features" in Section 6. If caused by a control stream error, a message explaining the error is displayed on the system console.

In anticipation of program errors, you may use the *ABNORM=label* parameter of the EXEC statement. This parameter causes a skip forward in the job control stream so that the job finishes executing and doesn't terminate abnormally. If, however, the operator issues a cancel instruction, the job terminates normally.

All terminations result in the deallocation of the system facilities (peripheral devices, main storage, disk work areas, etc.) allocated to the job.

Restarting a Job

What if your job terminates abnormally - specifically when your program is executing? If the program only processes a few records, you can rerun the job from the beginning without any great loss; but, if the program processes many records, rerunning the job increases processing time and cost. To help avoid this, OS/3 provides a restart facility that permits you to resume execution of your job from a particular job step or a checkpoint record. See "Restarting a Job" in Section 6 for more information.

Branching within a Control Stream

When you write a program, you can set alternate paths for the program to take. Normally, program statements execute consecutively in the order of their appearance. However, it is often necessary to alter this normal sequence and skip forward to a different point in the program - this is called branching. Similarly, alternate paths can be taken in job control streams. The SKIP and OPTION QUERY job control statements allow you to skip forward in the job control stream during your program's execution to another job control statement. The ABNORM parameter of the EXEC job control statement allows you to skip forward in the job control stream if your program causes an abnormal termination. (See Section 6.)

You can also branch from one job control statement to another in a control stream by using run-time conditional job control statements (they're called run-time statements because they are available and effective through the run symbiont). Run-time conditional job control statements are interpreted and acted upon while the run symbiont is scanning the control stream. They are not placed in the job's YRUN file; their actions are completed when the run processor has acted upon them. Only forward branches are allowed. The job control statements belonging to this category are GO, IF, and NOP. They are explained in "Run-Time Conditional Job Control Statements" in Section 7.

Jobs and Main Storage

After the supervisor is loaded into the system, the remaining main storage is available to job control, symbionts (like the run processor and the job scheduler), your jobs, shared code, and your programs. Naturally, the amount of available main storage varies, depending on the jobs, symbionts, and programs executing at the time. Job control assigns a portion of main storage to each job as the space becomes available. The amount of main storage assigned is that which is needed to execute the largest job step in the job. When a job is completed, the space it occupied is returned to the system.

Job Roll-Out/Roll-In

In "Considering Jobs for Execution - the Job Scheduler" in Section 1, we mentioned that the job scheduler considers jobs for execution by priority and the availability of main storage and peripheral devices. In general, if the necessary main storage and peripheral devices are not available, the job's execution, regardless of its priority, cannot begin. A different situation exists if roll-out (ROLLOUT=YES) is configured at SYSGEN time.

With roll-out, high, normal, and low priority jobs are rolled out to disk to provide enough main storage for preemptive jobs to be executed. When the preemptive priority section of the job queue table is empty, the job scheduler rolls first the high, then the normal, and last the low priority jobs back into main storage for execution. Remember though, even if roll-out is configured, the peripheral devices needed for the preemptive job must also be available; otherwise, roll-out does not occur.

Minimum and Maximum Main Storage

By minimum main storage size we mean the amount needed to successfully execute the largest step of a job. The maximum size is the amount that can be used, if available, to improve or speed up job step execution. As you'll see in Section 4, you can specify the minimum and maximum main storage size on the JOB statement or on the OPTION statement.

The total amount of main storage used by a job step also includes the size of the job prologue. The prologue contains information (control tables) needed to regulate your job. The size of the prologue, however, is automatically taken into consideration so you don't have to include it in any main storage size that you specify. Just keep in mind that the job prologue is part of the true main storage requirement for a job. This is illustrated in Figure 2-1.

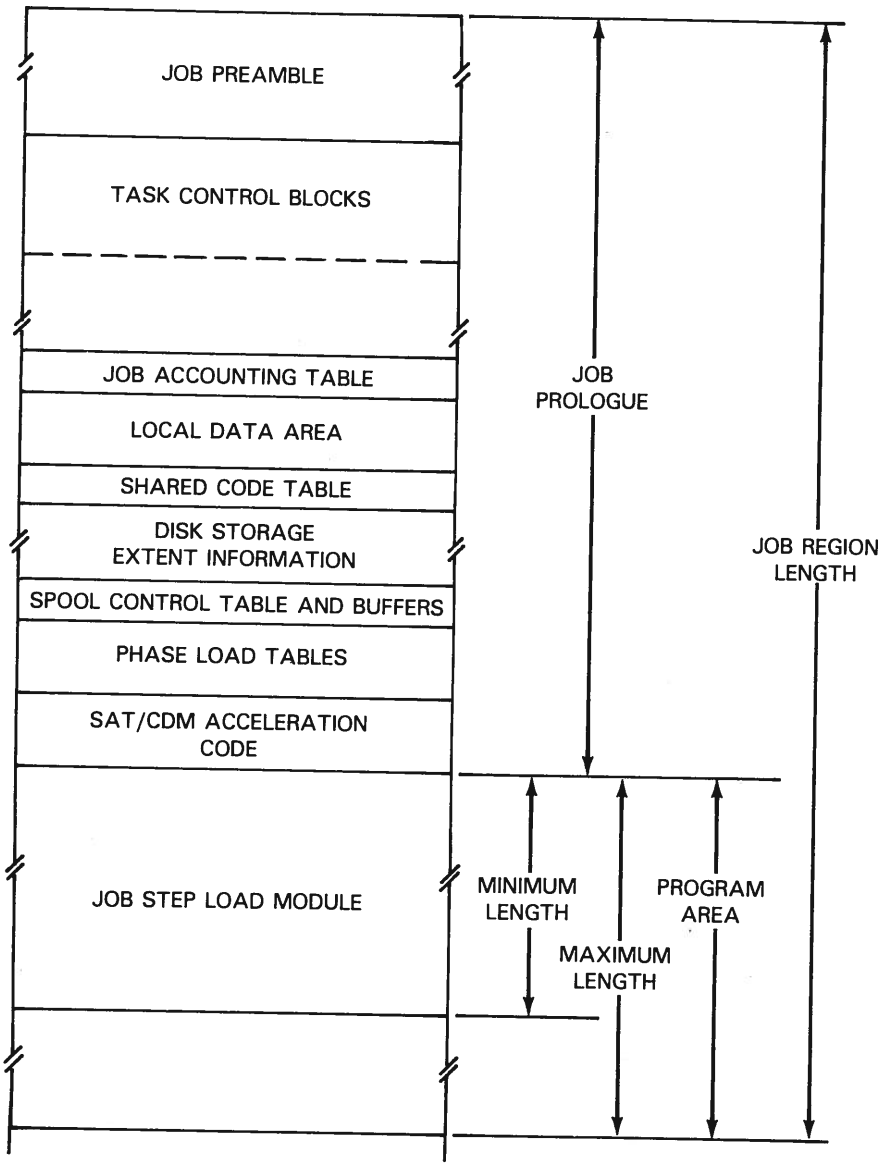


Figure 2-1. Job Region in Main Storage

Dynamic Expansion of Main Storage

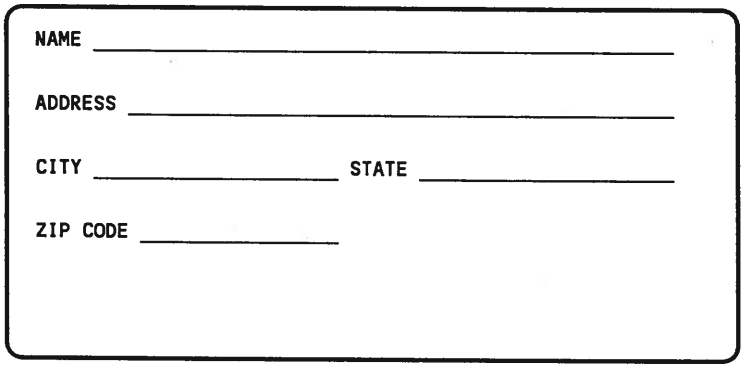
Your job may require dynamic expansion of its initial main storage allocation to load software modules (data management modules, for example), or to accommodate other program modules called by your job. This capacity for dynamic expansion of the job region is called the DLOAD facility. For more information about this facility, see "Defining Software Facilities Needed by Your Job" in Section 6.

Section 3
Minimum Control Stream Requirements

What Is a Minimum Control Stream?

A minimum control stream consists of only those job control statements needed to properly direct the execution of a job.

Let's assume you want to execute a program that has been compiled, link edited, and stored in a library. This particular program does not use any input (cards, tape, disk, etc.) and the only output is directed to the printer. The purpose of the program is to print constants on adhesive-backed mailing labels, like this:



Granted, this isn't a widely used application, but it illustrates a bare minimum control stream.

Constructing the Minimum Control Stream

In order to run this program, we have to construct a control stream to tell the operating system what to do with it. Since the needs of the program are simple, we need very few job control statements.

First, a JOB statement is needed to indicate the beginning of the job to the operating system. Every job entering the system must start with a JOB statement. Each job step does not need a JOB statement, only one for the job as a whole. Next, since there is a print output, a DVC statement is needed to assign a printer to the job. And finally, every peripheral device we use has a file associated with it; every file needs a file name. An LFD statement provides the file name.

The DVC and LFD job control statements make up a basic device assignment set. Since the printer is the only peripheral device used by our program, no other device assignment sets are required.

In fact, there are no other processing options needed for this program. We are now ready to initiate the execution of the job step (our entire job consists of only one job step). We need an EXEC statement for this.

Now our program has all the job control statements that it needs to function. But, when it is finished, we have to tell the system that our control stream is finished. We need a /& job control statement.

Briefly, we have indicated all the job control statements needed for this simple program. They are:

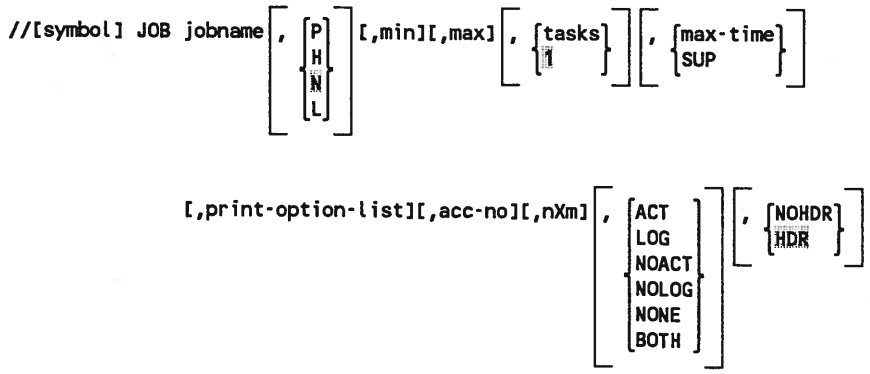
- JOB
- DVC
- LFD
- EXEC
- /&

We will cover each of these job control statements in its proper sequence. We will show all the parameters available for these job control statements, but, at first, only those parameters that are required will be described, along with any parameters that are generated by default. The optional parameters will be introduced into the discussion of job control at the appropriate time.

But, before we start our control stream, you should read the statement conventions in Appendix A. They explain how the job control statements are presented in text (how you can tell which parameters are optional, which are required, how a default option is shown, etc.) and how you code them.

The Beginning of the Job

The JOB job control statement is the first job control statement that you need. Its format is:



As you can see, it has quite a number of parameters. You can specify the name of the job, the priority, how much main storage is needed, the amount of tasks in a job step, how long the job should take, special information for display on the system console, accounting information, spooling buffer size, and log information (where your accounting record is kept).

The only parameter we are interested in right now is the *jobname* parameter, and any default parameters (shown by shading) that are generated.

The *jobname* parameter does just what it implies: it names the job. It consists of one to eight alphanumeric characters. Do not hyphenate the job name if you plan to save the job. The save processor does not allow or recognize hyphens.

For example, we assign the name POCO to the job. It's coded as:

```
// JOB POCO
```

By default, the job has a normal scheduling priority (N) and one task (1).

There is a special feature of the *jobname* parameter that helps you ensure that unique job names are always assigned - you can use trailing ampersands (&) in the job name. You could, for example, code:

```
// JOB POCO&&&&
```

When the stream is processed, the system replaces the ampersands with unique numbers.

When would you use this feature? If you have a job control stream (POCO for example) that is used frequently by different personnel - perhaps even concurrently from workstations - all the users could use POCO&&&& and be assured of having unique job names assigned. It is recommended that if you use this feature, you use at least three trailing ampersands.

You can override the parameters specified on the JOB control statement through selected features of the OPTION job control statement, which is explained in "Selecting Optional Features" in Section 6.

Identifying the Devices

The next entry needed in the control stream is for the printer. The DVC job control statement is used to request the assignment of peripheral devices to a job. Its format is:

```
//[symbol] DVC  [nnn[(n)]]  [ ,[addr      ] ]
              {RES       }  [  [OPT       ] ]
              [RUN       ]  [  [IGNORE    ] ]
                            [  {ALT       } ]
                            [  [I         ] ]
                            [  [O         ] ]
                            [  [REQ[(n)]  ] ]
                            [  [REAL      ] ]
```

The DVC job control statement specifies the logical unit number associated with a peripheral device type. It can also be used to assign alternate devices, or to specify that the job should be executed even if the requested devices are unavailable.

Here, again, we are only interested in the required parameter specifying the logical unit number. There are no default parameters.

The *nnn* is a decimal number indicating the logical unit number of the device. By looking at the following information (taken from Table A-1 of the *Job Control Programming Reference Manual*, UP-9984) we see that the category for printers is in the range of 14-29. If we are willing to use any printer that is available, we use logical unit number 20 or 21. But, it just so happens that there also are a 0776 printer subsystem and a 9246 printer subsystem available in the system. Our program uses a special character that is only present on the 0776 printer, so we will use logical unit number 24.

Logical Unit No.	Device Type Code	Device Type and Features
20, 21	04FF0000	Any printer, no features specified
22, 23	04400000	9246 printer, no features specified
24, 25	04100000	0776 printer, no optional features

You'll notice that there are two other choices for the DVC job control statement: RES and RUN. They will be discussed and used in later examples.

We can now add the DVC job control statement to our control stream as follows:

```
// JOB POCO
// DVC 24
```

Notes:

1. *The* (n) *portion of the* nnn *parameter is used only when the logical unit number indicates a workstation device.*

2. *Logical unit numbers can be changed at system generation (SYSGEN) time to suit the needs of a particular installation. You must be aware of any changes because they could cause device assignment problems within your control stream, especially if you're using JPROCS supplied by Unisys.*

Assigning a Logical File Name to the File

Every device assignment set in the control stream ends with the LFD job control statement. This associates the file defined in the program with the file information in the control stream. Its format is:

```
//[symbol] LFD  {filename }  [ ,[n]  ]  [ ,[EXTEND  ] ]
                {*filename}  [ ,[8]  ]  [ ,{INIT    } ]
                                        [ ,{PREP    } ]
                                        [ ,[ID      ] ]
                                        [ ,[IGNORE  ] ]
```

The LFD job control statement specifies the file name of the file. It's also used to reserve main storage for information about disk file extents, write over the information of the file, and add to the data already in the file.

The *filename* parameter specifies the name of the file you are going to use, and must correspond to the name given to the file in the program. The file name for the LFD job control statement is determined in the following manner:

- The basic assembly language (BAL) programmer uses the name in the label field of the file definition macroinstruction.
- The COBOL programmer uses the external name from the SELECT entry in the environment division. (If the external name is omitted in COBOL 68, use the file name from the SELECT entry.)
- The FORTRAN IV programmer uses the device number from the READ or WRITE statement, prefixed by FORT.
- The FORTRAN 77 programmer uses F0 followed by the unit number unless a specific name was specified in the OPEN statement.
- The RPG II programmer uses the file name from the file description specification.

The *filename* parameter is normally one to eight alphanumeric characters, but if you are using a data management file, it has a maximum of seven characters. This is because data management allows only one to seven characters in the label field of the file definition macroinstruction.

If an asterisk is placed in front of the file name on the LFD job control statement, it means this is an input-only file; you cannot write on it. The operator should be notified of this so he can take appropriate action.

For our control stream example, we'll assume our program is a COBOL program. The file name for the printer in the FD entry is WRITEOUT. We can now add the LFD job control statement to our control stream.

```
// JOB POCO
// DVC 24
// LFD WRITEOUT
```

Executing the Program

We have defined all the requirements of the program to the operating system. Now we have to provide a job control statement to call the sorted program from a library and initiate execution. This is done with an EXEC job control statement. Before the program is actually loaded, any outstanding tape and disk mounting requests are completed.

The format of the EXEC job control statement is:

```
//[symbol] EXEC program-name[, {[library-name]}][,switch-priority][,ABNORM=label]
                              {$Y$RUN       }
                              {$Y$LOD       }
```

The EXEC job control statement identifies the name of the load module. It is also used to specify the library containing the load module, the task switching priority, and any action to be taken if the program causes an abnormal termination.

Once more, we are only interested in the required parameter and any default parameters generated.

The *program-name* parameter identifies the load module to be executed. Every program that is successfully compiled and link edited creates a load module. Every load module that is created and every routine supplied by Unisys must have a name. The LOADM linkage editor control statement assigns a name to a load module; the EXEC job control statement calls the load module by a program name. These names must agree.

For example, you link edit your program with the module name TESTR on the LOADM linkage editor control statement. The linkage editor creates the load module with the name TESTR. When you want to execute this program, your EXEC job control statement uses this same name: TESTR.

If, when you link edit your object module, you do not use a LOADM linkage editor control statement, the load module name, by default, is LNKLOD.

Assume that this program is stored in a library from which it can be retrieved as many times as needed. When the program was link edited, the linkage editor was instructed to place the load module in a specific, permanent library; otherwise, it automatically would have been placed in the job's YRUN file, which is only a temporary file. Assume it is located in the system load library file (YLOD), and the load module name is LABELS. Since YLOD is the default parameter generated for the load library, we only need to specify the program name, which is the same as the load module name: LABELS.

We can now add the EXEC job control statement to our control stream as follows:

```
// JOB POCO
// DVC 24
// LFD WRITEOUT
// EXEC LABELS
```

By default, the lowest available task switching priority established at system generation time is used.

Ending the Basic Control Stream

So far, we have provided all the job control statements needed to construct a basic control stream: JOB, DVC, LFD, and EXEC.

This control stream is all the system needs to execute our simple program. But, after the program executes, the system returns to job control to obtain the next job control statement. Because the job is finished, a /& job control statement is used to signal the end of the job. Its format is:

```
/&
```

This statement has no parameters, but it can have comments. These comments have no effect on the system; they only provide a means of annotation. The comments must be separated from the /& job control statement by at least one blank column.

The statement conventions for coding more than one job control statement on a line (multistatement coding) are presented in Appendix A. The /& job control statement, however, must be the only job control statement on a line.

Adding the /& job control statement, along with some comments, our control stream looks like this:

```
// JOB POCO
// DVC 24
// LFD WRITEOUT
// EXEC LABELS
/&                    END-OF-LABEL-JOB
```

Ending the Card Reader Operation

We have signaled the system we are finished processing. Now, we have to terminate the card reader operation - this informs the system that there are no more cards associated with the job. We do this with a FIN job control statement. Its format is:

```
//[symbol] FIN
```

There are no parameters.

We can now add a FIN job control statement to our control stream, as in the following example:

```
// JOB POCO
// DVC 24
// LFD WRITEOUT
// EXEC LABELS
/&                    END-OF-LABEL-JOB
// FIN
```

The FIN job control statement also signals the end of card input when merging job control statements with stored control streams, submitting data cards as input for a stored control stream, or storing a complete control stream.

Note: *Using the FIN job control statement is unnecessary when input is on data-set-label diskette or in the input spool file.*

The Control Stream So Far - A Review

We have defined everything the system needs to know about the job. It has been given a name, the system was instructed what load module to use, and the job has been assigned the peripheral device it needs. The program is ready for execution.

This control stream represents only a minimum application. We have only scratched the surface of the capabilities of the OS/3 job control. Throughout the rest of this guide, we are going to build on this minimum control stream by adding and modifying job control statements.

Let's assume that the program with a load module name of LABELS was recompiled and link edited after it was modified to accept input from the card reader. This new input contains name and address information that will be printed on the adhesive-backed labels along with the constant information as shown in the following sample.

```
NAME    JOHN A. SMITH
ADDRESS 143 S. 52ND. ST.
CITY    HOMETOWN        STATE   PA.
ZIP CODE 18908
```

Adding Card Input

Since the job will now accept card input, we must provide a device assignment set for the card reader. This means we have to insert a DVC and LFD job control statement for the card reader into the control stream. Once again, their formats are:

```
//[symbol] DVC  [nnn[(n)]]  [, [addr    ]]  [,HOST=host-id]
                {RES     }   [, [OPT     ]]
                [RUN     }   [, [IGNORE  ]]
                             [, {ALT     }]
                             [, {I       }]
                             [, {O       }]
                             [, [REQ[(n)]]]
                             [, [REAL    ]]

//[symbol] LFD  Ffilename G  [, [n]]  [, [EXTEND]]
                G*filenameF  [, [8]]  [, [INIT  ]]
                                      [, {PREP  }]
                                      [, [ID    ]]
                                      [, [IGNORE]]
```

The following section of Table A-1 in the *Job Control Programming Reference*, UP-9984, indicates that the category for card readers is 30-35.

Logical Unit No.	Device Type Code	Device Type and Features
30, 31	08FF0000	Any card reader subsystem, no features specified
32, 33	08200000	0719 card reader, no features specified
34, 35	08800000	0716 card reader, no features specified

For this example, we will assume the system you're using has only one card reader, an 0719 card reader. For a logical unit number, there are four alternatives. We can use 32 or 33, which assigns a 0719 card reader specifically, or, since the 0719 card reader is the only one we have, we can use 30 or 31, which allows us to use any available card reader.

If the system had two card readers, both of a different type, and a particular card reader is needed, you must be more specific in your assignment. If it's immaterial which card reader is used, you could assign the logical unit number for any card reader (30 or 31).

A point to remember about logical unit numbers: if you don't care about the specific device type, use the logical unit number that assigns any device within the category (20 and 21 for printer, 30 and 31 for card readers, etc.). In that way, if there is more than one type of device, you get the first one available. For instance, suppose you selected logical unit number 25 (Unisys 0776 Printer Subsystem) but there is also a 0770 printer connected to the system. The 0776 printer has 40,000 lines waiting to print, while the 0770 printer has a backlog of only 500 lines. By specifying only the 0776 printer, you must wait for the other 40,000 lines to finish printing. By specifying any printer, the output is sent to the first available printer. The logical unit number we are going to use for the card reader is 30.

Note: *When requesting the assignment of more than one device of the same type (two printers, for example), be sure you request the assignment of any specific devices you need before you request the assignment of general ones. This ensures that a specific device you may need (the 0770 printer, for example) will not be allocated for use as a general printer when it's needed as a specific device.*

Now that we have a DVC job control statement for the card reader, we need a corresponding LFD job control statement. Since this program is written in COBOL, we check the SELECT entry in the COBOL program and find that the file name is CARDIN. This file name is coded in the LFD job control statement.

We can now add the device assignment set for the card reader to the control stream. It can be placed anywhere in the control stream, with the following restrictions:

- It must be before the EXEC job control statement.
- It cannot be within embedded data.
- It cannot be within the device assignment set (DVC through LFD sequence) for another device.

Card Input and Embedded Data

To accept data input from a card reader, we must inform the card reader in some way that it is data to be read. In many cases, this data is caused to be read at execution time by data management. In this kind of application, the data cards follow the // FIN card that caused the card reader to be turned off previously. All that is additionally needed is a /* card after the data signifying end of data. There are no other parameters required, and no comments are permitted in the comment area of the card. This /* statement is always required for any type data. Thus, to our control stream we can now add the data, followed by the /* end-of-data statement, and run our job, which consists of the LABELS program. Basically, we are saying to the processor, run my job POCO which executes the program called LABELS - my data is a card file after the FIN statement when you are ready to execute. This will print the name and address information, plus constants, as shown, on adhesive-backed labels that the operator has previously placed in his printer. The following example illustrates this control stream:

```
// JOB POCO
// DVC 24
// LFD WRITEOUT
// DVC 30
// LFD CARDIN
// EXEC LABELS
/&                    END-OF-LABEL-JOB
// FIN
   data-cards
/*
```

Note: *You should be aware, however, that in the case of multiple files, if the first program in the series does not read all of its data cards (along with the /* that signals end of data), the next program step will pick up where the previous one left off. Additionally, if you are programming in higher level languages, such as RPG, COBOL, or FORTRAN, you cannot read multiple card files in a single program without closing and reopening the files.*

Another way in which data cards may be accepted, and which informs the card reader that data is being input, is the embedded data method. This means that the data is embedded within the control stream itself. All it requires is a start-of-data (/$) job control statement immediately after the EXEC statement, followed by the data and the /* end-of-data. /$ has no parameters, and may appear as the last job control statement on a multistatement line.

The advantage of this method is that the device assignment set is no longer required for the reader, since the control stream is already being read. Additionally, the data being read is instantly accessible, which is discussed later in Section 8. A disadvantage is that embedded data in a prefiled job control stream is harder to change than the data in a card file (which follows the // FIN job control statement). This is because the embedded data is actually a part of your control stream rather than a separate card file. Changing embedded data is discussed in "Substituting Embedded Data" and "Replacing Embedded Data Sets in Expanded Control Streams" in Section 6. An example of an embedded data control stream is:

```
// JOB POCO
// DVC 24
// LFD WRITEOUT
// EXEC LABELS
/$
   data-cards
/*
/&                  END-OF-LABEL-JOB
// FIN
```

You can use this method when you become familiar with the programming techniques needed by the language you're using - for example, a COBOL ACCEPT or FORTRAN READ instruction. In fact, programs supplied by Unisys (such as the COBOL compiler and the data utility routines) use this method. It entails the use of a supervisor macroinstruction in the program (if it's assembler language; if it's one of the other languages, there are similar instructions that are used). Again, if you decide to use the embedded data method, the changes to your job control stream are:

1. Remove the device assignment set for the card reader; it's not needed.

2. Place the data (/$, data cards, /*) after the EXEC job control statement. This is what's known as embedded data.

When you use the embedded data method, and you have an 0716 card reader supporting the 96-column card feature, your data file can use the full 96 characters. With data-set-label diskette, you can use up to 128 characters. But, even though your control statements also can be on 96-column cards and data-set-label diskette, only the first 72 columns (characters) can be used for job control statements.

In addition to embedded data, there is a dummy data set. A dummy data set consists of only a /$ and a /*. This is used with some language JPROCS. More information about dummy data sets can be found in the language manuals (COBOL, FORTRAN, etc.).

You can replace embedded data sets in translated, saved job control streams by using the DATA STEP job control statement. Refer to "Dynamic Skip Function from a Workstation" in Section 6 for more information.

The Program Is Changed - Another Device

So far, the program has been written to read name and address cards and print the information, plus constants, on adhesive-backed labels. The program has been refined once more. It is still going to print constants. However, the name and address file is now on magnetic tape, in ZIP Code® sequence. This tape was created by someone else's job. We want to list only the name and addresses of certain ZIP Codes; therefore, we modify the program to accept a table from the card reader. This table contains only the ZIP Codes we want to print. The program instructs the system to compare the ZIP Codes from the table with the file on the magnetic tape and print the names and addresses that match the ZIP Code table.

We have already provided the device assignment sets for the printer and the card reader. Even though the format of the card reader input is different (previously it was the name and address file, now it is the ZIP Code table), no changes are needed to the card reader device assignment set. It was a program change and does not affect the job control stream. The logical unit number is still 30 (DVC job control statement), and the file name in the program is still CARDIN (LFD job control statement). The only new item we have to provide in the control stream is a device assignment set for tape.

What Is Needed to Use a Tape?

We have already said that every peripheral device used needs the DVC and LFD job control statements. For readers, printers, and punches, this is all that is needed to complete the device assignment set. However, magnetic tapes have volume serial numbers, and, optionally, file identifiers. So, the device assignment set for a tape file could be either

```
// DVC ...
// VOL ...
// LFD ...
```

or

```
// DVC ...
// VOL ...
// LBL ...
// LFD ...
```

The first step is to provide a logical unit number and file name.

ZIP Code is a registered trademark of the U. S. Postal Service.

The Logical Unit Number and File Name for the Tape

The range of logical unit number for magnetic tapes is 90-127. The name and address tape is a 9-track, phase-encoded tape. We must be specific. The logical unit number selected for the DVC job control statement is 100. This gives us any tape drive that can read a 9-track, phase-encoded tape; the tape unit transfer rate is immaterial.

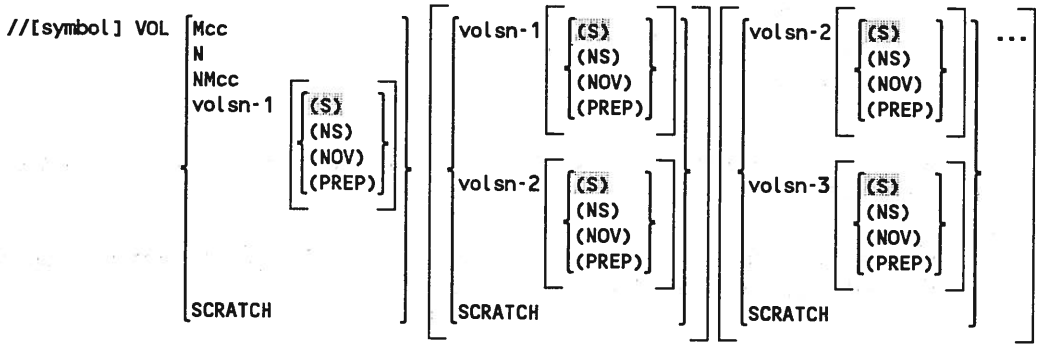
We can now add this partial device assignment set for tape to our control stream.

```
// JOB POCO
// DVC 24
// LFD WRITEOUT
// DVC 30
// LFD CARDIN
// DVC 100
// LFD NAMADD
// EXEC LABELS
/&                    END-OF-LABEL-JOB
// FIN
/*
```

These new DVC and LFD job control statements do not represent the entire device assignment set needed for tape. If we tried to run the job now, it would abort.

Supplying a Volume Serial Number for the Tape

Every tape file used in a job must have a VOL job control statement in the device assignment set. This identifies the volume to be used. Its format is:



The VOL job control statement supplies the volume serial number of the volume to be accessed by the job. However, a tape volume does not necessarily need a volume serial number, but it still must have a VOL job control statement.

You can also use the VOL job control statement to: count the number of blocks in the file; specify the mode characteristics of the tape; request data management to write a volume serial number; inhibit the checking of volume serial numbers if they are not known; or, to indicate that the volume may also be used by someone else at the same time that you are using it (this only applies to disk).

Again we are only interested in the required parameter. This parameter has several different options, but for this job, only the volume serial number is needed.

The *volsn-1* parameter is the 1- to 6-alphanumeric-character volume serial number of the first volume of the file. A file may span more than one volume. Perhaps the length of the file made it necessary to use three tapes (volumes) to hold the entire file. Since this file is on only one volume, only one volume serial number is needed. Assume it to be TAP111.

We can now add the VOL job control statement to our control stream as follows:

```
// JOB POCO
// DVC 24
// LFD WRITEOUT
// DVC 30
// LFD CARDIN
// DVC 100
// VOL TAP111
// LFD NAMADD
// EXEC LABELS
// /&              END-OF-LABEL-JOB
// FIN
   data cards
/*
```

This control stream could now be run, provided that the tape is unlabeled (no file identifier).

OS/3 data management supports a maximum of 151 explicit volume names per file for disk, diskette, and tape files.

Labeled Tapes for File Identification

Just as there can be one or more volumes in a file, there can also be one or more files in a volume. Suppose the tape volume contained five files. It would be necessary to have file identifiers on each particular file to access the proper file. Single-file tape volumes also can have file identifiers. This is done to ensure that the correct file is used. Even though the volume serial number is checked to see if the proper tape is mounted, it is possible that this tape does not have the proper file needed for the job. For example, someone could have inadvertently written on the tape because it did not have a file identifier to indicate that this tape already contains information to be saved. By using a file identifier, you indicate this is a *saved tape.* Had there been a file identifier on the tape, anyone trying to write on this tape would have been notified that this is a saved tape.

The LBL job control statement is used to either check or create a file identifier. Its format is:

```
//[symbol] LBL  {file-identifier ,}  [, {file-serial-number}]  [,expiration-date]
                {'file-identifier  }  [, {VCHECK          }]

              [,creation-date] [, {file-sequence-number}]] [, {generation-number}]
                               [, {1                   }]  [, {1                 }]

              [, {version-number}]
                 {1             }
```

The LBL job control statement identifies the file. It also can be used to: ensure that the correct members of a multivolume file are used; indicate the date the file can be deleted (by a SCR job control statement); indicate the date the file was created; indicate the position of the file in respect to the other files in a multifile tape volume; and, specify the generation and version number of a tape file, thus ensuring the most current edition of the tape file is used.

We only want to ensure that the proper file is on the tape volume, so we need only the required parameter.

The *file-identifier* parameter is 1 to 17 alphanumeric characters for tape, card, and diskette files. It is 1 to 44 alphanumeric characters for a disk file unless that file is a scratch (temporary) file; then the file-identifier is 1 to 39 alphanumeric characters. If the file-identifier contains embedded blanks, it must be enclosed by single quotation marks.

Assume that MASTERFILE is the file identifier assigned to this tape file when it was created. We can now add the LBL job control statement to the control stream as shown in the example.

```
// JOB POCO
// DVC 24
// LFD WRITEOUT
// DVC 30
// LFD CARDIN
// DVC 100
// VOL TAP111
// LBL MASTERFILE
// LFD NAMADD
// EXEC LABELS
/&                    END-OF-LABEL-JOB
/&
// FIN
   data cards
/*
```

The default parameters generated indicate this is the only file on the volume (1), and it is the only edition of the file (1).

Note: *File identifiers prefixed by $SCR refer to job step temporary files; those prefixed by $JOB refer to job temporary files.*

Another Programming Change - Another Device Assignment

The site manager has determined the label program doesn't fulfill all the requirements for which it was intended. Once more, it must be changed.

The name and address file was copied from the tape volume to a disk volume by using a Unisys data utility routine. Now, the input name and address file is on disk, the ZIP Code table is still input from the card reader, and the selected names and addresses, plus constants, are still printed on adhesive-backed labels. These selected names and addresses are now going to be saved and output to a file on a tape volume for a later processing application.

Although there may be many programming changes involved, the control stream changes are minimal.

The device assignment set for the card reader, the printer, or the tape doesn't need changing. Even though the tape was used previously as an input file, converting it to an output file is only going to involve changes in the program; it is not reflected in the control stream. After the tape was copied to disk, the information it contained was deleted in another procedure. We can use this tape with a volume serial number of TAP111 as the output tape. We can also use the same logical unit number in the DVC job control statement. NAMADD is used as the file name for the output tape file in the program. This allows us to continue using NAMADD as the file name in the LFD job control statement. However, we are going to give this tape file a different file identifier. In the previous device assignment set for the tape it was MASTERFILE. We want to change it to reflect its purpose.

It is no longer a master file for input; it is an output tape - let's call it OUTPUTTAPE. This requires a change to the *file-identifier* parameter of the LBL job control statement for the tape device assignment set. We do not need to change it, but to make the purpose and the name agree, we will. Changing the LBL job control statement makes our control stream look like this:

```
// JOB POCO
// DVC 24
// LFD WRITEOUT
// DVC 30
// LFD CARDIN
// DVC 100
// VOL TAP111
// LBL OUTPUTTAPE
// LFD NAMADD
// EXEC LABELS
/&                    END-OF-LABEL-JOB
// FIN
   data-cards
/*
```

We still must provide a device assignment set for the name and address file input from disk.

The Device Assignment Set for a Disk or Format-Label Diskette

The following chart lists the necessary job control statements for the basic disk and format-label diskette device assignment set.

Allocation	Your Disk or Format-Label Diskette	SYSRES or YRUN File (Disk only*)
Previously Allocated	DVC VOL LBL LFD	DVC LBL LFD
Not Allocated	DVC VOL LBL EXT LFD	DVC LBL EXT LFD

*A format-label diskette volume cannot be used as your SYSRES volume or the volume containing the YRUN file.

In our case we have a disk file, the extent was allocated, and the file is not SYSRES or the job's YRUN file. So the following job control statements are needed: DVC, VOL, LBL, and LFD.

The disk pack used for the name and address file fits on an 8494 Disk Subsystem. The logical unit number we are going to use for the DVC job control statement is 80.

Within the program, the file name from the FD entry is DKNAME. This is the file name for our LFD job control statement.

We need a VOL job control statement to indicate the volume serial number of the disk we are going to use. We need only the required parameter for the volume serial number. Assume the site manager had the name and address file copied to the disk with a volume serial number of DSK001.

Since most disk volumes contain many files, each file needs a file identifier. When the site manager copied this file, he allocated it with a file identifier of DSKMASTFIL. We must specify this in an LBL job control statement.

We now have all the information needed for the disk file. We can add the device assignment set for the disk input file to our control stream and run the job.

```
// JOB POCO
// DVC 24
// LFD WRITEOUT
// DVC 30
// LFD CARDIN
// DVC 100
// VOL TAP111
// LBL OUTPUTTAPE
// LFD NAMADD
// DVC 80
// VOL DSK001
// LBL DSKMASTFIL
// LFD DKNAME
// EXEC LABELS
/&                    END-OF-LABEL-JOB
// FIN
   data-cards
/*
```

The Device Assignment Set for Data-Set-Label Diskette

The prep routine for data-set-label diskette automatically allocates the entire diskette for one file and assigns a file identifier of DATA (unless you specify otherwise). When this file is used, you must include a device assignment set in your job control stream that consists of the DVC, VOL, LBL, and LFD job control statements. For example:

```
// DVC 130
// VOL DSL01
// LBL DATA
// LFD FILE01
```

You only include an EXT statement in the device assignment set (and specify your own identifier on the LBL statement) if the space wasn't already allocated during the diskette prep routine. See "Information about Data-Set-Label Diskette File Allocation" in Section 4 for information about the EXT statement.

The Device Assignment Set for a Workstation

The DVC and LFD job control statements are required for a basic workstation device assignment set. The UID, USE SFS, USE DP, and USE MENU statements are included under certain circumstances.

The UID Job Control Statement

The UID job control statement may be used as part of the device assignment set for a workstation when you want to ensure that specific workstations, identified by user-id or device address, are automatically connected to a job. This is done before a job's execution begins (if the workstation has not already been connected via the CONNECT command). Its format is:

```
//[symbol] UID  {user-id-1          } [,..., {user-id-255          }]
                {(addr-1)           }       {(addr-255)          }
                {user-id-1(addr-1)  }       {user-id-255(addr-255)}
```

A maximum of 255 workstations may be specified. You can specify YMAS as a user-id to assign the job's master workstation to a job. The *user-id* parameter is one to six alphanumeric characters in length. A device assignment set that assigns the workstation being used by user-id (JONES1) could look like this:

```
      .
      .
      .
// DVC 200
// UID JONES1
// LFD WKSTN
      .
      .
      .
```

Assigning workstations is discussed in more detail in "Specifying Multiple Workstations" in Section 4.

The USE Job Control Statement

If you are preparing a control stream for a program that uses screen format services, menu services, or the dialog processor, you must include a USE job control statement as part of your workstation device assignment set. Three different forms of the USE statement make it possible for you to specify which workstation service you want. These are as follows:

```
// USE SFS...     (for screen format services)
// USE MENU...    (for menu services)
// USE DP...      (for dialog processing)
```

Each statement and its accompanying parameters is discussed further in Section 6 in "The USE Statement for Screen Format Services", "The USE Statement for Menu Services", and "The USE Statement for Dialog Processing", respectively.

Job Step Temporary and Job Temporary Files

To satisfy the needs of the software components for disk work areas, files lasting for a job step and for the length of the job are provided. These files are deleted at the end of the job step or the end of the job. While these files are primarily used by the software components, the ability to allocate and use temporary files is also available to you.

Basically, you allocate job step temporary and job temporary files the same way you'd allocate any disk file. The only difference is you must prefix your file identifier with $SCR for a job step temporary file and $JOB for a job temporary file. For example, to allocate a job step temporary file, you could include the following device assignment set in your job control stream:

```
// DVC 50
// VOL D12345
// LBL $SCRWORK1
// EXT MI,,,CYL,2
// LFD WORKFIL
```

When a temporary work file ($SCR, $JOB) is allocated, the file label is modified by job control to allow concurrent jobs using the same file identifiers to access the proper work file. Every job in the system is assigned a unique job number. The label $SCR1 in JCL is allocated as $SCRnnnnn1, where nnnnn is the job number.

Job step temporary files are automatically deleted at the end of the job step, while job temporary files are automatically deleted at the end of the job. If the system is reinitialized in the middle of your job, job control automatically scratches job temporary files and job step temporary files when it reallocates them.

See "Setting Up Temporary Work Files" in Section 5 for information about using JPROCS to allocate job step temporary and job temporary files.

Basic Job Control Statements

This section has covered the job control statements needed to run most jobs. In the following section, we are going to use the basic job control statements and add the optional parameters, explaining how each parameter affects the performance of the job.

Section 4
Getting the Most Out of the Basic Job Control Statements

Optional Parameters Can Improve Job Performance

So far, in our discussions of basic job control statements, we've concentrated on the required parameters. A great deal of work can be accomplished using just these parameters. Sometimes, however, required parameters won't provide enough information. In other instances, the ability to provide more information to the system will speed up job execution. Additional information about a job and its peripheral devices is supplied via the optional parameters that are part of the basic job control statements. This section describes these parameters and shows how they are used.

Improving Your Control of the Job

The JOB control statement was used to give a name to the job. It is used also to specify the following: a selection priority; the main storage size for the job; how many tasks are in any one job step; how long the job should take; a list of the control streams on the operator's system console for debugging purposes; and spooling buffer sizes. Once again, its format is:

```
//[symbol] JOB jobname[, {P}][,min][,max][, {tasks}][, {max-time}]
                       {H}                 {1    }  {SUP     }
                       {N}
                       {L}

[,print-option-list][,acc-no][,nXm][, {ACT   }][, {NOHDR}]
                                     {LOG   }      {HDR  }
                                     {NOACT }
                                     {NOLOG }
                                     {NONE  }
                                     {BOTH  }
```

As you can see, some optional parameters generate default values when they are omitted. In the previous discussion of the JOB control statement, only the required parameter - *jobname* - was coded. By so doing, we indicated that, by default, the job is to have normal priority (N) and there is only one task (1). This points up the fact that when only the required parameters are specified, you are, in many cases, providing more information about the job than is contained in the required parameters. The default values were selected because they conform to the most frequently used programming practices. This allows you to code as short a control statement as possible. The less there is to code, the less chance there is of making a coding error.

Note: *The OPTION job control statement can be used to override individual parameters of the JOB control statement. Refer to "Selecting Optional Features" in Section 6 for more information.*

A Selection Priority for the Job

Jobs are selected for execution on a priority basis. The second parameter on the JOB control statement specifies the priority. There are four priorities: low (L), normal (N), high (H), and preemptive (P). Remember our discussion on the use of priorities in Section 1, where we outlined how the priority is used by the system for selecting jobs and what each priority means?

Most jobs are normal priority, which is by default, the parameter generated. If you need another priority, you have to specify it.

It so happens that the label job named POCO is needed in a hurry, so the system administrator allowed you to assign high priority. Added to the existing JOB control statement, it would be coded as:

```
// JOB POCO,H
```

Main Storage Needs

When the load module named on the //EXECUTE statement is in a load library on a mounted disk volume, you don't have to indicate the minimum amount of main storage to execute the load module. If the disk volume containing the load module is not already mounted, you must indicate the minimum amount of main storage needed to execute the module.

The *min* parameter does this. The minimum main storage size is specified in decimal or hexadecimal. The smallest amount that can be specified is 8K decimal bytes (2000 in hexadecimal). The area used by the job prologue is not included in this amount.

Assume the label program needs approximately 12K (12,288) decimal bytes (3000 in hexadecimal) and that it's in a load library on your own volume. The JOB control statement would now be:

```
// JOB POCO,H,3000
```

or

```
// JOB POCO,H,X'3000'
```

You can also specify the minimum main storage size in decimal. This is done by coding *D'number'* for the *min* parameter, as illustrated in the following JOB control statement:

```
// JOB POCO,H,D'12288'
```

For the sake of illustrating the omission of positional parameters, this JOB control statement is coded as follows when the priority is omitted (it would be assigned the normal priority, by default, by the system):

```
// JOB POCO,,3000
```

See "Coding Conventions" in Appendix A for information about coding numbers in job control statements.

Note: *If a job consists of multiple job steps, specify only the minimum main storage size needed by the largest load module.*

Consider the possibility that you may be running a 3-job-step job, consisting of perhaps a COBOL compile, followed by a link edit, and then the execution of the generated load module. OS/3 knows how much main storage to allocate for both the COBOL compiler and the linkage editor, but there is no way OS/3 can know how much is required for the execution of your program, since it is not generated until after all the job control has been interpreted. If your generated load module does not use more main storage than the COBOL compiler (which is larger than the linkage editor, thus the largest known job step), then your load module will have sufficient main storage allocated. On the other hand, if your load module is larger than the COBOL compiler, not enough main storage will be reserved.

More Main Storage to Speed Up the Job

In addition to specifying the minimum main storage, you can also request additional main storage. This is an amount that can be used, but is not required, to speed up job execution. However, the program must be structured to take advantage of the additional main storage; for example, a segmented COBOL program. Some of the routines supplied by Unisys that use extra main storage in this manner are sort/merge, linkage editor, and the language translators. Additional memory may also be advisable when running large assembly programs using many tags. As the minimum main storage size is specified in decimal or hexadecimal, so is the maximum; it is the fourth parameter (*max*) shown in the format.

We'll assume that the label program was structured to use 41K decimal bytes (A028 hexadecimal) of main storage, if it is available; also, that it uses the high scheduling priority and needs at least 12K decimal bytes (3000 hexadecimal). Added to our JOB control statement, it would be coded as follows:

```
// JOB POCO,H,3000,A028
```

You can also code X'A028' to represent the maximum main storage size in hexadecimal.

You can specify the maximum main storage size in decimal by coding *D'number'* for the *max* parameter (e.g., D'41000' instead of A028 or X'A028').

If we omitted the scheduling priority (it would default to normal) and the minimum main storage size, it would be coded as follows:

```
// JOB POCO,,,A028
```

Note: *If either the* min *or the* max *parameter is omitted, the value specified for one is used for the other. If both are omitted, and the load module is not located in YLOD (on SYSRES) or in an alternate load library on either SYSRES or the volume containing the job's YRUN file, job control automatically allocates 8K decimal bytes of main storage (2000 in hexadecimal). If you have requested a job dump through the OPTION statement (JOBDUMP), and you have not specified* min *or* max *on the JOB statement, job control nearly doubles the amount of main storage that is automatically allocated. If you specify* min *or* max *and intend to request a job dump, specify at least 14K decimal bytes (3500 in hexadecimal).*

Multitasking Specification

If a program is written in BAL, you can create multiple tasks within it by using the task parameter. This is called multitasking.

So far, we have been saying that job POCO is written in COBOL. For this example, assume that it is written in BAL, and that we are going to allow for 18 tasks to be active. The job still needs 12K decimal bytes to execute, but it can use 41K decimal bytes, and has a high scheduling priority. Adding the multitasking specification would make our JOB control statement look like this:

```
// JOB POCO,H,3000,A028,18
```

Each task specified requires 256 bytes in the job prologue. The maximum number of tasks you can have within a job is limited by the maximum size of the prologue (65535 bytes). If we omit the *task* parameter, job control assumes 1 by default.

Note: *There are other tables which require prologue space and their size varies depending, for example, on the number of files and spool buffers declared through job control. If you exceed the prologue size (you receive an R289 message and the job is not scheduled), you can reduce the number of tasks, files, or spool buffers specified.*

The Processing Time for the Job

After the same job has run several times, you probably know how long it takes to execute. Should it run longer, it may mean something is wrong - perhaps there is a "bug" that has never been encountered before. Rather than waste processing time, you can set a processing time limit using the *max-time* parameter. If the job executes beyond this time limit, a message is sent to the operator, who can either cancel the job or extend the time limit by any increment. If you specify *max-time*, you should tell the operator what action to take if the specified processing time is exceeded.

The *max-time* limit is specified in minutes. It refers to elapsed wall-clock time or to elapsed CPU time, depending upon how your supervisor is configured. If you want to suppress the *max-time* function completely for a particular job, you can specify SUP in the *max-time* parameter, rather than a number.

The system will adjust the *max-time* value to allow for the following conditions:

- Checkpoint/restart
- PAUSE job control statements
- SET CLOCK commands
- Roll-in/roll-out

If you omit *max-time*, the time limit set at system generation is used as the default value. The *max-time* parameter is supported only on supervisors configured with NORMAL or MAX timer services. If a timer service is not specified at system generation, *max-time* specifications are ignored.

Suppose you know that the job POCO should take no more than 15 minutes to run. Added to the other parameters of the JOB statement, the max-time parameter is coded as follows:

```
// JOB POCO,H,3000,A028,18,15
```

Debugging the Control Stream

With the *print-option-list* parameter, you can control the printing of job control statements and JPROC listing by specifying one or more available options. In a spooling system, statements are printed (without passwords) in the job log; otherwise, they are displayed on the system console. This gives a graphic display or printout of the control stream for debugging purposes. For example, if a particular control stream is run for the very first time and there are syntax errors in the coding, the system will generate an error message telling you so. If you have used one of the debugging list options, you receive a listing of your control stream.

The options for this parameter are:

B Lists job control statements with symbol substitution. This is the default in a spooling system.

D Lists job control statements (as they're read in by the run processor) without any symbol substitution

P Lists completed job control statements, which are generated by a procedure call statement in the control stream, showing the values assigned in the procedure definition statements

E Lists any data contained in the control stream

S Lists all the job control statements skipped as a result of an IF or GO job control statement

A Combines all the options

W Suppresses the display of job control warning errors on the console or workstation but not on the job log

N None of the options are in effect (the default in a nonspooling system).

You may specify more than one option on a JOB control statement. However, if more than one option is specified, the parameter group must be enclosed in parentheses. Each option must be separated by a comma and can be specified in any order. For example, *(S,P,E)* or *(P,E,S)*; when only one option is specified, no parentheses are needed.

When the D, P, E, or S options are chosen (separately or in combination) you get a listing of your basic job control statements with symbol substitution even if B is not specified.

Let's assume this is the first time we are running job POCO, and we want to list the basic job control statements with symbol substitution, the job control statements generated by a procedure call, and the data. These are options *B*, *P*, and *E*, but since the option B is in effect when either P or E is chosen, you don't have to specify it. Added to the other parameters of our JOB control statement, it would be coded as either:

```
// JOB POCO,H,3000,A028,18,15,(P,E)
```

or

```
// JOB POCO,H,3000,A028,18,15,(E,P)
```

Job Accounting and Spool Buffers

Use the *acc-no* parameter to provide the account number that has been assigned to you at your installation. This 1- to 4-alphanumeric-character parameter creates an entry in the job preamble for this account number, containing the total elapsed wall clock time. Wall clock time can be defined as the point in time when a job is initiated for execution, up to the time when the job terminates. Therefore, any time used by spool input and spool output is not included.

This parameter may or may not be required, depending on the accounting procedures used at your installation.

Suppose the account number assigned to you is A001. Adding this information would make the existing JOB control statement appear as:

```
// JOB POCO,H,3000,A028,18,15,(E,B),A001
```

The *nXm* parameter sets up buffers for the file. This buffer holds data from the time it first becomes available until the time it's needed for processing. Thus, the central processor does not have to wait as long for data. The job log and any spooled files that don't have their own buffers can share these buffers.

When coded, the *n* is the number of buffers, *X* is a constant, and the *m* is the number of (256-byte) blocks. Whenever *nXm* is omitted, a single 256-byte buffer (1X1) is reserved if only the job log is sharing the buffer with your spool files. If other spool files are also sharing the buffer, two buffers of 512 bytes each (2X2) are allocated for a total of 1024 bytes.

For example, if you wanted to allocate two buffers of 2048 bytes total, you would code:

```
// JOB POCO,H,3000,A028,18,15,(E,B),A001,2X4
```

The only values accepted for *m* are 1, 2, 4, 8, 16, and 32. Numbers larger than 32 default to 32. Numbers not in the acceptable range are changed to the lower acceptable constant (e.g., 6 is changed to 4).

Printing the Job Log File and Page Headers

The job log file contains the job accounting records, dumps created as a result of an OPTION job control statement with the *DUMP* parameter, and a log, or list, of messages and job control statements that were displayed on the system console. You can selectively print this job log file with your job, by using one of the following parameter choices of the JOB control statement:

```
ACT
LOG
NOACT
NOLOG
NONE
BOTH
```

The *ACT* parameter forces the printing of accounting records, regardless of the system options in effect. *LOG* forces the printing of job log records, regardless of the system options in effect. The *NOACT* parameter, when used, suppresses the printing of accounting records. The *NOLOG* parameter means do not print the log (which also contains dumps generated by an OPTION DUMP job control statement). If you code the *NONE* parameter, both the log and accounting records aren't printed. The *BOTH* parameter allows both the log and accounting records to print. If you don't specify one of these parameters, the system options in effect are used.

For example, if you want only the accounting information to print (no log records - *NOLOG*), you would code:

```
// JOB POCO,H,3000,A028,18,15,(E,B),A001,2X4,NOLOG
```

Cancel and snapshot dumps are never suppressed. If you're running in a nonspooling environment, this parameter is ignored.

At the beginning of the job log and accounting record printout, a page header, which consists of several lines of asterisks, is printed. This can be suppressed by coding the *NOHDR* parameter on the job control statement; by default, *HDR* is generated. Coded, it would be:

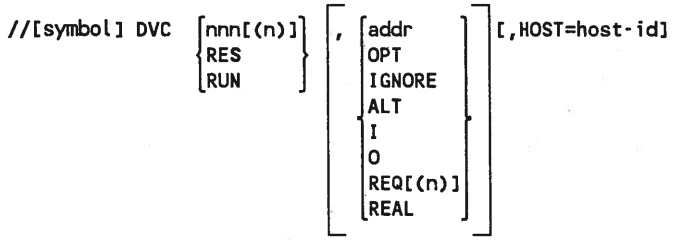
```
// JOB POCO,H,3000,A028,18,15,(E,B),A001,2X4,NOLOG,NOHDR
```

This parameter is ignored if you're not spooling.

A job log report program is also available that will provide you with a job accounting report based on the contents of the log file. For more information about the job log report program, refer to the *System Service Programs (SSP) Operating Guide*, UP-8841.

Identifying the Peripheral Devices a Little Further

The DVC job control statement associates a physical device type, specified by a logical unit number, with your job. It can also be used to: assign multiple devices, in a serial manner, during a job step; provide the physical address of the unit for using a specific device; or (in a DDP environment), indicate that a disk file is remotely located. Here, again, is its format:



Refer to this format when each new parameter is introduced.

Note: *A particular job cannot mix RBP destinations with auxiliary printers or DDP destinations.*

Using Multiple Devices, SYSRES, or the Job's YRUN File

The first parameter has three choices: *nnn*, *RES*, or *RUN*. (Remember, the *(n)* portion of *nnn* is only used when assigning workstations.)

We have already explained how to use *nnn* to specify a logical unit number (see "Identifying the Devices" in Section 3). However, if you want to use more than one print, punch, or card file in a job, you should assign a different logical unit number to each file because the run processor flags multiple occurrences of the same logical unit number in the same job step. If your system contains only 0776 printers, for example, you can use the logical unit numbers 20, 21, 24, and 25. Sometimes, in a spooling environment, you may want to assign more than four virtual printers or punches. To do this, you must use the EQU statement (see "Equating Logical Unit Numbers to Device Type Codes" in Section 6) to equate additional logical unit numbers to your devices. You can use any logical unit number that is not already in your system. The EQU statement is placed just before the device assignment set. To get an 0776 printer when you have already used the logical unit numbers 20, 21, 24, and 25, you might use the logical unit number 10, as follows:

```
// EQU 10,0410
// DVC 10
```

The number used for the type parameter of the EQU statement, 0410, is listed in Table A-1 of the *Job Control Programming Reference Manual*, UP-9984, as the device type code for the 0776 printer.

Note: *The maximum number of unique devices allowed in a job is 255. The maximum number of unit record devices (e.g., card readers, data-set-label diskettes, printers) allowed in one job is 42.*

You don't have to supply a logical unit number for files in SYSRES or the volume containing the job's YRUN file. Use RES to indicate that the file is on the SYSRES volume, or RUN to indicate that the file is on the volume containing the job's YRUN file. Whenever RES or RUN is used, you can omit the VOL job control statement in the device assignment set. The system differentiates between which volume is the SYSRES volume and which volume contains the job's YRUN file. *RES* or *RUN* can only be used for disk files.

In our control stream, we used this device assignment set for the name and address disk input file as follows:

```
// DVC 60
// VOL DSK001
// LBL DSKMASTFIL
// LFD DKNAME
```

If, instead of using the disk with a volume serial number of DSK001, the site manager puts the name and address file on the SYSRES volume, still using the file identifier of DSKMASTFIL, and assuming the file name in the program is still DKNAME, then the device assignment set is:

```
// DVC RES
// LBL DSKMASTFIL
// LFD DKNAME
```

The VOL job control statement is omitted because the file is on SYSRES.

Specifying Multiple Workstations

Suppose you want to access a workstation file from more than one workstation. The *(n)* portion of the DVC statement's *nnn* parameter allows you to associate up to 255 workstations of the type and characteristics specified by *(nnn)* with one file. Consider the following example:

```
// DVC 200(4)
      .
      .
      .
// LFD WKSTFILE
```

When the DVC statement is specified like this, up to four workstations can be logged on and then optionally connected (using the workstation CONNECT command) to the same job. These workstations access WKSTFILE.

If all four workstations must be connected for the job to begin execution, use the REQ parameter of // DVC, like this:

```
// DVC 200(4),REQ
```

The UID statement is used when you want specific required workstations automatically connected to the job.

The *REQ* parameter and the UID job control statement are discussed further in "Ensuring that Workstations are Connected to a Job" later in the section.

More Control over Peripheral Devices

The format shows there are eight possible choices for the second parameter of the DVC job control statement: *addr*, *OPT*, *IGNORE*, *ALT*, *I*, *O*, *REQ*, and *REAL*. They are explained in the following paragraphs, except for *I* and *O*, which are explained when we discuss spooling diskette files. Refer to "Spooling Input Card Data" in Section 6 for more information.

Assigning Devices by Physical Address and Assigning Real Devices

Every device has a physical address associated with it. This is a hexadecimal number representing the channel number, control unit address, and device number. It is assigned by a Unisys customer service engineer. You can specify it by using the *addr* parameter of the DVC job control statement.

It is unlikely you will need to use the *addr* parameter because the system can best assign devices, since it is aware of the requirements of all jobs being run. Your job may have special needs, however. Suppose you are running in a spooling environment. You have a large job where the format of the printed output is very important. You want to bypass spooling so that you can check your printed output immediately and stop the job, if necessary, to correct the format. Since it is a large job, you do not want it to go first to a spool file and then print if there are formatting errors. You would specify the physical address of a real (rather than a virtual) printer, like this:

```
// DVC 20,160
```

You may assign a real device and bypass spooling without specifying its physical address if you use the *REAL* parameter. The following statement, for example, allows you to request any real printer:

```
// DVC 20,REAL
```

If you use the *addr* parameter to request a specific tape or disk device, be sure the volume you want is not mounted on another unit. The // UID job control statement can be used to assign workstations by physical address. Refer to "Ensuring that Workstations are Connected to a Job" later in this section for more information.

Is This Device Needed for This Particular Run?

Sometimes, all the peripheral devices normally used by the job are not absolutely needed. You may have a case where a job normally produces print and tape output. Your system administrator needs the print output in a hurry, but is not worried about the tape output at this time. If necessary, the job can be rescheduled to produce the tape output.

Our control stream has device assignment sets for tape and print files. In the DVC job control statement of the device assignment set for the tape file, we can use the *OPT* parameter. This indicates that the peripheral device is optional; it is not essential to the running of the job. If it is not available at the time the job is put into execution, all references to this device are bypassed.

Added to our DVC job control statement for the tape output file, it would be coded as follows:

```
// DVC 100,OPT
```

Different Volumes on the Same Device

Within a job step, job control *normally allocates one device for each logical unit number* specified in the control stream. You might, however, have several different volumes to be processed serially within the same job step. This could require several different devices and your job would not be run until all the devices are free. You can suppress job control's check for one volume per logical unit number within a single job step and reuse the same device serially by specifying *IGNORE* on the DVC statement. Since *IGNORE* reduces the number of peripheral devices a job needs, it increases the chances of your job being run sooner.

If the first occurrence of a logical unit number does not specify *IGNORE* in the DVC statement, all subsequent references to that logical unit number must have *IGNORE* specified in the DVC statements.

If you use the *IGNORE* parameter, processing for the first volume must be completed before the second volume is needed, and so forth.

A typical application for the *IGNORE* parameter might be a program that takes information from a tape file, updates it with information from a card file, and creates a new tape. But a job is scheduled that lasts most of the day, and it uses all but one of the installations's tape drives. Since you need two tape drives, you would have to wait until that job was finished. However, you wrote the program so that it reads the input tape file completely, updates the information, and then writes it out to a new tape. Since the processing of the tape volume containing the input file is finished before the program creates the new tape file, you can use the same device by using the *IGNORE* parameter of the DVC job control statement in the device assignment set for the next file to be processed (the output file, in this case).

The *IGNORE* parameter tells the system to disregard the fact that there already has been a device assignment set for this logical number in this job step.

Suppose the input file is on a tape with a volume serial number of TAP111, a file identifier of FIRST, and the file name for the input file is MASTIN. The output file will be on a tape volume with a volume serial number of TAP222, have a file identifier of SECOND, and a file name of MASTOUT. The logical unit number we are going to use is 101.

The device assignment sets for the input and output files would be:

```
// DVC 101
// VOL TAP111
// LBL FIRST
// LFD MASTIN
// DVC 101,IGNORE
// VOL TAP222
// LBL SECOND
// LFD MASTOUT
```

When you use this feature of job control, make sure you inform the operator of the tape mounting sequence.

Users of the Unisys sort/merge routine will find the *IGNORE* parameter useful on tape sort applications that use tape volumes as input, work areas, and output.

When a job consists of more than one job step, the system assumes that the first device assignment set for a logical unit number will be used in subsequent job steps until a new device assignment set for the same logical unit number occurs. For instance, if you wanted to use the tape file with a volume serial number of TAP222 in the next job step, you would have to specify the following device assignment set at the beginning of the new job step:

```
// DVC 101
// VOL TAP222
// LBL SECOND
// LFD xxxx (this depends on your program)
```

Otherwise, the system assumes the tape with a volume serial number of TAP111 is to be used.

Multiple Volumes in a File? Use Alternate Devices to Decrease Operator Setup Time

The file is large - in fact, so large it needs four tape volumes to hold it. When the program uses four tape volumes, the operator can mount them, one at a time, on the device associated with the logical unit number on the DVC job control statement. When a volume is processed, the operator removes it from the device and mounts the next volume on the device. Meanwhile, processing time is wasted while the system waits for the new volume to be mounted. The operator must do this for every volume of the file.

One way of avoiding this is to use the *ALT* parameter on the DVC statement. This allows you to alternate the same logical unit number between two devices, provided that two devices of the same type are available. One device uses the logical unit number while the first volume is being used, then the logical unit number switches to the other device for the next volume. After the second volume is finished, and if there are any more volumes in the file, the logical unit number is switched back to the first device, and so on, until all volumes are used. In this way, the operator can mount two tape volumes, on two different physical devices associated with a logical unit number, in their proper sequence. When the first volume is finished, the system switches to the device containing the second volume. Meanwhile, the operator can unload the first volume and mount the third volume on the device. In this way, no time is wasted because of setup time. All alternate devices must be of the same type. This is especially helpful when small tape reels are used. Note that alternating is restricted to the boundaries of one job step, and that if only one device is available, a job will execute with only one device (even though *ALT* is specified).

The *ALT* parameter of the DVC job control statement doesn't work correctly if it is used more than once in a jobstream. A separate drive is allocated for each *ALT*, and if there are insufficient drives to accommodate all of the *ALT*s, only one drive is allocated even if two drives are available. If the *ALT* function is needed more than once in a jobstream, the following job control can be used:

```
// DVC 90 // VOL A
// DVC 91 // VOL B
```

Assume a job has four tape volumes, using logical unit number 100. You can switch between the two physical devices associated with logical unit number 100 by coding the DVC job control statement as follows:

```
// DVC 100,ALT
```

Of course, the VOL job control statement must be modified to indicate the volume serial numbers of the four different tape volumes. We'll discuss the use of optional parameters for the VOL job control statement later. Briefly, the following example is how multiple volume serial numbers are coded.

```
// DVC 100,ALT
// VOL T11111,T22222,T33333,T44444
```

To ensure that alternation occurs between devices, you may explicitly declare two devices in your job control stream. This means you'll have two DVC statements, each specifying a different logical unit number. Consider the following example:

```
// DVC 100
// VOL T11111,T33333
// DVC 101
// VOL T22222,T44444
```

In this case, the operator can always alternate between the two devices specified by the logical unit numbers 100 and 101, until all volumes are used.

Users of the sort/merge routine will find it helpful to alternate when sorting many tapes with the same label on a master tape.

Ensuring that Workstations Are Connected to a Job

You can use the *REQ [(n)]* parameter of the DVC statement or the UID job control statement when you want to ensure that workstations are connected to a job.

REQ tells the system that the workstations you've specified through the *nnn[(n)]* parameter of the // DVC statement are required and must be connected (using the workstation CONNECT command) for the job to begin execution. You can further tailor the DVC statement by specifying that only a certain number of the workstations must be connected before the job is executed. You do this with the *(n)* portion of the *REQ* parameter. If you prepare your statement like this:

```
// DVC 200(8),REQ(1)
```

it tells the system that eight workstations can be connected to the job and that one of the eight is required and must be connected for the job's execution to begin.

Notes:

1. *The* (n) *portion of the* nnn *parameter and the* REQ>(n)] *parameter are used to assign workstations only. Up to 255 workstations can be assigned to a single workstation file.*

2. *The* nnn *parameter of / / DVC is used differently for workstations than for other devices. If you specify the logical unit number 200 (any workstations) and tailor the specification by using the* (n) *portion of the* nnn *and* REQ *parameters, multiple workstations (of any type) are assigned to the job.*

Recall from "The UID Job Control Statement" in Section 3 that the UID statement is used if you want specific workstations connected to a job automatically. This is done before the job's execution begins (if the workstations specified have not already been connected using a CONNECT command). You identify a particular workstation by its user-id, device address, or both. For example:

```
// DVC 200
// UID WS1,(018),WS2(019)
// LFD WKSTFILE
```

The UID statement in this example indicates that the following three workstations will automatically be connected: any workstation logged on with a user-id of WS1, the workstation with the address 018 and logged on with any user-id, the workstation with the address 019 and logged on with a user-id of WS2. If these three conditions are not satisfied, the job remains in the scheduling queue. Remember that workstations specified in the UID statement are required; therefore, the job will not run until these devices are available (that is, logged on).

Although the *(n)* portion of the *nnn* parameter and the *REQ [(n)]* parameter are generally unnecessary in the DVC statement when the UID statement is used, you may encounter a special situation. For example:

```
// DVC 200(4)
// UID WS1,WS2
// LFD WKSTFILE
```

The DVC statement indicates that the job can use up to four workstations. The two identified in the UID statement are required and, provided they're logged on, will automatically be connected at execution time. Two more workstations (any two) can optionally log on and then connect to the job with the CONNECT command.

Remember, you can specify YMAS as a user-id to assign the job's master workstation to a job.

Specifying a Remote Disk File

To indicate that a disk file is located at a remote host in a DDP network, specify the *HOST=host-id* keyword parameter on the // DVC statement. The host-id is one to four alphanumeric characters long and identical to the label-id of the LOCAP macroinstruction in your ICAM network. $HOST (in place of a host-id) indicates that the file is located at the job's remote originator (the remote host that initiated the job).

Consider the following:

```
// JOB MYJOB
      .
      .
      .
// DVC 50,HOST=A123
// VOL D000028
// LBL FILE1
// LFD REMOTE
      .
      .
      .
// EXEC PROGA
/&
```

The DVC statement in the preceding device assignment set means that the disk file is located at host A123.

Note: *The host you specify (using either a host-id or $HOST) must be a remote host. If you specify a local host, you'll receive a data management error message (DM21 INVALID DEVICE ASSIGNMENT).*

For information about DDP facilities, see the *Distributed Data Processing Programming Guide*, 7004 4508. For more information about the originator, see the OPTION ORI statement in "Selecting Optional Features" in Section 6. See "How Job Control Statements are Presented" in Appendix A for information about coding job control statements containing positional as well as keyword parameters.

Indicating Use of the DDP Program-to-Program Facility

If your program is written in BAL and uses consolidated data management macros, you can use DDP's program-to-program facility. In its simplest form, this facility allows a program at one host (the primary) to initiate communication with a program at another host (the surrogate). The job control stream for each program participating in this simple conversation must contain a DVC PROG job control statement. Used in place of // DVC, // DVC PROG begins the device assignment set for the program-to-program type file. The format is:

```
//[symbol] DVC PROG [,job-name][,HOST=host-id]
```

You can specify one // DVC PROG statement in any single-step job control stream. (A single-step job requests the execution of only one program.) The device assignment set must contain a // LFD statement and may contain a // LBL statement for cataloging purposes.

The *job-name* parameter identifies the name of the other participant in the program-to-program communication. For example, when specified in the // DVC PROG statement for the primary, *job-name* identifies the surrogate. When specified in the // DVC PROG statement for the surrogate, *job-name* identifies the primary. This parameter is required in the // DVC PROG statement for the primary, but is optional in the // DVC PROG statement for the surrogate.

The *HOST=host-id* parameter simply identifies a particular host in a DDP network. The host-id is one to four alphanumeric characters long and identical to the label-id of the LOCAP macroinstruction in your ICAM network. You use $HOST (in place of a host-id) to indicate the originator (the host that initiated the job). Consider the following control streams:

```
HOST AAAA                              HOST BBBB
// JOB MYJOB                           // JOB YOURJOB
      .                                      .
      .                                      .
      .                                      .
// DVC PROG,YOURJOB,HOST=BBBB          // DVC PROG
// LFD THISFIL                         // LFD THATFIL
      .                                      .
      .                                      .
      .                                      .
// EXEC PROG1                         // EXEC PROG2
/&                                     /&
```

The // DVC PROG statement in MYJOB indicates that communication can only be established with PROG2 - the program identified in YOURJOB at host BBBB. PROG1, in this case, must act as the primary. The // DVC PROG statement in YOURJOB means that PROG2 is a surrogate in the program-to-program communication with PROG1. PROG2 can also act as the surrogate when other job control streams declare // DVC PROG,YOURJOB,HOST=BBBB. Now consider the following:

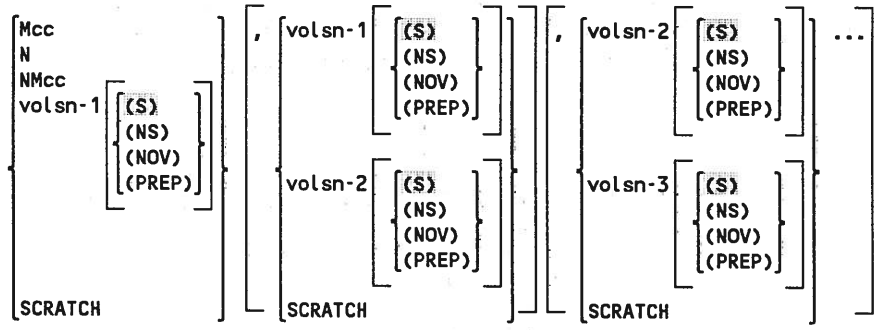
```
HOST AAAA                              HOST BBBB
// JOB MYJOB                           // JOB YOURJOB
      .                                      .
      .                                      .
      .                                      .
// DVC PROG,YOURJOB,HOST=BBBB          // DVC PROG,MYJOB,HOST=AAAA
// LFD THISFIL                         // LFD THATFIL
      .                                      .
      .                                      .
      .                                      .
// EXEC PROG1                         // EXEC PROG2
/&                                     /&
```

These two job control streams indicate that only PROG2 at host BBBB and PROG1 at host AAAA can communicate with each other. The first program to open the program-to-program type file is considered the primary.

Although primarily intended for communication between programs executing on different hosts, the program-to-program facility can be used between programs executing on the same host. For more information about DDP's program-to-program facility, see the *Distributed Data Processing Programming Guide*, 7004 4508.

More Information about the Characteristics of Your Volumes

We have used the VOL job control statement to specify the volume serial number. It also has additional parameters for further identifying each volume to the system. Once again, its format is:



Refer to this format when each new parameter is introduced.

Notes:

1. *If all the volumes used to contain a multivolume file are going to be online simultaneously (mounted on different devices during the course of a single job step), the* NOV *and* PREP *options, if used, must be specified for each volume.*

2. *The DVC specification in the device assignment set is used to determine if more than one device is being used.*

3. *In a multivolume file, if the individual volumes are mounted on separate devices, the* NOV *and* PREP *options can be specified only for the individual volumes.*

4. *If the* PREP *option is specified for any volume in a multivolume file sequentially mounted on one device, it applies to all volumes in a multivolume file.* NOV *must be specified for the last volume in the file for it to apply to all volumes in the file.*

More Than One Volume in a File

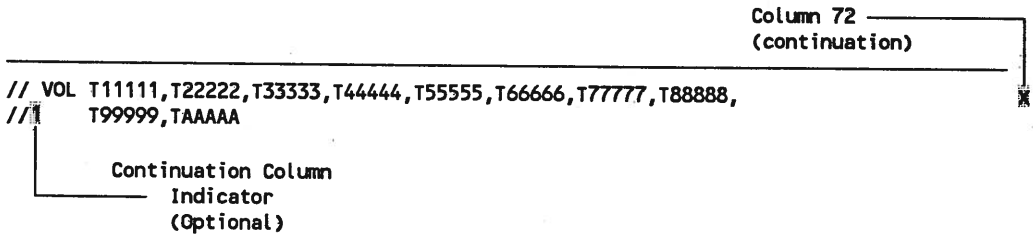
When we discussed the ALT parameter of the DVC job control statement, it was stated that all volumes in the file must be specified on the VOL job control statement of the device assignment set for the two devices sharing a logical unit number. (See "Multiple Volumes in a File? Use Alternate Devices to Decrease Operator Setup Time" earlier in this section). The example given was:

```
// DVC 100,ALT
// VOL T11111,T22222,T33333,T44444
```

Each group of numbers specified on the VOL job control statement (T11111, T22222, etc.) represents the volume serial number of the volumes in the sequence in which they are mounted.

Remember, whenever there is more than one volume in a file, notify the operator of the mounting sequence.

If more than eight volume serial numbers are listed, a nonblank character must appear in column 72 of the VOL job control statement and one or more continuation cards (Appendix A) must follow. For example:



You can also specify multivolume files by using separate VOL control statements, like this:

```
   .
   .
   .
// VOL T11111
// VOL T22222
// VOL T33333
   .
   .
   .
```

This method has an advantage over the continuation method in that you can change VOL specifications easier if they are coded separately.

The VOL statement's *(NOV)* and *SCRATCH* parameters provide you with the option of not listing each specific volume serial number in a multivolume file. For further discussion of these parameters, see "Ignoring or Changing the Volume Serial Number" later in this section.

Special Characteristics of Tape Volumes

Tape volumes have certain mode characteristics, such as bytes per inch, parity, and the number of tracks (7 or 9). The mode characteristics of tape volumes are specified using the *Mcc* parameter. The values for *cc* are given in Table 4-1.

Suppose you are using a UNISERVO® 12 magnetic tape subsystem, and the tape volume is 7-track, 200 bytes per inch, even parity, with the translate and convert features off. The mode setting is 20 and it would be coded as M20. The volumes being used are coded as the remaining parameters.

```
// VOL M20,T11111,T22222
```

If the *Mcc* parameter is omitted, the mode settings specified at system generation time are used.

If your supervisor supports block numbering and you have specified *BKNO=YES* in your program's file definition macroinstruction (or *BC$CLNM* for PIOCS), data management will check block numbers on input tape volumes or write sequential block numbers on output tape volumes. If you want to suppress block numbering or checking during *initialized processing*, you use the *N* parameter on the VOL job control statement. Initialized processing includes use of the *TPREP* utility routine or the *PREP* option on the VOL statement as well as processing of input or output files with nonstandard labels or no labels. When you specify *N*, block numbering is suppressed for all volumes included on the VOL statement. For noninitialized processing, the *N* parameter is ignored. That is, if your supervisor supports block numbering and you have specified it in the file definition macroinstruction, you cannot suppress checking or writing of block numbers by using the *N* parameter. For details about block-numbered tapes, see the *Consolidated Data Management Macroinstructions Programming Guide*, 7004 4607.

For example, to suppress block numbering on two tape output volumes with volume serial numbers of T11111 and T22222, code as follows:

```
// VOL N,T11111,T22222
```

When both the *N* and *Mcc* parameters are used, code them as one parameter. For example:

```
// VOL NM20,T11111,T22222
```

UNISERVO is a registered trademark of Unisys Corporation.

Table 4-1. Mode Characteristics

Tape	cc	Bytes per Inch	Parity	Translate Feature	Convert Feature
UNISERVO 12/16 Magnetic Tape Volumes					
7-track	10	200	Odd	Off	On
	20	200	Even	Off	Off
	28	200	Even	On	Off
	30	200	Odd	Off	Off
	38	200	Odd	On	Off
	50	556	Odd	Off	On
	60	556	Even	Off	Off
	68	556	Even	On	Off
	70	556	Odd	Off	Off
	78	556	Odd	On	Off
	90	800	Odd	Off	On
	A0	800	Even	Off	Off
	A8	800	Even	On	Off
	B0	800	Odd	Off	Off
	B8	800	Odd	On	Off
9-track	C8	800	Odd	Off	Off
	C0	1600	Odd	Off	Off
UNISERVO 22/24 Magnetic Tape Volumes					
9-track	C8	800	Odd	Off	Off
	C0	1600	Odd	Off	Off
UNISERVO 26/28 Magnetic Tape Volumes					
9-track	C0	1600	Odd	Off	Off
	D0	6250	Odd	Off	Off
2145 and BT3200 Magnetic Tape Volumes					
9-track	C8	800	Odd	Off	Off
	C0	1600	Odd	Off	Off
	D0	6250	Odd	Off	Off
5073 Magnetic Tape Volume					
18-track	NA	38000	Odd	Off	Off

Extending Your Tape Volumes

If you recall, when we were assigning file names to files, we used the LFD job control statement (see "Assigning a Logical File Name to the File" in Section 3). Well, now we'll use this same statement to extend our file. Once again, here is the format:

```
//[symbol] LFD  {filename }  [, {N}]  [, {EXTEND}]
                {*filename}    {8}     {INIT  }
```

Looking at the format, we see the optional parameter EXTEND. The EXTEND parameter lets us add information to the present end of a tape or disk file, provided our program allows us to do so and the following job control conditions are met:

- The *PREP* option is not specified on the VOL job control statement.
- The file being extended is the only file on the volume.
- The file uses standard labels.
- The file specified is an output file.

The following example shows the use of the LFD statement to extend the file ADDR1:

```
// LFD ADDR1,,EXTEND
```

The following device assignment set, which includes this LFD statement, illustrates how to extend a file (MAST) on volume T11111.

```
// DVC 100
// VOL T11111,T22222,T33333
// LBL MASTER
// LFD MAST,,EXTEND
```

If you expect additional volumes will be needed to accommodate extension of the file, you can add the volume serial numbers of any tapes to the VOL statement. The following device assignment set indicates that the extension of MAST will result in a multivolume file.

```
// DVC 100
// VOL T11111,T22222,T33333
// LBL MASTER
// LFD MAST,,EXTEND
```

If you are extending a tape file that already has multiple volumes, your VOL statement has to specify only the last volume containing the file plus any additional volumes. You must include the serial number of the file's first volume as the second parameter (*file-serial-number*) of the LBL statement. See "Multivolume File? Assign Each Volume a File Serial Number" later in this section for more information. Suppose, for example, the file MAST is on volumes T11111, T22222, and T33333. If you expect the file's extension to require an additional tape volume, you would code the device assignment set as follows:

```
// DVC 100
// VOL ,,,T33333,T44444
// LBL MASTER,T11111
// LFD MAST,,EXTEND
```

The volume serial number T11111 is required to identify T33333 and T44444 (the new volume) as being part of the same file.

Note: *When referencing multivolume files on the VOL statement, any undeclared volume serial numbers must be represented with commas. Additionally, if* Mcc, N, *or* NMcc *are not specified for the first positional parameter, you must supply a comma. In the VOL statement in our previous example (/ / VOL ,,,T33333,T44444) the first comma represents the first positional parameter. The second and third commas represent T11111 and T22222, respectively.*

The *Consolidated Data Management Macroinstructions Programming Guide*, 7004 4607, also contains information about extending tape files.

Sharing Disk Volumes

More than one job can share a disk volume. But suppose you are updating a file that will be accessed by other user jobs. They should not access the file until the update is completed, or else their output would not be the most current. You can indicate, on the VOL job control statement, that the disk volume is nonsharable; thus the file cannot be accessed. The system will not allow other jobs to begin execution until your job has finished the update.

Assume the file being updated has a volume serial number of DSK083 and it should be nonsharable. You indicate this by using the (*NS*) parameter. The parentheses are coded as part of the parameter, and there is no comma separating the volume serial number and the (*NS*) parameter. This is coded as:

```
// VOL DSK083(NS)
```

When there is more than one volume in the file (DSK083, DSK076, and DSK093, for instance) and they are all nonsharable, code it in this manner:

```
// VOL DSK083(NS),DSK076(NS),DSK093(NS)
```

Sharable disk volumes are the default condition.

Ignoring or Changing the Volume Serial Number

Through the VOL job control statement, you have the option of ignoring volume serial numbers. This allows the use of any available volume or one with an unknown volume serial number.

For example, you want to create a tape file. The operator is told that you can mount any unused tape with a volume serial number (it does not contain a permanent file, and you do not want a scratch tape because you are creating this file for other jobs). Since you don't know what tape the operator will use, you don't know the volume serial number for your VOL job control statement. By using the *(NOV)* parameter and a dummy volume serial number, you can use a volume without specifying the correct volume serial number.

Code it this way:

```
// VOL DUMMY(NOV)
```

Notice that there is no comma separating the dummy volume serial number and the *(NOV)* parameter. The parentheses are part of the parameter.

After the job is processed, you should be informed, in some manner, of the volume serial number of the created tape. This volume serial number must be used on the VOL job control statement for any subsequent job using this tape volume.

Notes:

1. *The volume serial number DUMMY is used here just as an example. You can use your own dummy volume serial number, but if it isn't a unique one, keep the following in mind: if two or more jobs use the same dummy volume serial number for a disk volume, these jobs can run concurrently and share the same disk volume. This may or may not be desired. If a job uses the* (NOV) *parameter with a dummy volume serial number for one type of volume (e.g., a tape), and a second job uses the* (NOV) *parameter with the same dummy volume serial number for another type of volume (e.g., a disk) or, for another nonsharable volume (e.g., another tape), the second job is not executed until the first job is finished.*

2. *If you specify a volume serial number and the volume with that serial number is mounted on a device before the job goes into execution, that volume (and the device on which it's mounted) is used even if you've specified a different physical device number on the DVC statement. If, however, you use // VOL DUMMY (NOV) the physical request is not ignored.*

With the VOL statement's *SCRATCH* parameter you can specify a multivolume file without listing each volume's serial number. Consider this example:

```
// VOL VSN1,VSN2,SCRATCH
```

This statement declares a multivolume file and requests that the volume VSN1 with the serial number be mounted first and volume VSN2 be mounted second. The *SCRATCH* parameter indicates that after VSN2, any volumes can be mounted.

When you request scratch processing, a message to mount a scratch volume is displayed (after any explicitly requested volumes have been taken care of) on the system console. Any volume will then be accepted until the end of file. Remember, because data management cannot check for the proper serial numbers at this point, you should make sure that the operator knows exactly what volumes to mount and the sequence to mount them in.

The *SCRATCH* parameter can also be used alone. For example:

```
// VOL SCRATCH
```

This statement requests scratch processing for all volumes in the file.

You may want to use the *SCRATCH* parameter if you have a 20-volume diskette file for example, and you don't want to list 20 volume serial numbers in your job control stream. When coding job control statements remember that the *SCRATCH* parameter can only appear *once* in a VOL statement and it is always the *last* parameter specified.

You can also suppress checking of volume serial numbers for all volumes of a multivolume file by specifying *NOV* in the VOL statement for the last volume of the file.

You can change a volume serial number by specifying the new volume serial number followed by the *(PREP)* parameter. You can also use this to assign a volume that currently does not have a volume serial number (scratch volume or a new volume). Any information that is currently on the volume is scratched.

Your job creates an output tape that you want saved and to be assigned the volume serial number of TAP099. It would be coded as follows:

```
// VOL TAP099(PREP)
```

Once again, there is no comma separating the new volume serial number and the *(PREP)* parameter. The parentheses are part of the parameter.

Notes:

1. *Be very careful when you use the* PREP *option on a file to be processed by the librarian. When you specify the* PREP *parameter, the tape is prepped every time it is opened as output. The librarian closes output tape files whenever they are to be used as input and then reopens them as output. If a tape file is to be reused as an output file within the same job, the librarian closes it as input and reopens it as output. This reopening causes the tape to be reprepped (if* PREP *was specified), thereby effectively erasing all the information previously produced. Therefore, use this option only if the file will be output only, or output, then input. Otherwise, use the* TPREP *utility to prep the file. The* PREP *option cannot be suppressed. You must redefine the tape file without specifying the* PREP *option on the VOL statement.*

2. *For multivolume files, if* PREP *is specified for any of the volumes, all volumes in the file are prepped.*

3. SCRATCH *lets you mount additional tape volumes (unlimited processing); however, these additional volumes are not prepped if* PREP *is specified. If they must be initialized, use the* TPREP *utility routine.*

Multivolume Files Online Simultaneously

You may have an application, a data base system, for example, that requires a large multivolume file, with the volumes online simultaneously (since they are accessed in a random manner). Suppose you have a 3-volume file (volumes A B, and C). You would code the device assignment set for the file like this:

```
// DVC 50
// VOL A
// DVC 51
// VOL B
// DVC 52
// VOL C
// LBL DATA
// VMNT=NO
// LFD BASE
```

More Information on Disk and Format-Label Diskette File Allocation

You use the EXT job control statement to allocate the space (extent) needed by a disk or format-label diskette file. The format is:

```
//[symbol] EXT {MI}  [,{C }]  [,{inc}]  [,{addr    }]  [,{mi     }]
             {ST}    [{CF}]    [{0  }]   [{Tccc:hh }]   [{(bi,ai)}]
                      {F }     {1  }     {BLK      }
                                         {TBLK     }
                                         {CYL      }
                                         {TRK      }
                                         {OLD      }

[,{mj     } ,...][,OLD][,FIX][,NTERM]
  {(bj,aj)}
```

All the parameters are optional.

The File Type

With the first parameter of the EXT statement, specify the type of file you're allocating the extent for.

MIRAM files are discussed in the *Consolidated Data Management Programming Guide*, UP-9978. System access technique files are described in the *Supervisor Technical Overview*, UP-8831.

For the EXT job control statement, you can specify MIRAM (multiple indexed random access method) files, indicated by coding MI, or SAT (system access technique) files, indicated by coding ST.

If, for example, you wanted to use the multiple indexed random access method, you would code:

```
// EXT MI,C,,CYL,1
```

Formatting a File and Using Contiguous Space

Files are formatted using the parameters *F*, *BLK*, and *(bi,ai)*. These indicate that you are going to format the file, *F*, in terms of blocks, *BLK*, to a certain length, *(bi,ai)*. The *bi* indicates the number of bytes in the block, and the *ai* indicates the number of blocks in the file. Files can be formatted only in terms of blocks.

Suppose that you have a MIRAM file to allocate and it contains 5000 blocks, each 472 bytes long. Refer to the format of the EXT job control statement to see the correct position of each of the parameters you are going to see: *MI*, *F*, *BLK*, and *(bi[,ai])*. It would be coded as follows:

```
// EXT MI,F,,BLK,(472,5000)
```

You can set up your program to access a particular block (or blocks) within the file.

The EXT job control statement is also used to allocate space contiguously. When you allocate a file, there may not always be a single extent (a single contiguous area) available on the disk or format-label diskette. Suppose, for example, you need 10 cylinders for a file but there aren't 10 contiguous cylinders anywhere on the volume. Instead, there are 2 contiguous cylinders in one place, 3 in another, and 5 more in another. If this is the case, OS/3 disk space management divides the file among 3 different areas resulting in a 3-extent file. The *C* parameter (shown as one of the choices in the second parameter in the format) can prevent this from happening so that if enough contiguous cylinders cannot be found, the file won't be allocated.

Note: *A single file on disk or format-label diskette can have no more than 16 physical extents. If a file already occupies 16 extents but more are needed, you must use another volume even if sufficient space is still available on the original volume. (The file becomes a multivolume file.) A VTOC listing of the volume will tell you in advance how many extents an existing file occupies. Just remember there can be only 16 extents for a single volume file, 32 extents if the file occupies two volumes, 48 for three volumes, and so on.*

When you specify the absolute starting address (the *addr* parameter, explained in "Terms of Allocation" later in this section), you must allocate contiguously. You must also specify *addr* in hexadecimal. The use of continguous space reduces file access time, thus reducing job processing time.

To allocate a MIRAM file that contains 1000 blocks, each containing 1024 bytes, and you want contiguous disk space, code as follows:

```
// EXT MI,C,,BLK,(1024,1000)
```

The *C* and *F* parameters can be combined to form one parameter. Use this if you want contiguous, formatted disk space. A comma is not needed to separate these parameters.

For example, to allocate 300 blocks, each 256 bytes in a contiguous area, using the multiple indexed random access method, code the following:

```
// EXT MI,CF,,BLK,(256,300)
```

Notice that we've been coding *BLK* in these examples. *BLK*, however, is the default condition - you could have coded the last example like this:

```
// EXT MI,CF,,,(256,300)
```

Your Disk or Format-Label Diskette File Needs More Space

When a disk or format-label diskette file is allocated, a certain area is reserved for a file. It is possible, however, the amount that you estimate may not be enough. There may be more information than you realized; an update of the file made it larger than originally intended, or, you may be replacing existing information with new information (this requires the use of the *INIT* parameter of the LFD job control statement, which is explained in "The Expiration and Creation Date of the File" later in this section). This new information may require more space than you had previously allocated.

Job control can extend the requested area, if necessary. Let's say you're setting up a file to contain 700 or 800 entries for an accounts payable procedure, and you estimated the file would need 100 blocks, each 256 bytes in length. Since this is only an estimate, you can use a parameter in the EXT job control statement to allocate more space if it is needed. This is called dynamic extension. If it isn't needed, it isn't allocated. In this way, you don't waste space by allocating more than necessary.

The parameter used to provide this dynamic extension is the third parameter group in the format. The *inc* parameter is the amount of additional space that you request. This dynamic extension is in terms of cylinders.

Specifying 0 indicates you do not want to allow for dynamic extension of the file. Use this when you want to limit the amount of information placed in the file. If nothing is specified, by default, one cylinder is generated.

Assume, for the accounts payable application, that we estimated 100 blocks, each 256 bytes long, on a formatted, MIRAM file. We want two additional cylinders if dynamic extension is necessary. The coding would be:

```
// EXT MI,F,2,,(256,100)
```

Terms of Allocation

We've already covered some allocation terms in previous examples: *BLK* for allocating in terms of blocks and *CYL* for allocating in terms of cylinders. With the *addr* parameter you can also specify the absolute cylinder address in hexadecimal at which the file is to begin. When you do this, allocation is in terms of cylinders.

Note: *The absolute address can be specified in decimal by coding D'number', or hexadecimal by coding X'number'. Any number not preceded by D or X and enclosed in single quotes is considered hexadecimal.*

Let's say you need one MIRAM file, allocated contiguously, allowing 5 cylinders for dynamic extension, and it must start at cylinder 78. Code it:

```
// EXT MI,C,5,4E
```

Do you recall specifying the amount of blocks needed for the file? One of the examples looked like this:

```
// EXT IS,C,,BLK,(1024,1000)
```

Specifying (1024,1000) told job control how many blocks to allocate: 1000. When you specify allocation in terms of cylinders or by absolute address, you must indicate how many cylinders to allocate for the file by using the *mi* parameter.

If you wanted 10 cylinders, it would have been coded as:

```
// EXT MI,C,5,CYL,10
```

The *TRK* parameter allows you to allocate disk and format-label diskette files in terms of tracks. The *TBLK* parameter allows you to allocate a file in blocks by track rather than in blocks by cylinder (*BLK* parameter). The *Tccc:hh* parameter is similar to the *addr* parameter because it lets you specify the absolute hexadecimal *(X'number'* or *number)* or decimal *(D'number')* starting address of the file. The address, however, is a track address in cylinder/head format and the allocation is in terms of tracks, not cylinders. For more information about file allocation by track, see the *Consolidated Data Management Macroinstructions Programming Guide*, 7004 4607.

Note: *You cannot allocate a disk file or format-label diskette file by track (using* TRK, TBLK *or* Tccc:hh*) when creating MIRAM files with IRAM characteristics.*

Remember that when you specify *CYL*, *addr*, *TRK*, or *Tccc:hhh*, you must specify the number of cylinders or tracks with the *mi* parameter.

Allocation Amounts

The parameters for indicating the amount of space wanted were shown, indirectly, when we discussed formatting and terms of allocation. These were coded as the fifth parameter, *mi* or *(bi,ai)*.

The *mi* parameter is used with either the *CYL*, *addr*, *TRK*, or *Tccc:hh* parameter, and indicates the amount of cylinders or tracks needed by the file. These were covered in the last example of "Terms of Allocation" in this section.

The *(bi,ai)* parameter is used with the *BLK* or *TBLK* parameter for allocating in terms of blocks (rounded up to cylinders or tracks, respectively). Remember *BLK* is the default parameter so you don't need to specify it. The *bi* indicates the amount of bytes in the block, and the *ai* indicates the number of blocks in the file. For instance, this example

```
// EXT MI,C,5,CYL,10
```

indicates an allocation of 10 cylinders, while either of these examples

```
// EXT MI,F,10,,(256,100)

// EXT MI,F,10,BLK,(256,100)
```

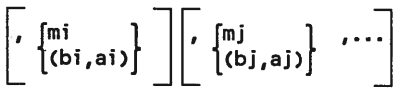
indicates an allocation of 100 blocks, each 256 bytes in length.

You can specify any number of separate disk areas (extents) for an individual file. A reason for using several different extents for a single file would be to decrease data access time, thus reducing processing time. Assume the program is designed such that the file can be divided into two different extents. The first extent contains data used only by the first part of the program; the second extent contains data used only by the second part of the program.

For instance, the first extent contains hourly pay rates for calculating gross pay, and the second extent contains payroll deductions to subtract from the gross pay to get the net pay. Once the gross pay is calculated, the first extent is no longer needed; the program will not need this information again. It only needs the deduction information in the second extent to finish processing. In this way, one large extent is divided into two smaller extents, reducing the amount of access arm movement for the disk unit.

For example, you have a file divided into two different extents. The total size of the file is 20 cylinders. The first part of your program uses 12 cylinders, and the second part needs 8 cylinders. They can both be specified on the same EXT job control statement. The information in the first four parameters applies to both extents in the file.

Look at this portion of the format:



The *mj* parameter means the same as the *mi* parameter and the *(bj,aj)* parameter means the same as the *(bi,ai)* parameter. The only difference is that *mj* and *(bj,aj)* are used for additional extents in the file. So, we could code the two extent files (12 cylinders and 8 cylinders) as:

```
// EXT MI,C,1,CYL,12,8
      └──────┘ │ │
          1    2 3
```

Notes:

1 This applies to both extents.

2 This is the allocation for the first extent.

3 This is the allocation for the second extent.

If you allocated in terms of blocks, with the first extent occupying 300 blocks, each 256 bytes in length, and the second extent occupying 700 blocks, each 256 bytes in length, it would be coded as:

```
// EXT MI,C,1,BLK,(256,300),(256,700)
      └──────┘ └──────┘ └──────┘
          1        2        3
```

Notes:

1 This applies to both extents.

2 This is the allocation for the first extent.

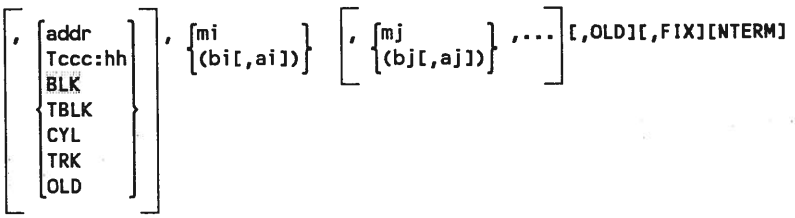
3 This is the allocation for the second extent.

You can also specify separate extents for an individual file by coding separate EXT statements, as we did when we coded separate VOL statements for a multivolume file. Refer to "More Than One Volume in a File" earlier in this section for details. You have coded separate extent specifications for our previous example, like this:

```
.
.
.
// EXT MI,C,1,BLK,(256,300)
// EXT MI,C,1,BLK,(256,700)
.
.
.
```

Changing the Specifications of a Previously Allocated File

Sometimes, you may want to change some of the information pertaining to a previously allocated file. Use the *OLD* parameter to do this. The following portion of the EXT job control statement format shows *OLD* as either the fourth or seventh parameter:



When coded as the fourth parameter, *OLD* means you want to change the automatic allocation amount for dynamic extension (the third parameter) for a previously allocated file. Suppose you specified one cylinder when a MIRAM file was originally created. To change this specification to five, you code the EXT statement as follows:

```
// EXT ,,5,OLD
```

You can omit the first and second parameters, since they are ignored if specified.

When *OLD* is coded following the allocation amount (*mi*, *mj*, etc.), it increases the original allocation amount for your extents.

Let's assume your file was originally a 30-cylinder, sequential file and you discover you really need 50 cylinders. To obtain these extra 20 cylinders, you can change the allocation amount for the file by using this EXT job control statement:

```
// EXT  ,,,CYL,20,OLD
```

When changing the allocation amount, you may omit the first, second, and third parameters since they are ignored, if specified.

Allocating Space in the Fixed-Head Area of Your 8417 Disk

If you have an 8417 disk subsystem with a fixed-head feature, use the *FIX* parameter with your EXT statement when you want to allocate the extent in the fixed-head area. See the *Consolidated Data Management Macroinstructions Programming Guide*, 7004 4607, for information about the 8417 fixed-head disk.

No Terminate Option for Insufficient Extent Space

The *NTERM* option, when used, informs you if the extent cannot be allocated because of insufficient disk space or because a specified absolute disk area is already in use (error code 36). Rather than terminating the job, which is what happens without this option, the system displays a JC48 message and waits for either a retry (R) or cancel (C) reply. This allows your operator to evaluate the files currently on the disk and to clear those that are not needed so your job can continue.

Information about Data-Set-Label Diskette File Allocation

To allocate space for a file on data-set-label diskette, include an EXT statement in the device assignment set for the *diskette*.

A data-set-label diskette file is always a 1-extent, nonextendable, sequential file. Therefore, several of the EXT statement parameters and options that we discussed in the preceding section do not apply. To help you avoid confusion, refer to the following EXT statement for data-set-label diskette:

```
//[symbol] EXT MI,C,0,BLK,(bi,ai)[,NDI]
```

Just as for disk, the first parameter of the EXT statement indicates file type. The extent for a data-set-label diskette file must be contiguous and cannot be dynamically extended. So specify *C* for the second parameter and *0* for the third parameter. Space on a data-set-label diskette is allocated by block, so *BLK* and *(bi,ai)* must be specified for the fourth and fifth parameters respectively. Specify the last parameter, *NDI* (non-data-interchange), for all System 80 data-set-label diskettes that are not basic data exchange (BDE) diskettes. If you omit this parameter, it is assumed that you're allocating a BDE diskette (a single-sided, single-density diskette having 128-byte sectors, 26 sectors per track, and 73 tracks.) For more information about the characteristics of data-set-label diskettes, see the *Consolidated Data Management Macroinstructions Programming Guide*, 7004 4607.

The following is an example of an extent statement for a data-set-label diskette file having 100 blocks of 80 bytes each:

```
// EXT MI,C,0,BLK,(80,1000)
```

Using Your File Identifier More Efficiently

So far, the LBL job control statement was used to designate the individual files on a volume by providing a file identifier (*labeling a file*).

We are now going to explain the optional parameters, and a special variation of the file-identifier parameter, that improve file handling efficiency. Once again, the format of the LBL job control statement is:

```
//[symbol] LBL   {file-identifier  }  [, {file-serial-number}]  [,expiration-date]
                 {'file-identifier'}     {VCHECK           }

[,creation-date] [, {file-sequence-number}]  [, {generation-number}]
                    {1                    }     {1                }

[, {version-number}]
   {1             }
```

As each individual parameter is introduced, refer to this format.

But first, we'll describe the special variation of the *file-identifier* parameter. Sometimes, you may not want more than one job to access a particular file at the same time, for example, when it is being updated. If it's a disk file, you can make it lockable by assigning a 6-byte lock ID as a prefix to your file identifier. Ninety-nine lock IDs are available: $LOK01 through $LOK99. The lock ID may be followed by up to 38 characters. The LBL statement for a lockable file might be coded this way:

```
// LBL $LOK15MASTERFILE
```

Once you have assigned a lock ID to the file, it is locked automatically each time it is opened. The type of lock (read-only or write-only) is determined by the ACCESS parameters in your file definition macroinstruction for the file. See the *Consolidated Data Management Macroinstructions Programming Guide*, 7004 4607, for a complete description of the file lock facility.

Multivolume File? Assign Each Volume a File Serial Number

When using a file consisting of multiple volumes, a file serial number can be assigned to identify each volume as being a member of the file. In this way, a volume that is not a member of the file cannot be used.

The file serial number is identical to the volume serial number of the first volume of the file. For instance, there are four volumes in a file, in this sequence:

1. XYZ
2. P10
3. A79
4. TPL

The file serial number for all the volumes in this file would be XYZ.

You use the *VCHECK* parameter to either create a file serial number on output volumes, or to check the file serial number on input volumes. This *VCHECK* parameter instructs job control to use the first volume serial number specified on the VOL statement as the file serial number.

Once again, we have the four volumes, XYZ, P10, A79, and TPL, in that order, in a file. We want to write a file serial number on them. Arbitrarily, the file identifier we are going to use is OUTPUT. Your VOL and LBL statements would look like this:

```
// VOL XYZ,P10,A79,TPL
// LBL OUTPUT,VCHECK
```

If this file was already created with a file serial number (input rather than output), it would be coded the same way. The *VCHECK* parameter writes on output and checks on input.

The *file-serial-number* parameter is also used to write or check the file serial numbers of volumes, but in a slightly different manner.

Again, we have these same four volumes (XYZ, P10, A79, and TPL) in the file. But, you only want to use the last two volumes, A79 and TPL, in that order, on this run. This is a previously created file; when it was created, the *VCHECK* parameter was used, giving a file serial number of XYZ to each volume. If we used the *VCHECK* parameter now, while trying to read only these two volumes, A79 and TPL, job control would use the volume serial number of the first volume specified on the VOL statement, A79, as the file serial number value. Since these volumes were created with a file serial number of XYZ, the job would not run. But, the *file-serial-number* parameter allows you to specify the particular file serial number to use. This case would be coded like this:

```
// VOL ,,,A79,TPL
// LBL OUTPUT,XYZ
```

Note: */When referencing multivolume files on the VOL statement, any undeclared volume serial numbers must be represented with commas. Additionally, if* Mcc, N, *or* NMcc *are not specified for the first positional parameter, you must supply a comma. In the VOL statement in our previous example (/ / VOL ,,,A79,TPL) the first comma represents the first positional parameter. The second and third commas represent XYZ and P10 respectively.*

If either *VCHECK* or the *file-serial-number* parameter is omitted when a multivolume file is created, there is no file serial number for the file, or, if it's a tape volume, there is no VOL1 label.

The Expiration and Creation Date of the File

You can limit the life of files by writing an expiration date with the LBL statement. This date indicates whether or not a file can be deleted by a scratch routine (by using the SCR job control statement, explained in "Scratching Unwanted Files" in Section 6) or by a function of data management. This is coded as the third parameter on the LBL job control statement, and can take either of two forms:

- yyddd

  In this form, *yy* is the year, and *ddd* is the day of the year. For example, February 10th is the 41st day of the year (31 in January, plus 10).

- Rdddd

  In this form, *R* is a constant, and indicates a retention cycle is being used based on the creation date (either the next parameter, or the date set in the system). The *dddd* indicates the amount of days (1-9999).

For instance, you create an output tape with a file identifier of XRAY, and you want it to have an expiration date of the 98th day of 1991. This would be coded as:

```
// LBL XRAY,,91098
```

If you omit the expiration date when *writing* a file, the current date is inserted for you. If you omit it when *allocating* a file, no date is specified and zeros are inserted. If you omit the date and allocate, then write to the file (in the same job step), the current system date is used.

The *creation-date* parameter indicates the date the file is generated. If omitted for a tape file or a disk output file, the date stored in the job preamble is used. If omitted for a disk input file, this field is ignored.

The creation date has only one form: *yyddd*, where *yy* is the year and *ddd* is the day.

If you want a creation date of the file, identified by XRAY, to be the 100th day of 1991, code:

```
// LBL XRAY,,,91100
```

Indicating the Position of the File when Several Are on a Tape Volume

When you place more than one file on a single tape volume, you can indicate each file's position on the tape by assigning sequence numbers. Later, if you want a particular file on that volume, you simply reference the file (in the // LBL statement) by its identifier and sequence number. You can only assign sequence numbers to standard labeled tape files.

When you create a tape file, you use the fifth positional parameter of the // LBL statement (*file-sequence-number*) to assign a sequence number. The following statement, for example, assigns a sequence number of 3 to PRMAST - the third output file on a volume to contain 5 files:

```
// LBL PRMAST,,,,3
```

Later, when you want to read (input) PRMAST, you can go directly to that file by including the same statement (// LBL PRMAST,,,,3) in your device assignment set. When you specify the file sequence number, data management searches for the first file with that number. If it's found, data management then checks the file identifier for a match. (If the file sequence number you specify is not found or if the file identifiers don't match, a data management error results.)

Remember, you must assign file sequence numbers when a tape file is created in order to reference that file by sequence number later. If you don't assign a sequence number on output, data management assigns a number 1 to the file regardless of its position on the tape volume. If you don't provide a sequence number on input, data management does not check for a sequence number but expects to use the first file encountered. In either case, omitting file sequence numbers means using another method to position the tape to the file you want (e.g., the // MTC statement or reading and closing preceding files without rewinding until the desired file is reached).

Different Versions of a File

Ordinarily, only one generation of a file is used by a program. There are instances, however, when more than one generation of the same file may be needed. For example, one generation contains payroll deductions only used in January, March, and May, and another generation has the payroll deductions used only in February, April, and June. To indicate the different generations of a file, you can use the 1- to 4-digit *generation-number* parameter of the LBL job control statement. This is used only with tape files, and is the sixth parameter shown in the format. By using this parameter, you can be sure the correct generation is used.

Suppose you did have two different generations of the payroll deduction file, with a file identifier of CUSTMAST, and you want to use the second generation. This would be coded as:

```
// LBL CUSTMAST,,,,,2
```

If you omit this parameter, data management assumes 0001.

Let's go one step further. Each generation of a file can have several different versions. Again, we have these two different generations of the CUSTMAST payroll deduction file. Generation 1 is used in January, March, and May, and generation 2 is used in February, April, and June. But, suppose each of these generations had two unique sections. Version 1 is used in odd-numbered years, and version 2 is used in even-numbered years.

We could use the 1- to 2-digit *version-number* parameter to do this.

Suppose it is January, 1990. We need generation 1 (January) and version 2 (1990 is even numbered). This would be coded as:

```
// LBL CUSTMAST,,,,,1,2
```

If the *version-number* parameter is omitted, data management assumes 01.

Changing the Label of a Disk File

The REN statement is used to permanently change the label of a disk file through job control - a simpler procedure than the alternative methods for renaming disk files.

The format of REN is:

```
//[symbol] REN lfdname, {new-label  } [,NTERM]
                        {'new-label'}
```

The *lfdname* parameter identifies the file to be renamed. It must match the *lfdname* in the LFD statement for the file.

The file's new label is specified in the *new-label* parameter. *New-label* replaces the existing label identified in the device assignment set for the file. If new-label contains embedded blanks, it must be enclosed by single quotation marks. It may be from 1 to 44 alphanumeric characters in length.

Specifying optional parameter *NTERM* causes any fatal errors encountered during the renaming process to be ignored, but permits the job to continue. If this parameter is present, the job continues running if a renaming error occurs, but the file is not renamed. If *NTERM* is omitted, the job terminates at the point of error.

The REN statement is checked for syntax errors by the run processor during job stream validation. If no errors are detected, the job is queued and becomes a scheduling candidate. The run processor passes information from the REN statement to the step processor, which performs the actual renaming during job execution.

The device assignment set for the file to be renamed must precede the REN statement. It is a good idea to place the REN statement within the control stream as close to the device assignment set for the file as possible, since // REN is only effective against files on volumes mounted when the REN statement is encountered.

A file is renamed in the job step containing // REN, prior to execution of the program for that step, or prior to job termination if no EXEC statement follows // REN. Subsequent references to the renamed file must use *new-label* in the LBL statement of the device assignment set for the renamed file.

Notes:

1. *The REN statement is used only to rename disk or format-label-diskette files; it may not reference device assignment sets for data-set-label diskette or tape volumes.*

2. *REN statements are not permitted against files on SYSRES that begin with Y, or against files on SYSRUN that begin with YR.*

3. *Don't use / / SKIP to bypass a device assignment set referenced by a REN statement that is not also bypassed. If you do, you'll get an error during the renaming process. (See "Adding Cards to a Stored Control Stream" in Section 6 for more information.)*

4. *If you rename a cataloged file, you must recatalog the file under the new name.*

Suppose you have a program that calculates the engineering department's payroll and outputs a disk file labeled EGRPAY. The control stream to rename the file EGRCOST looks like this:

```
      .
      .
      .
// DVC 50
// VOL DSK01
// LBL EGRPAY
// LFD DSKOUT
      .
      .
      .
// REN DSKOUT,EGRCOST
      .
      .
      .
```

The file's label is now EGRCOST. Suppose that a subsequent job step uses EGRCOST as input for calculating company-wide costs. Building on our first example, the renamed file is referenced subsequently in the control stream like this:

```
      .
      .
      .
// DVC 50
// VOL DSK01
// LBL EGRPAY
// LFD DSKOUT
      .
      .
      .
// REN DSKOUT,EGRCOST
      .
      .
      .
// DVC 50
// VOL DSK01
// LBL EGRCOST
// LFD DSKIN
      .
      .
      .
```

A single REN statement applies only to the first volume in a multivolume file. To rename a multivolume file, therefore, you must specify a unique REN statement for each volume in the file.

If EGRPAY in our first example had been a multivolume file, we would have renamed it this way:

```
      .
      .
      .
// DVC 50
// VOL DSK01
// LBL EGRPAY
// LFD DSKOUT1
      .
      .
      .
// DVC 51
// VOL DSK02
// LBL EGRPAY
// LFD DSKOUT2
      .
      .
      .
// REN DSKOUT1,EGRCOST
// REN DSKOUT2,EGRCOST
      .
      .
      .
```

Use the REN statement carefully to avoid renaming a file concurrently used by another job. To help prevent this problem, establish nonsharable status (using the NS option of the VOL statement) for endangered disk volumes, or use passwords known only to selected personnel.

Specifying Qualifiers for File Identifiers

The QUAL job control statement is used to prefix a qualifier to all subsequent file identifiers in a job. The format of the QUAL statement is:

```
//[symbol] QUAL qualname
```

The *qualname* is a 1- to 8-character alphanumeric name. When specified, this name followed by a slash becomes the qualifier, and is automatically prefixed to each subsequent file identifier in your job control stream.

Consider the following example:

```
// QUAL SMITHCO
// DVC 60 // VOL DISK01
// LBL PAYABLES.TAXES
// LFD PAYFILE
// DVC 60 // VOL DISK01
// LBL INCOME.INTEREST
// LFD INFILE
```

In this example, SMITHCO is specified as the qualifier and will be prefixed, along with a slash, to each subsequent LBL file identifier producing:

SMITHCO/PAYABLES.TAXES and SMITHCO/INCOME.INTEREST.

The qualifier remains in effect until the end of the job or until another QUAL statement is encountered. If the next QUAL statement specifies another *qualname*, that name becomes the qualifier for any subsequent file identifiers. If no name is specified (e.g., // QUAL), use of the qualifier is terminated.

An LBL file identifier that is already prefixed with an alphanumeric name and a slash overrides the QUAL statement qualifier. Consider this example:

```
// QUAL SMITHCO
// DVC 60 // VOL DISK01
// LBL PAYABLE.TAXES
// LFD PAYFILE
// DVC 60 // VOL DISK01
// LBL INCOME/INTEREST
// LFD INFILE
```

INCOME/ in the second LBL statement is already considered a unique qualifier; therefore, SMITHCO/ will be prefixed to PAYABLE.TAXES but not to INCOME/INTEREST.

Because the QUAL statement is especially useful in identifying cataloged files (see "File Cataloging" in Section 6), QUAL is also discussed in the *File Cataloging Technical Overview*, 7004 4615.

More Information about the Logical File

So far, you know the LFD job control statement is used to provide a file name that associates the file defined in the program with the file information in the control stream. Now, by introducing the optional parameters, you will see some of the other functions it provides. Once again, its format is:

```
//[symbol] LFD  {filename }  [, {n}]  [, {EXTEND}]
                {*filename}  [  {8}]  [  {INIT  }]
                                      [  {PREP  }]
                                      [  {ID    }]
                                      [  {IGNORE}]
```

Refer to this format as each parameter is introduced.

We have already discussed the *filename* parameter. An asterisk (*) indicates that the file label is lockable.

Reserving an Extent Information Storage Area

Files are defined on disk and format-label diskette volumes in terms of extents. An *extent* is space on the volume made up of contiguous tracks. If you recall, we used the EXT job control statement to split up a file into two extents. So, in the strict sense, an extent is not always the entire disk area a file requires; at times it is, but at other times it isn't.

Information about the extents is placed in the job's prologue along with other information needed to regulate your job. (See "Minimum and Maximum Main Storage" in Section 2.) On the JOB statement, you specify the minimum and maximum amount of main storage needed to execute your largest job step. However, in order for job control to reserve sufficient main storage for the extent information in the prologue, you must specify the number of physical extents a file has in the second parameter of the LFD statement. Assume, for example, that the file named DSKOUT has 10 extents. To reserve space for information about these extents, you code the LFD statement as follows:

```
// LFD DSKOUT,10
```

The space acquired by using this parameter influences the total main storage requirement for the job. If you specify a value of zero, job control does not reserve main storage for extent information. If you omit this parameter, main storage sufficient for information about eight extents is reserved. The maximum value you can specify for this parameter is 20.

Note: *If you specify a greater number of extents on the LFD statement than a file actually has, main storage is used unnecessarily for the extent information. Although this should be taken into consideration, problems are more likely to arise if you specify fewer extents on the LFD statement than the file actually has.*

Specifications for Existing Files

The third parameter of the LFD job control statement provides five different options: *INIT*, *EXTEND*, *PREP*, *IGNORE*, and *ID*.

The *INIT* option is used to initialize an existing disk file; that is, *INIT* causes all information except the allocated space and file identifier to be discarded when the program using that file opens it. When you specify *INIT* for an output file, the output will start at the beginning of the file. When *INIT* is specified for an input file, an end-of-file will be indicated when your program reads the first record. You can specify *INIT* for all disk and diskette files.

Suppose you already reserved for the old file rather than allocate a new one? In this way, you are not leaving dead space on the disk volume.

We'll assume that the file name is SORTOUT. The device assignment set to reuse WORK2 on disk volume DPS028 would look like this:

```
// DVC 50
// VOL DSP028
// LBL WORK2
// LFD SORTOUT,,INIT
```

Notice the logical unit number for the DVC job control is 50. This indicates any disk device can be used. Also note the absence of an EXT job control statement. It wasn't needed; specifications for the new file are the same as the old one.

Note: *The INIT parameter must not be used for a file that contains checkpoint records. The use of this parameter causes writing to begin at the start of file every time you log a checkpoint record to the file, thus overwriting any checkpoint records already existing on the file.*

The *EXTEND* option allows you to add information to the present end of an existing output file if the instructions in your program allow you to do so. *EXTEND* has no effect on input files.

You can specify *EXTEND* for tape files; *EXTEND* logically does not affect disk and diskette files.

Suppose you allocated four cylinders for your file, but filled only two cylinders with information. Now, you have more information to add to this file, and your program allows you to do so. You must also instruct job control that you intend to do this.

If the file name were ADDON, you could extend the file like this:

```
// LFD ADDON,,EXTEND
```

Remember, whether or not you can actually extend a file depends on the instructions given in your program. In COBOL, for example, an OPEN OUTPUT statement does not permit file extension even if you specify the *EXTEND* parameter on the LFD statement.

The IGNORE option lets you specify that a file is an optional file. This means you can decide at job execution time, without having to change the program, whether accesses to the file should be processed or ignored based on the file resources that are available. This option should be used only with consolidated data management. It is ignored if specified for a file that is accessed using basic data management (including SAT files).

When the IGNORE option is specified, the following status is returned to the executing program:

- File initialization (OPEN) - successful status
- Input operation - end of file (EOF) indication
- All other operations - successful status (request is ignored)

Use this option when a program reads or writes multiple files and it is not necessary that all of the files be accessed. For example, a program updates some files and produces a report, but the report is not always needed. You would specify *IGNORE* in the // LFD statement of the printer device assignment set. Another example is where the program reads multiple input files, but one of them is not available at the time and is not essential to the execution of the program.

When this option is specified for an input file, the program should be performing sequential retrieval; otherwise, it may not be expecting an EOF indication.

Indicating Where the Load Module Is Located

An EXEC job control statement is required to call a load module and initiate execution. Once again, the format is:

```
//[symbol] EXEC program-name [, {[library-name]
                                 {$Y$RUN
                                 {$Y$LOD }] [,[±]switch-priority][,ABNORM=label]
```

The second parameter indicates the name of the library (on disk) containing the load module. This can be either YLOD, YRUN, or the LFD name of the alternate load library you have previously specified in the control stream.

As you can see, the shaded default option is YLOD. This is where you would store most of your programs. When you omit the second parameter, YLOD is searched. If the load module is not found here, then the job's YRUN file is searched.

Another choice you have is the job's YRUN file, which is where the linkage editor stores your load module if you have not indicated a specific library. You would code YRUN in the EXEC job control statement if you have a load module with the same program name in YLOD. Let's assume that you have a load module named PAYROL in YLOD and that you want to make some changes to this program. Take the source deck, make the necessary changes, and compile it with the same program name: PAYROL. When it comes time to execute, the system is told that the load module to fetch is PAYROL. Without specifying the library name on the EXEC job control statement, the default (YLOD) is assumed, so the system fetches the PAYROL load module from YLOD. But, you wanted the one from the job's YRUN file. You are going to receive the wrong load module. So, in this case, you had better indicate that you want to fetch from the job's YRUN file. Remember, the job's YRUN file is only a temporary file. Any load module you store here is available only during that job.

Let's say your load module is named PAYROL and it is loaded in the job's YRUN file. It would be coded as follows:

```
// EXEC PAYROL,$Y$RUN
```

If the load module cannot be found in the job's YRUN file, YLOD is searched to see if it was stored there.

The remaining choice for this parameter, *library-name*, is used when the load module is stored in a private load library of your own. If you do this, you must define this library in a device assignment set, and the *library-name* must agree with the file name on the LFD job control statement. Normally, if the module is not found in this library, YLOD and then YRUN are searched. If, however, you specify NSRCH on the OPTION job control statement, only the library named on the EXEC statement is searched for the load module; YRUN and YLOD are not searched. (See "Selecting Optional Features" in Section 6.)

Let's say the load module is named PAYROL, and it is stored in a library with a file identifier of PAYLIBRARYMAST, on disk volume DISK01. You used PAYLIB as the file name on the LFD job control statement, and, as the file identifier on the LBL job control statement, you would, of course, have to use PAYLIBRARYMAST. The device assignment set and the EXEC job control statement would be coded as:

```
// DVC 50
// VOL DISK01
// LBL PAYLIBRARYMAST
// LFD PAYLIB <----------┐
// EXEC PAYROL,PAYLIB <--┘
```

If this library is not accessed by your program (if it is only accessed by the system to obtain the load module named on the EXEC job control statement), the file name on the LFD job control statement need not agree with any specification within your program. It serves only to associate the library in the device assignment set with the library on the EXEC job control statement. As the file name on our LFD job control statement, we could have used any name as long as it agrees with the specification on the EXEC job control statement.

If the load module is not located, YLOD and then the job's YRUN file is searched.

Task Switching Priority

The EXEC job control statement is also used to specify task switching priorities. This synchronization and rotation of central processor control from task to task is a function of the supervisor, and is described in the *Consolidated Data Management Programming Guide*, UP-9978.

The *switch-priority* is the third parameter of the EXEC job control statement. The priority you specify can be an absolute value ranging from 1 to 60, with the lower number representing the higher priority (1 is the highest priority). Assume, for example, your job has one step and you want a switching priority of 10 assigned to the specified program. (The load module name for the program-name is SWITCH and it is stored by default in YLOD.) You could code the EXEC statement as follows:

```
// JOB MYJOB
      .
      .
      .
// EXEC SWITCH,,10
/&
```

You can also specify a relative value such as +3 or -3 to change priority for a program specified in a particular job step with respect to the job's overall priority (as set, for example, by a SWITCH operator command or an OPTION PRI job control statement).

When specifying priorities this way, remember that a plus (+) value *decrements* the overall assigned value. This results in a *higher* task switching priority. A minus (-) value *increments* the overall assigned value. This results in a *lower* task switching priority.

Suppose you code the following:

```
// JOB MYJOB          ] Assigns an overall task switching priority of 7
// OPTION PRI=7       ] to each program.
      .
      .
      .
// EXEC PROG1         ]
      .                ]
      .                ] The programs specified in these 2 job steps
      .                ] have a task switching priority of 7.
// EXEC PROG2         ]
      .                ]
      .                ]
      .                ]
// EXEC PROG3,,+2     ]
      .                ]
      .                ] The program specified in this job step has a
      .                ] task switching priority of 5.
// EXEC ORIG4,,3      ]
      .                ] The program specified in this job step has a
      .                ] task switching priority of 3.
      .                ]
// EXEC PROG5,,-4     ] The program specified in this job step has a
      .                ] task switching priority of 11.
      .                ]
      .                ]
/&
```

The OPTION PRI job control statement is discussed in "Selecting Optional Features" in Section 6.

If you omit a task switching priority, the lowest available priority (the highest number) is used.

You should understand that the task switching priority specified on the EXEC job control statement is only the initial switching priority for that job step. There are two ways it can be changed during the job step:

- The operator can raise or lower the priority using the SWITCH console command if a job is getting too much or too little CPU time.
- The program itself may raise or lower its priority using the CHAP (change priority) macroinstruction. This function is described in the *Consolidated Data Management Programming Guide*, UP-9978.

As you can see, the effect of the switching priority really depends on the task switching priorities specified for other jobs running at the same time as your job. Your job will not gain any advantage by specifying a task switching at priority of 1 (the highest priority) if all other jobs also use priority 1. There is a case, however, where the assigned switching priority is particularly significant. Recall from Section 1 that the RUN symbiont is one portion of job control that reads and analyzes job control streams. The RUN symbiont is only one of many OS/3 symbionts that perform system functions, usually in response to operator console commands. Normally, all symbionts run at priority 0, i.e., higher than any user job. A SYSGEN option, however, allows

the supervisor to be configured so that all symbionts run at some lower (user) priority. For example, suppose symbionts run at priority 2 under your supervisor. The only jobs that should be run with task switching priority 1 would be those that are extremely time-critical and cannot tolerate the loss of CPU time whenever a symbiont is active. Other jobs should be run at priority 3 and lower.

Avoiding Abnormal Termination due to Program Errors

The *ABNORM=label* keyword parameter of the EXEC job control statement is used to bypass job control statements if your program contains errors that may cause the job to abnormally terminate. If the program has such errors, control of the job skips to the statement whose label you specify in this parameter so that the job's execution can continue. Any subsequent action depends on the contents of the target job control statement. A more specific example for using this parameter is given in "Bypassing Job Control Statements to Avoid Abnormal Termination" in Section 6. For now, just remember that *ABNORM=label* is a keyword parameter, not a positional parameter, and, therefore, may be coded in any position. For example:

```
// EXEC MYPROG,ABNORM=ERROR
```

Also remember that the operator can still cancel (normally terminate) your job even though you specify this parameter.

The Job Control Language So Far

We have now covered the job control statements you'll probably use most frequently for your jobs. Section 5 deals with system JPROCS provided in the basic OS/3 software package. Their use eliminates the need of repeatedly coding a series of job control statements that perform a specific function.

Section 5
Doing It the Easy Way - with Procedure Calls

What Is a Procedure?

Have you ever heard someone say: "I've made that mistake before. There must be some way, some *procedure, to make sure it won't happen again"?* Common errors are: keystroke errors, forgotten commas, statements out of sequence, etc. - errors that occur because of repetition rather than unfamiliarity. If we could reduce the number of job control statements coded, the bulk of these errors would also be reduced. What is needed is a *procedure* that allows you to write a series of job control statements, store them for later use, and, by writing a single job control statement, call in these job control statements whenever needed.

This *procedure* exists - it allows you to write and call your own procedures, or to call procedures supplied by Unisys. In Sections 6 through 9, you'll learn how to write, store, and call your own procedures. This section discusses how to use procedures supplied by Unisys. These procedures are called by job control procedure call statements (JPROC calls) in the control stream. Each JPROC call generates a ready-to-use set of job control statements. Optional parameters in the JPROC call line enable you to tailor the job control statements generated to suit your needs. (See "Job Control Procedures" in Appendix C for a complete listing of JPROCS.)

When you use more than one JPROC call, keep this in mind: only one JPROC call can appear on a single card. JPROC calls can be part of a multistatement line of coding, but

- It must be the only JPROC in the line.
- It must be the last statement on the line.

You can code this:

```
// job control statement      // jproc call
```

but not this:

```
// jproc call      // jproc call
```

and not this:

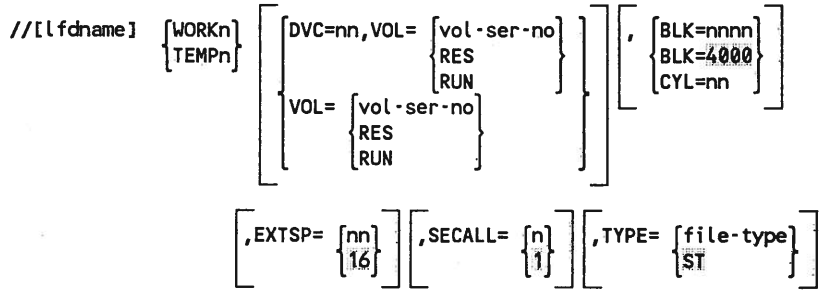
```
// jproc call      // job control statement
```

Setting Up Temporary Work Files

Temporary work files are used extensively by programmers to store intermediate processing results and data that will only be used in a particular job or job step. Depending on file characteristics and the device used, from three to five job control statements are needed in the device assignment set for each temporary work file. The WORK and TEMP JPROC calls allow you to generate any device assignment set needed for temporary work files.

The difference between the two JPROCS is that WORK sets up temporary files for one job step and deletes them at the end of the job step. TEMP sets up temporary files for the duration of the job, deleting them at the end of the job. WORK and TEMP also generate different default file name values - we'll explain these in a moment.

The format for WORK and TEMP is:



Suppose your assignment is to write a program that reports the grades for each student in the local school district. The program must list each student's name, grouped by school, in descending grade order. The disk area that stores the data used to calculate the order will never be used again once the job step terminates - an ideal candidate for a temporary file created by WORK.

Ignoring all optional parameters, the basic WORK JPROC call is:

```
// WORKn
```

Where *n* is a number in the range 1 through 10. Up to 10 temporary work files can be set up for each job step (or job, if you're using TEMP). If no specific device or volume is requested, the file is allocated on either SYSRES or the job's YRUN file; odd-numbered files go to SYSRES and even-numbered files to YRUN. So, if you want one temporary file allocated on the job's YRUN file, for the duration of the job step, you would code the following:

```
// WORK2
```

These job control statements would be generated:

```
// DVC RUN
// EXT ST,,1,BLK,(256,4000)
// LBL $SCR2,16
// LFD $SCR2
```

We'll discuss the generated EXT statement in conjunction with the *BLK*, *EXTSP*, and *TYPE* parameters. For now, it's sufficient to know that 4000 blocks, each 256 bytes long, are allocated by default.

The *lfdname* parameter of WORK and TEMP supplies a file name for the generated job control statements. It is one to eight alphanumeric characters in length. The file identifier on the LBL statement generated by WORK is always prefixed by $SCR, which identifies job step temporary (scratch) files. The number after $SCR corresponds to *n* in WORKn. If you omit the *lfdname* parameter of the WORK JPROC call and code the following:

```
// WORK1
```

the generated statements are:

```
   .
   .
   .
// LBL $SCR1
// LFD $SCR1
   .
   .
   .
```

The file name in your program must also start with $SCR. In addition, you must use the same WORK JPROC call each time the program is run. If the JPROC call is changed to // WORK7, for example, the file name in your program must be changed to $SCR7.

For TEMP, unlike WORK, the generated file identifier is $JOB if you omit the *lfdname* parameter. Therefore, if the file name in your program begins with $SCR, you must use the *lfdname* parameter of the TEMP JPROC call, like this:

```
//$SCR1 TEMP1
```

to generate:

```
   .
   .
   .
// LBL $JOB1
// LFD $SCR1
   .
   .
   .
```

If you had not used the *lfdname* parameter in this example, the generated file name would have been $JOB1, which would not have matched the file name in your program.

You can have the control statements generated by WORK and TEMP listed by specifying the P option on the JOB statement. If you have spooling in your system, the control statements will be printed in the job log. Otherwise, they will be displayed on the system console.

When the job step terminates, all temporary files created by WORK are scratched. Files created by TEMP are scratched at the end of the job.

The *lfdname* parameter can also indicate a file's function when using the WORK JPROC call. For example, if you code

```
//GRADEOUT WORK2
```

the generated job control statements are:

```
// DVC RUN
// EXT ST,,1,BLK,(256,4000)
// LBL $SCR2
// LFD GRADEOUT
```

It is easier to remember what GRADEOUT contains than it might be to remember what $SCR2 contains.

The remaining optional parameters of the WORK and TEMP JPROC calls are keyword parameters. If you are unsure of the rules for coding them, turn to Appendix A to refresh your memory.

Notes:

1. *Work file labels ($JOB and $SCR) are modified to be unique for the job by inserting a job-id after the first 4 bytes of the label. This enables all residual work files (for example, work files not scratched when an HPR occurs) to be cleared from the disk during IPL.*

2. *Within user JCL, work file labels (those starting with $JOB or $SCR) cannot exceed 39 characters.*

Using Your Own Volume

By default, temporary work files are allocated on SYSRES or YRUN. But what if you needed several work files and there isn't enough available space on these volumes? In this case, you would use your own volume by specifying the *VOL* parameter. Building on our last example, if your own volume is DISK01, you would code:

```
//GRADEOUT WORK2 VOL=DISK01
```

This device assignment set is generated:

```
// DVC 50
// VOL DISK01
// EXT ST,,1,BLK,(256,4000)
// LBL $SCR2
// LFD GRADEOUT
```

Note that the logical unit number generated for the DVC job control statement is 50. The WORK and TEMP JPROCS automatically assign the first available logical unit number in the range 50 through 59, but you can use the *DVC* parameter to assign another logical unit number selected from Table A-1 in the *Job Control Programming Reference Manual*, UP-9984. In order to avoid a conflict, for example, you may want to assign a different logical unit number to the temporary work file if you've already assigned DVC 50 to a disk volume in your job control stream.

Suppose we select logical unit number 60 (indicating any 8419 disk) and add the *DVC* parameter to our example, like this:

```
//GRADEOUT WORK2 VOL=DISK01,DVC=60
```

Since *DVC* and *VOL* are keyword parameters, they do not have to be in any specific order. So, it could be coded:

```
//GRADEOUT WORK2 DVC=60,VOL=DISK01
```

Either of these two JPROC calls generate these job control statements:

```
// DVC 60
// VOL DISK01
// EXT ST,,1,BLK,(256,4000)
// LBL $SCR2
// LFD GRADEOUT
```

You can use the *VOL* parameter and omit the *DVC* parameter - job control will assign a logical unit number. The converse is not true; if you use the *DVC* parameter, you must use the *VOL* parameter.

Providing the Extent Specifications

When the WORK or TEMP JPROC calls allocate temporary work files, they are, by default, 4000 blocks - each 256 bytes long. However, you can change this by using the *BLK* parameter or the CYL parameter.

Possibly, your file doesn't require 4000 blocks. Maybe you only need 1000 blocks. Don't tie up 3000 blocks that your program isn't going to use. Use the *BLK* keyword parameter to indicate that only 1000 blocks are needed:

```
//GRADEOUT WORK2 DVC=60,VOL=DISK01,BLK=1000
```

which would generate these job control statements:

```
// DVC 60
// VOL DISK01
// EXT ST,,1,BLK,(256,1000)
// LBL $SCR2
// LFD GRADEOUT
```

Suppose you want to allocate 3 cylinders for the file instead of 1000 blocks. In this case, specify the *CYL* parameter in the JPROC as follows:

```
//GRADEOUT WORK2 DVC=60,VOL=DISK01,CYL=3
```

This JPROC generates the following job control statements:

```
// DVC 60
// VOL DISK01
// EXT ST,,1,CYL,3
// LBL $SCR2
// LFD GRADEOUT
```

In "Reserving an Extent Information Storage Area" in Section 4, we used the second parameter of the LFD job control statement to tell the system how many extents existed in the file. Job control used this to calculate the amount of main storage needed to contain the information about the extents. For the WORK and TEMP JPROC calls, you do this with the *EXTSP* keyword parameter.

When the number of extents is omitted, 16 is assumed. If you know your data will take less than 16 extents, it's a good practice to specify the *EXTSP* parameter. For example, your data may only need one extent; it is foolish to let the system allocate 16.

Assuming only one extent, we would code:

```
//GRADEOUT WORK2 DVC=60,VOL=DISK01,BLK=1000,EXTSP=1
```

These statements would be generated:

```
// DVC 60
// VOL DISK01
// EXT ST,,1,BLK,(256,1000)
// LBL $SCR2
// LFD GRADEOUT,1
```

In "Your Disk or Format-Label Diskette File Needs More Space" in Section 4, we discussed the dynamic extension of a disk file. You can indicate how much additional area to allocate on the WORK and TEMP JPROC calls, too. Use the *SECALL* keyword parameter.

In the grading report, we estimated 1000 blocks were needed for 5000 students. If this amount is exceeded, you will, by default, receive one additional cylinder. The dynamic extension process takes a little time, which increases processing time. Normally, one additional cylinder is enough extra space to contain any additional information, but, at different times in the school year, you are called upon to do the grading report for a neighboring school district. This district has 15,000 students. This will no doubt exceed the 1000 blocks, and the overflow of data will take up more than one cylinder; it will be closer to five cylinders. Job control will keep on dynamically extending the file, in increments of one cylinder, until the needed space is acquired. Since each dynamic extension takes time, why not request that the extension be made all at once, by increasing the dynamic extension amount? This additional space is only allocated when needed (and most times you run this job, dynamic extension will not be needed). The relative cost of extra temporary space acquired infrequently, by dynamic extension, is minimal compared with the processing time cost required to allocate one cylinder five times. Since you know when the special runs for the other school district will occur, they can be scheduled when the use of these five additional cylinders will not hinder jobs being run.

Let's add a 5-cylinder dynamic extension to the example we've been using:

```
//GRADEOUT WORK2 DVC=60,VOL=DISK01,BLK=1000,EXTSP=11,SECALL=5
```

This generates these job control statements:

```
// DVC 60
// VOL DISK01
// EXT ST,,5,BLK,(256,1000)
// LBL $SCR2
// LFD GRADEOUT,1
```

You should now be able to use the WORK and TEMP JPROC calls and tailor them to your own needs.

By default, both the WORK and TEMP JPROCS set up temporary SAT files, but you can also specify MIRAM files *(MI)* using the *TYPE* parameter:

For example, we can include the *TYPE* parameter in the previous example to indicate a MIRAM file type. Code the JPROC as follows:

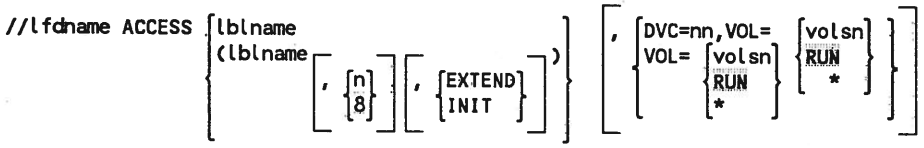
```
// GRADEOUT WORK2 DVC=60,VOL=DISK01,BLK=2000,EXTSP=11,SECALL=5,TYPE=MI
```

This generates the following job control statements:

```
// DVC 60
// VOL DISK01
// EXT MI,,5,BLK,(256,1000)
// LBL $SCR2
// LFD GRADEOUT,1
```

Accessing Previously Allocated Files

Ordinarily, to access a previously allocated disk file, you use the DVC, VOL, LBL, and LFD job control statements. These statements aren't needed, however, if you use the ACCESS JPROC call. Its format is:



This JPROC call can be used to access any tape or previously allocated disk file, except a multivolume file. For instance, to access multivolume files, a file serial number must be specified (otherwise, data management returns an error indication). There is no parameter in the ACCESS JPROC call for this specification.

The ACCESS JPROC call uses both positional and keyword parameters; if you're a little hazy on the rules for coding, consult Appendix A.

Let's digress a moment, and discuss the *DVC* and *VOL* parameters. The rules governing their use are exactly the same as for the WORK and TEMP JPROC calls. See "Using Your Own Volume" in Section 5. If you omit the *VOL* parameter, the file is assumed to be on the volume containing the job's YRUN file.

Let's set up a situation where the ACCESS JPROC call can be used to advantage. Suppose we want to write an inventory control program for a metal fabricating plant. This plant produces many different items: office furniture, aircraft parts, aluminum siding, and such. Each item produced depletes a central metal inventory, and the purchasing agent wants to know when to order new stocks of metals. After making some further assumptions (DVC=60 and VOL=DKWORK) we have the information needed to code a useful ACCESS JPROC call:

```
//MMIFIL ACCESS METALMASTINV,DVC=60,VOL=DKWORK
```

This ACCESS JPROC call generates this device assignment set:

```
// DVC 60
// VOL DKWORK
// LBL METALMASTINV
// LFD MMIFIL
```

The ACCESS JPROC call has two optional positional parameters that allow you to generate a complete LFD job control statement. In "Specifications for Existing Files" in Section 4, we discussed how the optional parameters of the LFD job control statement are used. Well, the optional positional parameters of the ACCESS JPROC call correspond exactly to the parameters of the LFD job control statement.

Compare these formats:

```
                                 +----------------------------+
//[symbol] LFD  {filename }      | [ ,{n} ] [ ,{EXTEND} ]     |
                {*filename}      |    {8}       {INIT  }      |
                                 +----------------------------+

                                 +----------------------------+
//lfdname ACCESS {lblname        | [ ,{n} ] [ ,{EXTEND} ]     | ) } [ ,{DVC=nn,VOL=     {volsn} } ]
                 {(lblname       |    {8}       {INIT  }      |     [  {                {RUN  } } ]
                 {               +----------------------------+     [  {                {*    } } ]
                                                                    [  {VOL= {volsn}          } ]
                                                                    [  {     {RUN  }          } ]
                                                                    [  {     {*    }          } ]
```

The two enclosed portions are identical, both in format and function.

The *n* parameter specifies the number of extents reserved in main storage, and the default value is 8.

The other optional positional parameter provides two different options: *EXTEND* and *INIT*.

As a brief recap of "Specifications for Existing Files" in Section 4, we can say that using the *EXTEND* option allows you to add information to the present end of the file. With the *INIT* option, you can write over the existing information in the file (except for the file identifier).

When you code any of these options, or specify the number of extents in the file, with the *lblname* parameter, you have to enclose them all within parentheses.

Since the metal fabricating plant buys and sells a lot of materials, the metal master inventory file changes a great deal. You must update the metal master inventory file to reflect any new materials purchased and sold, and perform the main processing function.

All new material is purchased on the tenth of the month. On the eleventh, it's time to add the new material to the metal master inventory file. Our call line would look like this:

```
//MMIFIL ACCESS METALMASTINV,DVC=60,VOL=DKWORK
```

By default, space is reserved for eight extents. The following device assignment set is generated:

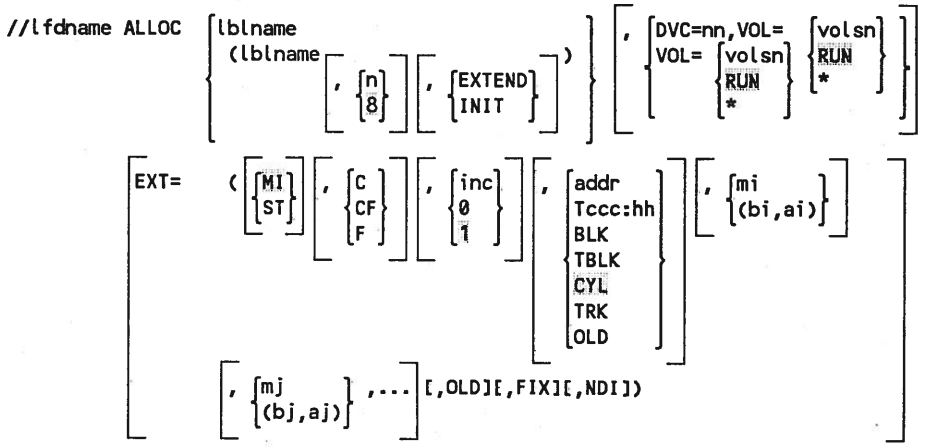
```
// DVC 60
// VOL DKWORK
// LBL METALMASTINV
// LFD MMIFIL
```

While there are more minor limitations to the ACCESS JPROC call, there are many instances where it's very useful.

Allocating a File with a JPROC Call

You saw how we used the ACCESS JPROC call to access an existing disk file. This replaced four job control statements, helping to reduce the possibility of coding errors. You save more coding time by using the ALLOC JPROC call to allocate disk and diskette files. It's a combination of the ACCESS JPROC call and the EXT job control statement.

The format is:



The *EXT* keyword parameter provides all the options available as positional parameters on the EXT job control statement. The only difference is the equal sign and the parentheses.

Note: *Section 4 describes the parameters and options available for data-set-label and format-label diskette using the EXT statement.*

Your site processes payrolls for 25 different companies. Each company has a file containing each employee's name and hourly wage. This file is accessed during the processing of the company payroll (a use for the ACCESS JPROC call). Originally, though, each company file was on punched cards, and each of them must be loaded into its own disk area. (Here is one use for the data utility card-to-disk routines; why write your own program when one is already provided?) To do this, there must be a device assignment set for each file being created. This means 25 device assignment sets for the 25 files. Looking back at "Basic Job Control Statements" in Section 3, we see that the site manager needed five job control statements to allocate his disk file: DVC, VOL, EXT, LBL, and LFD. This means 125 job control statements would be needed. The ALLOC JPROC reduces this to 25.

For our example, assume that the file requirements (such as access method area needed, etc.) are identical for each of the 25 files, so most of the parameters for the job control statements (and the ALLOC JPROC call) would be the same. Of course, each file must have its own unique file identifier, but the information about the extents is the same, all the files can be stored on the same disk volume, and, since you're using the same program to store them (the data utility routine, run 25 times), the file name is the same.

We'll assume that disk volume DSP028 will hold these files. It's the only volume with DSP028 as the volume serial number, so a logical unit number in the range of 50 through 59 (any disk device) suffices. If we omit the *DVC* keyword parameter, job control assigns the first available number in this range. Assume that the first one available is 50. The data utility card-to-disk routine uses OUTPUT1 as the file name in the LFD job control statement; this is the value we must use as the *lblname* parameter. All the file names for the different data utility routines can be found in the *Data Utilities Operating Guide*, 7004 4516.

We are going to take the default value for the number of extents (8), and we don't want to use any of the options for a previously allocated file.

After defining the extent information, we'll have the parameters that are common to all files. The only thing left will be to supply a unique file identifier for each file. All the files are MIRAM files (which is a default condition, MI), allocation is contiguous, with one cylinder for dynamic allocation. The initial allocation is two cylinders. Now we have what we can call our master ALLOC JPROC call for the 25 different files. The only thing missing is the file identifier.

From the information we've gathered, our master ALLOC JPROC call for the file identifier would look like this:

```
//OUTPUT1 ALLOC xx...xx,VOL=DSP028,EXT=(,C,1,CYL,2)
```

Now, we need file identifiers for each file. Each of the 25 files must be given a unique file identifier so the proper file can be accessed at the proper time. The names of two of the companies are Target Manufacturing, Incorporated, and the Reality's Dress Company. Why not use TARGET and REALITYS as the file identifiers? It makes them easier to remember and identify. The ALLOC JPROC call for Target Manufacturing, Inc., would be:

```
//OUTPUT ALLOC TARGET,VOL=DSP028,EXT=(,C,1,CYL,2)
```

and the generated job control statements would be:

```
// DVC 50
// VOL DSP028
// EXT ,C,1,CYL,2
// LBL TARGET
// LFD OUTPUT
```

The ALLOC JPROC call for the Reality's Dress Company would be:

```
//OUTPUT1 ALLOC REALITYS,VOL=DSP028,EXT=(,C,1,CYL,2)
```

The only difference in the generated job control statements is the file identifier of the LBL job control statement: one is TARGET; the other is REALITYS.

Note: *If the EXT keyword parameter is omitted, job control assumes one cylinder for dynamic extension and, therefore, allocates one cylinder of extent space for a MIRAM file.*

Now, let's see how to avoid the error of assigning one volume to different devices.

Too Many Devices for the Same Volume

Many applications use two files on the same volume. A common mistake is to assign the files - thus the volume - to two different devices during the job. Using the DVCVOL JPROC helps to avoid this. This JPROC assigns logical unit numbers for the generated DVC job control statements. It also generates a VOL job control statement with the volume serial number you specify in the JPROC call. The format is:

```
//[symbol] DVCVOL {vol-ser-no} [,lun] [,NOVOL= {Y}]
                {RES       }             {N}
                {RUN       }
```

The *symbol* in the label field is only used as a target for the job control statement that causes a branch.

The DVCVOL JPROC assigns the logical unit numbers 50 through 59, in ascending sequence, to the different volume sequence numbers in the order they are encountered in the control stream. If you had three volumes, A, B, and C, in that sequence, A would be 50, B would be 51, and C would be 52. It is possible, however, to override these volumes and assign a specific logical unit number to a specific volume by using the *lun* parameter.

The *NOVOL* parameter (NOVOL=Y) performs the same function as the *NOV* parameter of the VOL job control statement. It suppresses the checking of volume serial numbers.

Once a logical unit number is assigned by the DVCVOL JPROC call to a volume, the same logical unit number is assigned whenever this volume is encountered in the job. If volume A was assigned 50 in one job step, and you tried to assign it to 51 in the next job step, the system overrides the 51 and assigns 50.

If you tried to do this by using just the DVC and VOL job control statements, assigning 50 in the first job step and 51 in the next job step, your job may run, but you may have to demount the volume from DVC 50 and mount it on DVC 51.

When you use the DVCVOL JPROC call, the LBL and LFD job control statements for the file must be present in the control stream after the DVCVOL JPROC call. If you're allocating a file on a disk volume, the EXT job control statement must, of course, also be used.

There is a limit to the number of volumes you can assign using the DVCVOL JPROC call in a job: 10.

Another point worth remembering: the DVCVOL JPROC call can be a member of a multistatement line of coding, but it must be the last statement on the line.

Let's set up a control stream with some DVCVOL JPROC calls, and see what job control statements are generated. The numbers refer to the explanation following the example.

```
    // DVCVOL DISK01
    // LBL A
1.  // LFD A  // DVCVOL DSK002
    // LBL B  // LFD B
2.  // DVCVOL DK0003,69
    // LBL C  // LFD C
3.  // DVCVOL DISK01
    // LBL X
    // LFD X
4.  // DVCVOL DK0003,67
    // LBL Y
    // LFD Y
```

1. This is an example of a multistatement line of coding. Note that the DVCVOL JPROC call is the last statement on the line. The next line and the line after example 2 are also multistatement lines.

2. This line assigns a specific logical unit number, 69, to the volume DK0003.

3. This DVCVOL JPROC call is used again for the volume DISK01. It was also used in the first DVCVOL JPROC call on the first line. It will be assigned the same logical unit number assigned to the first call for the volume DISK01. You'll see this more clearly when we show the job control statements generated by these DVCVOL JPROC calls.

4. This is another example calling for the volume DK0003, which was already assigned a logical unit by a DVCVOL JPROC call. Notice that it also requests a specific logical unit number: 67. Since this volume already was assigned to logical unit number 69 in example 2, the request for logical unit number 67 is ignored, and it is assigned to logical unit number 69.

Here are the generated job control statements. They should give you a clearer picture of how each DVCVOL JPROC call functioned.

```
1.  // DVC 50
    // VOL DISK01
    // LBL A
    // LFD A
2.  // DVC 51
    // VOL DSK002
    // LBL B
    // LFD B
3.  // DVC 69
    // VOL DK0003
    // LBL C
    // LFD C
4.  // DVC 50
    // VOL DISK01
    // LBL X
    // LFD X
5.  // DVC 69
    // VOL DK0003
    // LBL Y
    // LFD Y
```

1. Volume DISK01 was the volume encountered in the first DVCVOL JPROC call - it's assigned to logical unit number 50. The LBL and LFD job control statements are not generated by the JPROC call. Remember, these were supplied in the control stream. If another DVCVOL JPROC call for volume DISK01 is encountered in this job, it is automatically assigned to logical unit number 50.

2. A DVCVOL JPROC call for volume DSK002 was the next one encountered. It's assigned the next available logical unit number. Since 50 was already assigned to volume DISK01, 51 is the next available logical unit number.

3. The next DVCVOL JPROC call was for volume DK0003. Normally, it would be assigned to logical unit number 52, which was the next one available. But, the DVCVOL JPROC call for this volume requested a specific logical unit number, 69, so that's what is assigned.

4. Another DVCVOL JPROC call for volume DISK01 was encountered. Since this volume was already requested and assigned earlier in the control stream, this occurrence is assigned the same logical unit number: 50.

5. The volume DK0003 was requested by another DVCVOL JPROC call. Even though a specific logical unit number, 67, was requested, it was assigned to logical unit number 69, since this is the logical unit number assigned earlier in the job. The first number encountered is used, and any other logical unit numbers requested for the volume in the same job are ignored.

To assign multiple diskette volumes through a JPROC call, use the DVCDKT JPROC. It functions the same as the DVCVOL JPROC except that it assigns the logical unit numbers 130 through 132. Its format is:

```
//[symbol] DVCDKT vol-ser-no[,lun][,NOVOL={Y}]
                                       {N}
```

There is also a JPROC call for tape units: DVCVTP. Except for a few minor differences, it functions the same as the DVCVOL and DVCDKT JPROCS. Its format is:

```
//[symbol] DVCVTP vol-ser-no[,lun][,PREP={Y}][,NOVOL={Y}]
                                       {N}        {N}
```

The DVCVTP JPROC call assigns the logical unit numbers 90 through 99. Additionally, DVCVTP has the keyword parameter *PREP=Y*. If specified, this parameter functions the same as the *PREP* option of the VOL job control statement ("Ignoring or Changing the Volume Serial Number" in Section 4); it causes any information currently on the tape volume to be effectively erased.

Using the Linkage Editor

So far, we've discussed how to execute programs stored in a library. These programs were not always located in this library. At one time they could have been on punched cards in one of the programming languages, such as COBOL or RPG II.

These programs are compiled or assembled using a language translator, which converts the program instructions into a form understandable to the computer (an object module). Each language translator has a JPROC call you can use to generate the job control statement needed to direct the operation of the language translator; in other words, you get an object module from source input. The JPROC call for each language translator can be found in the appropriate assembler, COBOL, FORTRAN, and RPG II publications.

In this guide, we'll explain the JPROC call for the linkage editor. But, before we do, a word or two about the linkage editor.

The linkage editor converts an object module into an executable load module. Only load modules can be executed, and the only method of converting object modules to load modules is by using the linkage editor. The function of the linkage editor is fully covered in the *System Service Programs (SSP) Operating Guide*, UP-8841.

The format of the linkage editor JPROC call is:

```
//[symbol]  {LINK }  [input-module-name-1,...,input-module-name-10]
            {LINKG}
        [,PRNTR= {lun[,dest]}]
                 {N[,dest]  }
                 {20[,dest] }
        [,IN= {(vol-ser-no,label)}]  [,OUT= {(vol-ser-no,label)}]
              {(RES)              }          {(RES,label)      }
              {(RES,label)        }          {(RUN,label)      }
              {(RUN,label)        }          {(*,label)        }
              {(*,label)          }          {(N)              }
              {(RUN,$Y$RUN)       }          {(RUN,$Y$RUN)     }
        [,RLIB= {(vol-ser-no,label)}]  [,ALIB= {(vol-ser-no,label)}]
                {(RES,label)       }           {(RES,label)      }
                {(RUN,label)       }           {(RUN,label)      }
                {(*,label)         }           {(*,label)        }
        [,SCR1= {vol-ser-no}]  [,STD= {YES}]  [,ALTLOD= {(vol-ser-no,label)}]
                {RES       }          {NO }             {(RES,label)      }
                                                        {(RUN,label)      }
                                                        {(*,label)        }
                                                        {(RUN,$Y$RUN)     }
[,OPT='options'][,CLIB= {(vol-ser-no,label,modname)}]
                        {(RES,label,modname)       }
                        {(RUN,label,modname)       }
                        {(*,label,modname)         }
[,CMT='comment'][,ENTER=expression]
```

There are two choices in the *operation* field: LINK and LINKG. By specifying LINK, you execute the linkage editor. By specifying LINKG, you execute the linkage editor and the load module you just created (without using an EXEC job control statement).

As you can see, all the parameters are optional. But this JPROC call has default values, which generate the job and linkage editor control statements sufficient to accomplish a link edit, and assumes the following:

- All the object code you specifically want included in the load module is in the job's YRUN file.
- Any object code that may have to be automatically included in the load module (such as error processing routines) is in YOBJ.
- The load module produced is given the name LNKLOD, and it's stored in the job's YRUN file.

You can alter these default conditions using the optional parameters. There are also parameters that allow you to choose special options (such as a specific printer, a certain scratch work file, etc.).

Let's see what job and linkage editor control statements are generated when you omit all the parameters. We'll use both the LINK and the LINKG operations. For these examples, assume that the program was just compiled (or assembled) by a language translator, and the object code was placed in the job's YRUN file. This occurred in the last job step, but it is still the same job. The job's YRUN file is only a temporary file, lasting for the length of the job. So, if you placed the object code in the job's YRUN file in one job and tried to locate it in another job, you wouldn't find it.

```
/*  (this is the end of the language translator job step)
// LINK
/&
// FIN
```

Here's what job control statements are generated:

```
     /*  (this is the end of the language translator job step)
1.   // DVC 20      // LFD PRNTR
     // DVC RES
     // EXT ST,C,1,BLK,(256,10)
     // LBL $SCR1   // LFD $SCR1
3.   // EXEC LNKEDT
     /$
4.        LOADM LNKLOD
     /*
5.   /&
     // FIN
```

1. This is the device assignment set that's generated for the printer. Notice that we've used multistatement coding, showing the DVC and LFD job control statements on the same line.

2. The linkage editor always uses one scratch work area. The JPROC call assigns it to the SYSRES device, and makes it a job step temporary file. (The file identifier begins with $.) This work area is scratched at the end of the job step.

3. This calls the linkage editor from YLOD and initiates its execution.

4. The generated load module must be assigned a name. The default is LNKLOD. This is on the LOADM linkage editor control statement, which is treated as data by job control, thus the /$ and /* job control statements.

5. As always, this indicates the end of the job.

This example is fine if you don't want to execute the program, since default conditions assign the load module to the job's YRUN file, which is only a temporary file. This load module is not available to another job (but it is to another job step in the job). This application is useful if you only want to see the output of the linkage editor; but it isn't much help if you want to execute. This does not mean that you can never access a load module in a job other than the one in which it was link edited. You can, but you have to assign it to a library other than the job's YRUN file. You'll see how later on, when we discuss the optional parameters. But first, let's see how to execute the load module that was placed, by default, in the job's YRUN file.

There are two ways you can execute a load module placed in the job's YRUN file: first, you can execute it in a subsequent *job step* after link editing, using the default LNKLOD load module name on the EXEC job control statement; or, second, you can use the LINKG operation, which automatically executes the load module.

Here's method 1 (LINK):

```
/* (end of language translator job step)
// LINK
// EXEC LNKLOD,$Y$RUN
/&
// FIN
```

The job control statements generated are:

```
/* (end of language translator job step)
// DVC 20          // LFD PRNTR
// DVC RES
// EXT ST,C,1,BLK,(256,10)
// LBL $SCR1       // LFD $SCR1
// EXEC LNKEDT
/$
    LOADM LNKLOD
/*          |
// EXEC LNKLOD,$Y$RUN
/&
// FIN
```

The load module name on the LOADM linkage editor control statement and the program name on the EXEC job control statement is the same: LNKLOD. Since we know the linkage editor always assigns this as the default load module name, we use it as the program name. Also note that *YRUN* is the second parameter on the EXEC job control statement. Remember, in "Specifying Qualifiers for File Identifiers" in Section 4, we said this parameter indicates the name of the library containing the load module. If omitted, YLOD is searched and then the job's YRUN file. Since the job's YRUN file is searched eventually, why specify it? Time. We know, it's in the job's YRUN file, so why search YLOD needlessly? Go directly to the job's YRUN file.

Now, here's method 2 (LINKG):

```
/* (end of language translator job step)
// LINKG
/&
// FIN
```

And here are the generated job control statements:

```
/* (end of language translator job step)
// OPTION GO
// DVC 20          // LFD PRNTR
// DVC RES
// EXT ST,C,1,BLK,(256,10)
// LBL $SCR1       // LFD $SCR1
// EXEC LNKEDT
/$
    LOADM LNKLOD
/*
/&
// FIN
```

The only difference between this LINKG operation and the LINK operation is the generated OPTION job control statement. The *GO* means the load module should be automatically executed when linkage editing is completed. You don't need an EXEC job control statement.

The LINKG operation generates a load module name of LNKLOD and is loaded, by default, in the job's YRUN file. This means it is not available after the job is completed. The LINKG operation is useful when you're testing programs or running programs that are infrequently used.

So far, we've covered only the basic use of the linkage editor JPROC call. Now, let's add some optional parameters and make it do exactly what we want.

Generating LOADM and INCLUDE Linkage Editor Control Statements

Up until now, the module name for the generated LOADM linkage editor control statement has been LNKLOD (the default name). You can override this using the *label* field of the JPROC call, shown as *symbol* in this portion of the format:

```
//[symbol]  {LINK }  [input-module-name-1,...,input-module-name-10]
            {LINKG}
```

The *symbol* parameter is a 1- to 6-alphanumeric-character name. If fewer than six characters are specified, it's padded on the right with zeros. If it's omitted, the value for the first *input-module-name* specified is used for the load module name. If the *input-module-name* parameter is also omitted, LNKLOD is used.

Since we mentioned *input-module-name*, now is a good time to explain it. This parameter allows you to specify up to 10 object modules to be included in the load module you're constructing. In other words, it specifies the module names for the INCLUDE linkage editor control statements. Each *input-module-name* can be from one to eight alphanumeric characters long. If this parameter is omitted, the value specified as *symbol* is also omitted, all object modules residing in the job's YRUN file are included in the load module. An explanation of how the linkage editor JPROC searches for input modules to be included in the load module is given in the description of the IN parameter (see "Making the Linkage Editor Suit Your Needs" in Section 5).

If you are specifying more than one object module name, you may want to specify a value in the symbol field that is representative of all the *input-module-names* to be included. Also, if all eight positions are used for the first *input-module-name* and it is also to be used as the *symbol* by default, the last two positions are truncated by the linkage editor to obtain a 6-character *symbol*, and the linkage editor diagnostic message K016 is issued.

Let's look at examples showing different ways of assigning module names for the generated LOADM and INCLUDE linkage editor control statements.

Here's the first example:

```
/* (end of language translator job step)
//PROG LINK
// EXEC PROG,$Y$RUN
/&
// FIN
```

Here's what is generated:

```
/* (end of language translator job step)
// DVC 20          // LFD PRNTR
// DVC RES
// EXT ST,C,1,BLK,(256,10)
// LBL $SCR1      // LFD $SCR1
// EXEC LNKEDT
/$
    LOADM PROG
    INCLUDE PROG
 /*
 // EXEC PROG $Y$RUN
/$
// FIN
```

By using PROG as the *symbol*, you get PROG as the module name on the LOADM linkage editor control statement. By default, it's also the module name for the INCLUDE linkage editor control statement. (You'll notice there's no space between // and PROG on the JPROC call.) You also use it as the *program-name* parameter on the EXEC job control statement.

The same job and linkage editor control statements would have been generated if you specified it like this:

```
/* (end of language translator job step)
// LINK PROG
// EXEC PROG $Y$RUN
/&
// FIN
```

Notice that PROG, in this case, is specified as the *input-module-name*, rather than the symbol. Remember, one can substitute for the other if it's omitted.

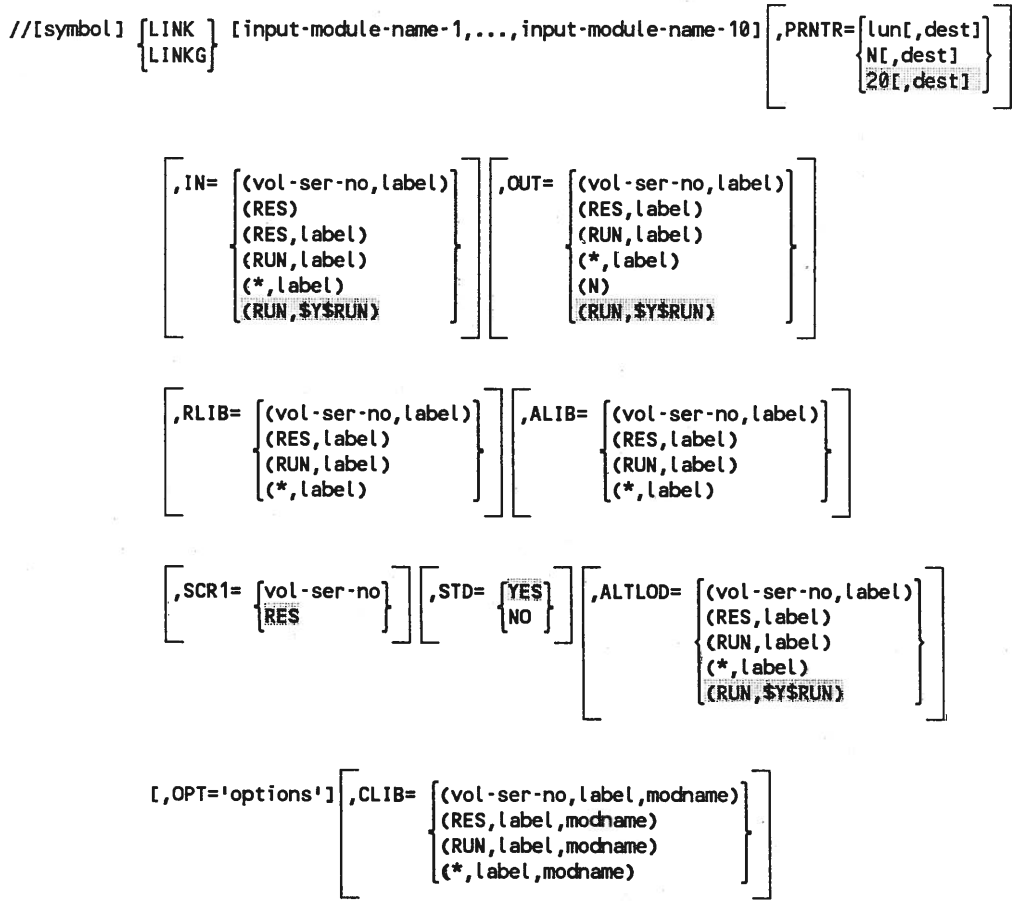
You could make this job a little easier by getting rid of the EXEC job control statement, like this:

```
/* (end of language translator job step)
// LINKG PROG
/&
// FIN
```

The following subsections describe the rest of the parameters used with the linkage editor JPROC call, and present examples showing what job control statements are generated when you specify a particular parameter.

Making the Linkage Editor Suit Your Needs

Once again, the format of the linkage editor JPROC call is:



We've already covered *symbol* and *input-module-name*, and the difference between LINK and LINKG. The remaining parameters are used to define particular input and output files, to indicate libraries to be searched for modules to be automatically included, to define scratch work areas, and to specify the alternate library that contains the linkage editor (normally it's YLOD). If you want to assign a specific printer, there's a parameter for that. And, if you are going to provide your own linkage editor control statements (you might want to do multiple link edits in a single job step), you must use a parameter to indicate this.

Let's start with the *PRNTR* parameter. If *PRNTR=N* is specified, the LINK JPROC does not generate a device assignment set for a printer. Also, it is assumed the PRINT file is not to be sent to a terminal. Remember, since no device assignment set is generated, you must supply your own. The *lun* subparameter is used if you want to assign a logical unit number for a specific printer (20 is the default, indicating that any printer can be used). The *dest* subparameter indicates the remote destination identifier (one to six alphanumeric characters) for the print output file when dealing with remote batch processing, which requires that every unit record device must have a destination.

There may be times when you want to change the spooling environment, the standard load code, or the vertical format buffer used by the linkage editor. (These buffers are changed with the SPL, LCB, and VFB job control statements, described in "Controlling Spooled Output with a Job Control Statement", "Specifying Unique Load Codes", and "Using Forms Control" in Section 6.) This is accomplished by coding N as the value of the *PRNTR* parameter. When you code N, the JPROC *will not* generate a device assignment set for the printer; you must physically insert the printer's device assignment set into the control stream *before* the JPROC call. This device assignment set consists of a DVC job control statement and an LFD job control statement (which must have a value of PRNTR for the file name). The SPL, LCB, or VFB job control statement you want to use is placed between the DVC and LFD job control statements. For example:

```
// DVC 20
// VFB LENGTH=78,OVF=75
// SPL RETAIN
// LFD PRNTR
// LINK PRNTR=N
```

Note: *Other JPROCS allow you to use the PRNTR=N parameter and supply your own device assignment set for the printer. All the language JPROCS and the JPROCS for generating control streams for data utility routines allow you to specify* PRNTR=N. *This parameter is used in these JPROCS exactly as it's described for LINK/LINKG.*

Next, let's look at the parameter for the input file definition:

```
[,IN= [(vol-ser-no,label)]
      [(RES)           ]
      [(RES,label)     ]
      [(RUN,label)     ]
      [(*,label)       ]
      [(RUN,$Y$RUN)    ]
```

The linkage editor uses two processes to include modules - specific and automatic inclusion. Modules specified in the *input-name* parameter and modules specified on embedded INCLUDE statements are specifically included. For each *input-module-name* specified, the linkage editor performs a specific inclusion search in the following manner: If the *IN* parameter is specified, only the file it identifies is searched; if the *IN* parameter is not specified, first, YRUN is searched for object modules to include. Then the file defined in the *RLIB* parameter is searched (if the *RLIB* parameter was specified) and, finally, YOBJ is searched.

For automatic inclusion, the linkage editor performs a search in the following manner: The file defined by the *ALIB* parameter is searched first (if the *ALIB* parameter was specified), and then the file defined by the *RLIB* parameter (or the default YOBJ) is searched. Modules are automatically included to satisfy the external requirements of modules that have already been included by either automatic or specific inclusion. Automatic inclusion may be suppressed by specifying the NOAUTO option.

Here are the options available to you through the *IN* parameter.

The first option is *IN=(vol-ser-no,label)*. The *vol-ser-no* is the volume serial number of the disk volume you're using, and the *label* is the file identifier of the file used when the file was created.

The next choice is *IN=(RES)*. This means the file is on SYSRES in YOBJ.

The following two choices are very similar: *IN=(RES,label)* and *IN=(RUN,label)*. In both, *label* stands for the file identifier. If you use *RES*, the file is on SYSRES; if you use *RUN*, the file is on the volume containing the job's YRUN file. (Remember, YRUN can be on the SYSRES device.)

The next choice is *IN=(*,label)*. This means the file is cataloged; therefore, its location is obtained from the file catalog.

The default parameter *(RUN,YRUN)* should not be coded when you want to use the default; its use in coding can cause an invalid file name.

Whenever you use the *IN* parameter, with both subparameters *(vol-ser-no,label*, for example), and *STD=NO* is omitted, an *INCLUDE module-name/IN* linkage editor control statement is generated.

The next parameter we'll discuss defines the output file. Here's what it looks like:

```
[,OUT= [(vol-ser-no,label)]
       |(RES,label)      |
       |(RUN,label)      |
       |(*,label)        |
       |(N)              |
       |(RUN,$Y$RUN)     ]
```

Quite frequently, you will not want to permanently save the generated load module, particularly when you don't have all the bugs out of your program. However, once the program is working satisfactorily, you'll probably want to save the load module, rather than compiling (or assembling) and link editing it every time you run it (unless it's used only once a year, for example). This is done with the *OUT* parameter.

As we've said, most times you don't want to save the generated load module for any length of time (but you'll probably want to execute it in the next job step to see how close to the finished product you are). For this reason, the linkage editor JPROC places the generated load module in the job's YRUN file by default.

But, once the module is ready to be saved, you override the default in one of these ways.

You can specify *OUT=(vol-ser-no,label)*. This is the volume serial number and the file identifier of the file where you want to store the load module.

The following two choices are similar: *OUT=(RES,label)* and *OUT=(RUN,label)*. This is like the *IN* parameter we just discussed. *label* is the file identifier; *RES* means the file is on SYSRES; *RUN* means the file is on the volume containing the job's YRUN file.

The next choice is *OUT=(*,label)*. This means the file is cataloged; therefore, its location is obtained from the file catalog.

The last choice is *OUT=(N)*. This means you don't want to save the load module at all, not even for the next job step. When this option is used, all you get is a listing of the load module, which you can use for debugging. The generated load module is not placed in any file.

Just as with the *IN* parameter, the default *(RUN,YRUN)* should not be coded.

Whenever the *OUT* parameter is coded, a PARAM OUT=OUT job control statement is generated to specify the linkage editor option that an output file has been defined for the load module. The PARAM job control statement is explained in Section 6.

The linkage editor JPROC call assumes the output file is already allocated. If it isn't, you must allocate the file by placing a device assignment set in the control stream before the linkage editor JPROC call. Let's clarify this with an example. Suppose you want to store the load module on disk volume DISK01, and you want it placed in the file identified by SAVEDPROGRAM. This file has never before been allocated. So, what you have to do is allocate the file before you can link edit the module.

You've probably noticed that the logical unit number is not coded in the *OUT* parameter (or any other except for the printer). This is because the linkage editor JPROC call uses the DVCVOL JPROC call (a JPROC call within a JPROC call, which is in turn converted to DVC and VOL job control statements). In "Too Many Devices for the Same Volume" earlier in this section we explained how there can be conflicting device assignments and how the DVCVOL JPROC call eliminates this conflict. So, we'll use the DVCVOL JPROC call in the device assignment set.

The *OUT* parameter generates a file name of *OUT* for the generated LFD job control statement of the device assignment for the output file. So, we might as well use *OUT* as the file name when we allocate the file. (We don't have to, since the program does not have to have a match for this file name; it's only serving the purpose of completing the device assignment to allocate a file. Think of it as a job step in itself.) Remember that *OUT* is the file name used by the JPROC. In "Specifying Qualifiers for File Identifiers" in Section 4 we said that, if the load module is stored in a user library (a function of the *OUT* parameter), you have to use the file name of the device assignment set for this library as a parameter in the EXEC job statement. This will be a lot clearer in the example.

```
First we start to allocate the file
using the DVCVOL JPROC call.              // DVCVOL DISK01

Next, an EXT job control statement.       // EXT ST,C,3,CYL,1

Now the file identifier,                  // LBL SAVEDPROGRAM ──────────┐
                                                                     │
and then the file name                                               │
that allocated the file.                  // LFD OUT ──┐             │
                                                      │              │
Now, the linkage editor JPROC call                    │              │
(let's call the load module XYZ),         // XYZ LINK OUT=(DISK01,SAVEDPROGRAM) ─┘
                                                      │
and execute the program.                  // EXEC XYZ,OUT ──┘
```

If the file is already allocated, the load module created is appended to the present end of the file. If a load module with the same name already exists in the file, it is replaced by the new load module.

When you specify the LINKG operation, you can't use the OUT parameter to define a specific output file. You must use the job's YRUN file.

Next, the parameters *RLIB* and *ALIB* name libraries that contain object modules, such as your own (user-written) subroutines, for inclusion in the load module. To see exactly how and why different object modules are included into your load module, see the *System Service Programs (SSP) Operating Guide*, UP-8841.

By default, the linkage editor searches YOBJ for the needed modules for automatic inclusion processing. The *ALIB* parameter allows you to specify an additional file to be searched. This file is searched first. If all the needed modules are not found here, YOBJ, or the file named by the *RLIB* parameter, is searched.

The *RLIB* parameter names the file to be searched before YOBJ during specific inclusion processing, and in place of YOBJ during automatic inclusion processing when no *ALIB* parameter is specified.

Both the *RLIB* and *ALIB* parameters look very much alike:

```
[,RLIB= [(vol-ser-no,label)]
        {(RES,label)       }
        {(RUN,label)       }
        [(*,label)         ]]
```

```
[,ALIB= [(vol-ser-no,label)]
    •   {(RES,label)       }
        {(RUN,label)       }
        [(*,label)         ]]
```

In *RLIB=(vol-ser-no,label)* and *ALIB=(vol-ser-no,label)*, you provide the volume serial number and the file identifier of the file containing the library you want.

RLIB=(RES,label) and *ALIB=(RES,label)* are similar, just as are *RLIB=(RUN,label)* and *ALIB=(RUN,label)*. The *label* provides the file identifier; *RES* means the file is on SYSRES; *RUN* means the file is on the volume containing the job's YRUN file; the asterisk (*) means the volume is identified in the file catalog.

Whenever you use the *RLIB* or *ALIB* parameters, PARAM job control statements are generated to specify the linkage editor option for libraries for inclusion processing. These PARAM job control statements are:

- PARAM RLIB=RLIB
- PARAM ALIB=ALIB

The linkage editor needs one scratch work file. Normally, SYSRES is used, but you can use a different volume:

```
SCR1= [vol-ser-no]
      {RES       }
```

This parameter, whether specified directly or indirectly through default, generates all the job control statements needed to allocate a job step temporary work file.

The linkage editor JPROC call often follows immediately after one of the language translation JPROC calls. Each of the language translators also uses scratch work files (the *SCR1* parameter; some also use *SCR2* and *SCR3*). The *SCR1* parameter coded for the linkage editor must agree with the *SCR1* parameter for the language translator; you can't contradict this assignment without getting errors. So, if you specified SCR1=DSP028 in the language translator JPROC call, you must do the same in the linkage editor JPROC call.

You've already seen that the *symbol* field provides a name for the generated LOADM linkage editor control statement, and the *input-module-name* parameters provide the names for the generated INCLUDE linkage editor control statements. However, there are times when you want to physically place these linkage editor control statements in the control stream as data; you don't want the JPROC call to generate them. You indicate this by using the *STD* parameter.

For instance, you may want to include only certain parts of an object module to form a load module. Since there's no provision for doing this with the JPROC, you have no choice but to physically place the linkage editor control statements you need in the control stream. But, you have to use the *STD* parameter to tell the linkage editor JPROC that you're going to do this, or else it automatically looks at the *input-module-name* parameters, and then the *symbol* field, for the name of an object module to include. Since you didn't specify the linkage editor control statements through the JPROC call (they're physically in the control stream), these fields would be blank.

Another case: you may want to use additional linkage editor control statements as well. (OVERLAY, for example, there's no parameter for this.) Whenever you place *any* linkage editor control statement physically in the control stream, *all* the needed linkage editor control statements must be physically placed in the control stream.

The *STD* parameter looks like this:

```
STD= {YES}
     {NO }
```

Indicating *NO* means you're going to physically place the linkage editor control statements in the control steam. The default value, *YES*, means you want them generated automatically.

STD=NO tells the JPROC to ignore any specifications in the JPROC call for automatically generating INCLUDE and LOADM linkage editor control statements.

Next, let's look at the parameter telling the JPROC where to find the linkage editor:

```
ALTLOD= [(vol-ser-no,label)]
        |(RES,label)      |
        {(RUN,label       }
        |(*,label)        |
        [(RUN,$Y$RUN)     ]
```

Normally, the linkage editor resides in YLOD. However, you may want to use a copy of the linkage editor that is not in YLOD. The *ALTLOD* parameter allows you to identify the file that contains the linkage editor you want to use. You may specify a volume serial number, *RES*, *RUN*, or an asterisk *(*)*. *RES* means the file is on SYSRES; *RUN* means the file is on the volume containing the job's YRUN file; and the asterisk means the volume is identified in the file catalog. In all cases, the label provides the file identifier. If the *ALTLOD* parameter is omitted, the normal procedure of searching YLOD and the YRUN is performed.

The next parameter we discuss is one making available certain linkage editor options. The parameter looks like this:

```
OPTION='options'
```

The options that may be specified here are all the keywords appearing in the linkage editor //PARAM and LINKOP control statements that do not need to be equated to subparameters as, for example, SHARE, NOSHARE, AUTO, and NOAUTO. Refer to the linkage editor section of the *System Service Programs (SSP) Operating Guide* (UP-8841) for all the options.

The *CLIB* parameter looks like this:

```
CLIB= [(vol-ser-no,label,modname)]
      |(RES,label,modname)       |
      {(RUN,label,modname)       }
      |(*,label,modname)         |
      [                          ]
```

You use this parameter to specify where the linkage editor control statements reside that are to be processed for this link-edit job. As the parameter indicates, you must supply the name of the source module and the file in which it resides. You must also specify the disk volume on which the file resides.

The *CMT* parameter inserts a character string in the comment field of each phase header record produced for the generated load module. Its format is:

```
CMT='comment'
```

The character string you choose may run up to a maximum of 30 characters and must be enclosed in apostrophes. It may contain blanks, commas, and other special symbols, excluding apostrophes.

The *ENTER* parameter specifies the transfer address. The *ENTER* parameter looks like this:

```
ENTER=expression
```

The expression is a decimal number from one to eight digits, a hexadecimal number from one to six digits in the form X'nnnnnn', a previously defined symbol, or a previously defined symbol plus or minus a decimal or hexadecimal number, in the form we've just discussed.

Now, let's do some coding.

```
                                                    Column 72
1.  // JOB LNKJPROC
2.  //BELLPR LINK PAYROLL,IN=(DISK01,PRAREA),                    X
    //1          OUT=(DISK01,BELLHRLPR)
3.  /&
    // FIN
```

1. This is the JOB control statement telling the operating system that the name of the job is LNKJPROC.

2. This is the JPROC call. (We're only link editing, not automatically executing, also. Thus, the operation is LINK, not LINKG. Besides, the OUT parameter is used. When an output file is specified, the LINKG operation can't be used.) As you can see, we used the *IN* and *OUT* parameters. The source deck was already compiled (let's say yesterday), and the *IN* parameter indicates it's stored in the file identified by PRAREA, on disk volume DISK01. The *OUT* parameter indicates we also want the load module to be stored on disk volume DI3K01. This payroll is for the Bell Historical Library, so we chose a file identifier that closely represents the name: BELLHRLPR. (Assume this file has already been allocated; otherwise, we'd need a device assignment set to allocate the file.)

   When the object module was created (compiled or assembled), it was given the name PAYROLL. So, this is the name of the object module we want to obtain from the file identified as PRAREA. This provides us with the *input-module-name parameter*, which generates an **INCLUDE** linkage editor control statement for this module.

   We're providing a name for the load module by using the *symbol* field. We also want to make this name readily identifiable with the company name. Since the *symbol* field is limited to six characters maximum, we can't use BELLHRLPR, as we did for the output file identifier. (Also, two identical names in the same JPROC call could cause confusion.) We'll shorten it to BELLPR. This is what will appear on the generated LOADM linkage editor control statement. When you want to execute this load module, this is the *program-name* you'd use on the EXEC job control statement.

3. This ends the job and terminates the card reader operations.

Now here's what the JPROC call generated:

```
     // JOB LNKJPROC
     // DVC 20   // LFD PRNTR
1.   // DVC 50   // VOL DISK01
     // LBL PRAREA    // LFD IN
     // DVC 50   // VOL DISK01
2.   // LBL BELLHRLPR   // LFD OUT
     // DVC RES
3.   // EXT ST,C,1,BLK,(256,10)
     // LBL $SCR   // LFD $SCR1
     // EXEC LNKEDT
4.   // PARAM OUT=OUT
     /$
5.      LOADM BELLPR
        INCLUDE PAYROLL,IN
     /*
     /&
     // FIN
```

1. This is generated by the *IN* parameter. The linkage editor uses the DVCVOL JPROC (which we're showing in its generated form: DVC and VOL). DISK01 is the first volume requested in the job, so it receives the first logical unit number: 50. The *IN* parameter always generates a file name of IN for the LFD job control statement.

2. This is generated by the *OUT* parameter. Again, DISK01 was requested in the JPROC call, and since it was already assigned to logical unit number 50, this number is assigned to this volume every time it's encountered in the job. The *OUT* parameter always generates a file name of OUT for the LFD job control statement.

3. This is the device assignment set for the scratch work area, which was generated by default in this case.

4. This is the PARAM job control statement generated by the *OUT* parameter.

5. This is the object module name (PAYROLL) and the load module name (BELLPR). These linkage editor control statements are generated by the *input-module-name* parameter and the *symbol* field. The IN shown on the INCLUDE linkage editor control statement is generated because both subparameters on the IN keyword parameter are used.

We've now covered all the parameters of the linkage editor JPROC call and provided examples of their use. You should now be able to use this JPROC call correctly.

Personalizing Your Print Output

Unisys provides the WRTBIG and WRTSML JPROCS to produce block characters on your printed output. Any type of information can be printed by WRTBIG and WRTSML - your name or a message, for example.

WRTBIG and WRTSML function in the same way; the only difference between the two is the size of the block characters produced. Those created by WRTSML are smaller than those created by WRTBIG.

WRTBIG and WRTSML produce block characters formed by the characters themselves, arranged in the pattern of the characters being printed. You can print the letters A through Z and the numbers 0 through 9. In addition, you can use WRTBIG and WRTSML to print these special characters:

(	Left parenthesis	[	Left bracket
)	Right parenthesis	]	Right bracket
+	Plus	"	Double quote
&	Ampersand	'	Apostrophe
*	Asterisk	@	At
-	Hyphen	>	Greater than
/	Slash	<	Less than
?	Question mark	|	Vertical line
:	Colon	!	Exclamation Point
#	Number	;	Semicolon
=	Equal	_	Underscore
.	Period	,	Comma
$	Dollar	%	Percent
			Embedded blank

Note: *Some printers cannot print all of these characters - check with your system administrator.*

Up to eight blocks, or lines, of print can be generated by WRTBIG and WRTSML.

Each line produced by WRTBIG can contain up to 12 characters. A maximum of four lines can be printed on each page. WRTBIG produces characters 10 characters high and 8 characters wide. The letter P, for example, prints like this:

```
PPPPPP
PPPPPPP
PP    PP
PP    PP
PPPPPPP
PPPPPP
PP
PP
PP
PP
```

WRTSML produces characters seven characters high and five characters wide. Up to 20 characters can be printed on each line, and up to 6 lines can be printed on each page. The number 7 produced by WRTSML looks like this:

```
77777
    7
   7
  7
 7
7
7
```

Note that the character produced by WRTSML is 7 characters high and the one produced by WRTBIG is 10 characters high.

The format for WRTBIG and WRTSML is:

```
//[symbol]  {WRTBIG}  'block-1'[,'block-2',...,'block-8']
            {WRTSML}
            [,IN= [(vol-ser-no,label)]]
                  [(RES,label)       ]
                  {(RUN,label)       }
                  [(*,label)         ]
                  [(RES,SYSLOD)      ]
            [,LUN= {  ({nnn} , {lfdname} [,dest])  }]
                   {  {20 }   {PRNTR   }           }
```

The *'block'* parameter is where you code the actual information you want printed on a line. Notice there are eight *'block'* parameters - one for each line of print. Each parameter is enclosed by apostrophes. You can use blanks anywhere in the field to position the characters on the page.

For instance, if you coded this:

```
// WRTSML '   RETURN','     TO',' JOHN DOE'
```

you get:

```
RRRR   EEEEE  TTTTT  U   U  RRRR   N   N
R   R  E        T    U   U  R   R  NN  N
R   R  E        T    U   U  R   R  N N N
RRRR   EEE      T    U   U  RRRR   N  NN
R R    E        T    U   U  R R    N   N
R  R   E        T    U   U  R  R   N   N
R   R  EEEEE    T     UUU   R   R  N   N

                        TTTTT  OOOOO
                          T    O   O
                          T    O   O
                          T    O   O
                          T    O   O
                          T    O   O
                          T    OOOOO

 JJJ  OOOOO  H   H  N   N      DDDD   OOOOO  EEEEE
  J   O   O  H   H  NN  N      D   D  O   O  E
  J   O   O  H   H  N N N      D   D  O   O  E
  J   O   O  HHHHH  N  NN      D   D  O   O  EEE
  J   O   O  H   H  N   N      D   D  O   O  E
J J   O   O  H   H  N   N      D   D  O   O  E
 JJ   OOOOO  H   H  N   N      DDDD   OOOOO  EEEEE
```

Notice that even though the field can be 12 characters, it does not have to be. You can put the end apostrophe after the last character for the line. Also, note that if there are over 12 characters for WRTBIG or over 20 characters for WRTSML the field is truncated.

You can also use WRTBIG and WRTSML to print the date, the time the job started, the system version number, and the job name from the JOB control statement. This is done by coding the following as the first four characters in any *'block'* parameter (nothing else can appear in the parameter; the last eight positions must not be used):

- TIM$

  This prints the time of day in the form of hh:mm:ss (hours, minutes, seconds).

- DAT$

  This returns the date, in the form of yy/mm/dd (year, month, day).

- VER$

  This gives you the version number of the operating system in use.

- JOB$

  This prints the job name from the JOB control statement.

Each option can be used alone, or combined with other options or information.

Look at this example:

```
                                                        Column 72 ┌─┐
// JOB POCO                                                       │ │
// WRTBIG '*************',                                       X │
//1       'JOB$',                                                X │
//2       'DAT$',                                                X │
//3       '*************'                                         │ │
    (Remainder of your                                           └─┘
     control stream)
```

Note the use of statement continuation. The printout would look like this:

```
. .   . .   . .   . .   . .   . .   . .   . .   . .   . .   . .   . .   . .
 ..    ..    ..    ..    ..    ..    ..    ..    ..    ..    ..    ..    ..
 ..    ..    ..    ..    ..    ..    ..    ..    ..    ..    ..    ..    ..
. .   . .   . .   . .   . .   . .   . .   . .   . .   . .   . .   . .   . .

      PPPPPP     000000    CCCCCC    000000
      PPPPPPP   00000000  CCCCCCCC  00000000
      PP    PP  00    00  CC    CC  00    00
      PP    PP  00    00  CC        00    00
      PPPPPPP   00    00  CC        00    00
      PPPPPP    00    00  CC        00    00
      PP        00    00  CC        00    00
      PP        00    00  CC    CC  00    00
      PP        00000000  CCCCCCCC  00000000
      PP         000000    CCCCCC    000000

       999      0000          /    0000      8888          /    0000      8888
      99999    00  00        //   00  00    888888        //   00  00    888888
     99   99  00    00      ///  00    00    88  88      ///  00    00    88  88
     99   99  00    00     ///   00    00    88  88     ///   00    00    88  88
      99 999  00    00    ///    00    00     8888     ///    00    00     8888
       99999  00    00   ///     00    00    888888   ///     00    00    888888
          99  00    00  ///      00    00   88    88  ///      00    00   88    88
     99   99  00    00 ///       00    00   88    88 ///       00    00   88    88
      99999    00  00  //         00  00    888888  //         00  00    888888
       999      0000  /           0000      8888   /           0000      8888

. .   . .   . .   . .   . .   . .   . .   . .   . .   . .   . .   . .   . .
 ..    ..    ..    ..    ..    ..    ..    ..    ..    ..    ..    ..    ..
 ..    ..    ..    ..    ..    ..    ..    ..    ..    ..    ..    ..    ..
. .   . .   . .   . .   . .   . .   . .   . .   . .   . .   . .   . .   . .
```

The *IN* parameter identifies the file containing either the load module WRTBIG or the load module WRTSML. If you don't specify this parameter, it is assumed that the module you want is on SYSRES in the file YLOD. If the load module is on SYSRES, but in a file other than YLOD, specify *(RUN,label)*, where *label* is the file identifier. To indicate that the load module is on the volume containing the job's YRUN file, use *(RUN,label)*. If the file containing the load module is identified in the file catalog, use *(*,label)*.

The *LUN* parameter provides the logical unit number of the printer to be used. By default, 20 is used. But, if you want a specific printer, use the appropriate logical unit number. (Make sure the rest of your print output goes to this printer.)

If the file name in the job is not PRNTR (which the programs supplied by Unisys use), you indicate this through the *lfdname* of the *LUN* parameter (this is similar to the LFD job control statement).

The *dest* subparameter indicates the remote destination identifier (one to six alphanumeric characters) for the print output file when dealing with remote batch processing, which requires that every unit record device must have a destination.

You can change the standard load code or vertical format buffer used for the job by coding *N* as the value of the *LUN* parameter. This indicates that the JPROC is not to generate a device assignment set for the printer; you must physically place the device assignment set for the printer in the control stream *before* the JPROC call.

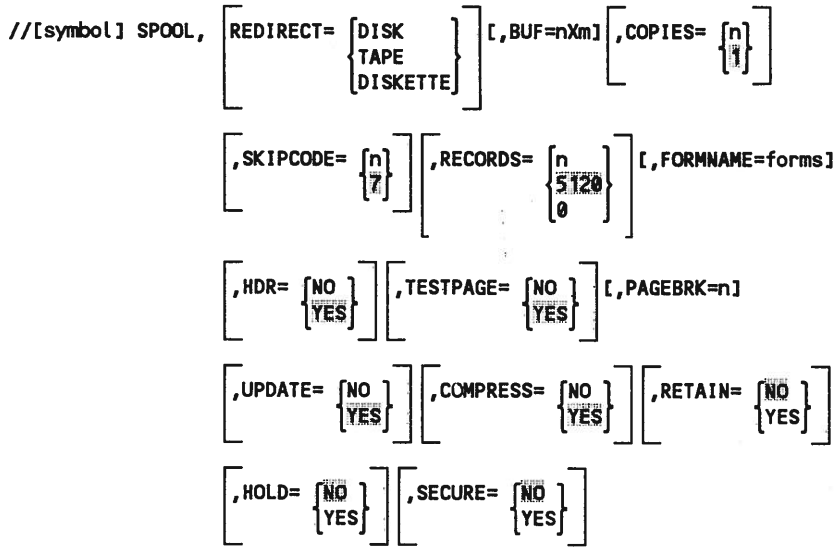
Suppose you wanted to use WRTSML to print the date at the beginning of the printout, and the file name for the printer in the program is LISTER. You would code it as:

```
// WRTSML 'DAT$',LUN=(,LISTER)
```

Controlling Spooled Output with a JPROC Call

The manner in which spooled output files (print, punch, or data-set-label diskette) are handled is set at SYSGEN time, but you can alter the standard operation of individual files with the SPOOL JPROC. To fully understand the function of this JPROC, you should be familiar with spooling, which is discussed in the *Spooling and Job Accounting Operating Guide*, 7004 4581.

When used, the SPOOL JPROC must be included in the device assignment set for the spooled output file. The format of the JPROC is:



Note: *When using the SPOOL JPROC for a spooled data-set-label diskette output file, only* BUF, RETAIN, UPDATE, COMPRESS, *and* HOLD *keyword parameters are meaningful.*

The *REDIRECT* keyword parameter redirects spooled output (output that's already in the spool file volume) to a disk, tape, or format-label diskette for temporary storage - the output is printed or punched later. A spooling component known as the output writer assigns the tape, disk, or format-label diskette volumes to be used for the redirected output so you don't have to include a special device assignment set in your job control stream for these volumes.

Notes:

1. *When you specify* REDIRECT=TAPE, *make sure that a DEV operator command, directing all spooled output to a tape volume, is not in effect for this copy of the output writer. A note to the operator should suffice.*

2. REDIRECT=DISKETTE *means redirect the spooled output to a System 80 format-label diskette only.*

The *COPIES* keyword parameter allows you to specify the number of times (up to 255) you want a spooled file printed or punched (output). If you don't specify this keyword, the file is output only once and then deleted from the spool file. If you specify 0, the output is written to the spool file but is immediately deleted instead of being printed or punched.

The *BUF* keyword parameter sets up buffers to be used by the spool subfile being created. The *n* specifies the number of buffers, *X* is a constant, and *m* specifies the size of each buffer (in 256-byte increments). If you omit this parameter, the spooled file shares the job log buffers along with other spooled files not having reserved buffers.

You must specify *SKIPCODE* if you're requesting a filed vertical format buffer (via the // VFB statement) that has more than seven skip codes or if the system default vertical format buffer has more than seven skip codes. Three skip codes are always included in this count: home position for current page, overflow for next page, and home position for next page. The four remaining are for user-specified skip codes. This parameter, therefore, specifies the total count of lines on a form where a skip code is allowed, plus three. Zero indicates no skip codes.

The *RECORDS* keyword specifies the number of records (lines, including spaces and skipped lines for print files, cards for punch files) the spool file can contain before a message asking if the job should be continued, breakpointed, or cancelled is sent to the operator. The operator receives this message only when the specified number is reached, and job processing stops until the operator replies. The specified number is rounded to the next higher multiple of 1024. For example, if you specify 7000, it's rounded to 7168. The highest number you can specify is 262,144. Value 0 specifies no limit is put on the number of records that can be entered into the spool file.

Note: *If you're executing a COBOL program that uses the WRITE verb with the AFTER clause, the number you specify for* RECORDS *should be double that of the actual number of records.*

If your spooled file is to be output on a special printer form or on special cards, you must identify the special form or card type in the *FORMNAME* parameter. The form name you specify is a 1- to 8-alphanumeric-character name assigned by your installation to each form. A message identifying the form or card type to be used is issued to the operator. Remember, a form name specified in a VFB statement overrides a form name specified in the SPOOL JPROC (see "Using Forms Control" in Section 6).

The *HDR* parameter *(HDR=NO)* suppresses the printing of a page header in burst mode at the beginning of the spooled file when it's output. If omitted, a page header is automatically printed.

If you specify the *FORMNAME* parameter, a query is directed to the operator asking if a sample test pattern page should be printed. Specifying *TESTPAGE=NO* suppresses this query. If the *TESTPAGE* parameter is omitted, the system default (YES) is used. This query does not occur for STAND1 forms.

You use the *PAGEBRK* parameter to specify the number of pages or cards to be spooled out before the file is breakpointed and printed or punched. The highest value you can enter is 32,000. When you omit this parameter, the file is printed or punched according to the burst or nonburst operating mode in effect.

The *UPDATE* parameter *(UPDATE=NO)* specifies that the spool file subdirectory entry is to be updated only when a file is closed. (In this case, if the system halts, you lose any output the program generated prior to restarting the IPL with spool file recovery.) If you omit this parameter, the spooler updates the subdirectory each time it crosses a logical track in the program file. (In this case, if the system halts, you can still print any output the program created prior to starting the IPL again.)

Using the *COMPRESS* keyword parameter *(COMPRESS=NO)*, you can prevent the system from attempting to compress data that's directed to the output spool file. Normally, you should not specify *COMPRESS=NO* if the data contains a large number of embedded blanks or if the file has a block size larger than 120. Specifying this parameter when the block size is 121 or greater results in an output spool file containing only one line per sector and requires that n x m be at least 2 x 4.

If you specify *RETAIN=YES*, the spooled output file is printed, punched, or placed on data-set-label diskette, but it is also retained in the spool file to be printed, punched, or output to data-set-label diskette again at a later time. If *RETAIN* is specified with *REDIRECT* (the first keyword parameter), the output file is redirected to a tape, disk, or format-label diskette and it is also retained in the spool file for printing, punching, or outputting to data-set-label diskette at a later time.

The *HOLD* keyword parameter *(HOLD=YES)* simply holds the spooled print, punch, or data-set-label diskette output file for later processing. (Files on hold are released when the BEGIN SPL command is issued or when a CC job control statement specifying the BEGIN SPL command is encountered in a job stream.) This parameter is useful if you have a large spooled file that will take a long time to output and you don't want to tie up the output device during peak processing time. Remember though, since the file being held remains in the spool file, there is a possibility that the spool file's available disk space may be exhausted. Also, if you specify *HOLD=YES* in conjunction with *RETAIN*, *REDIRECT*, or both, the output file is put on hold and the *RETAIN* or *REDIRECT* parameters are not acted upon until the file is released.

The last keyword parameter *(SECURE)* determines whether print output that's destined for an auxiliary workstation printer is secured or not secured. (Spooled output is directed to an auxiliary workstation printer via // ROUTE or // OPTION OUT.) We say the print file is secured if the workstation to which the auxiliary printer is physically connected must be logged on before the output file can be printed. If the workstation isn't logged on, the file will not be printed. If the file is not secure (this is the default), the file will be printed at the specified auxiliary workstation printer whether or not the workstation is logged on. Here is an example of a job using the SPOOL JPROC to control output spooling.

```
// JOB PAYROLL
// DVCVOL DSP028          Device assignment set for
// LBL JONESPAYROLL       the input file on disk.
// LFD JONESPAY
// DVC 130                                          Device assignment set for a
// SPOOL BUF=4X32              <-------------       spooled data-set-label diskette
// LFD JONESYTD                                     output file.  (See "Spooling
// DVC 20                 Device assigment set      Input Card Data" in Section 6
// SPOOL HOLD=YES         for a spooled printer     for information about the DVC
// LFD JONESCHK           output file.              statement for spooled
// EXEC JONCKS                                      data-set-label diskette files.)
/&
// FIN
```

We have now finished our discussion of what is known as basic job control. From this point on, we enter the area of advanced job control programming. You'll learn how to use the advanced job control statements to perform functions that cannot be done with the basic set. You'll also learn how to write your own job control procedure definitions, which you can store and call when needed.

By now, your grasp of job control should be such that you could construct control streams for the majority of jobs in your installation. When you finish Sections 6 through 9, you should be able to construct control streams for any job.

Section 6
Making Job Control Work for You

Advantages of Using Advanced Job Control Statements

As you have just seen, quite a lot can be done by using the job control procedure calls (JPROCS) and the basic job control statements supplied by Unisys. Now we'll see how to increase performance by using the advanced job control statements and JPROCS that you write yourself. Your basic objective is to run jobs in the most efficient, most economical, and quickest way possible. This objective is achieved not only by how you write a program, but also in the way you use job control.

Controlling Spooled Output with a Job Control Statement

In Section 5, we discussed how you can alter the standard operation (established at SYSGEN time) of spooled output files with the SPOOL JPROC. The SPL job control statement provides the same facilities and parameters as the SPOOL JPROC, so the following brief description of the SPL job control statement is essentially a review of "Controlling Spooled Output with a JPROC Call" in Section 5. When deciding whether to use the SPL job control statement instead of the SPOOL JPROC, keep the following in mind: although the SPOOL JPROC is easier to code because of its keyword (rather than positional) parameters, it takes more time for the run processor to process the SPOOL JPROC.

The format of the SPL job control statement is:

```
//[symbol] SPL [ {HOLD   } ] [,nXm] [ , {no-cop} ]
               [ {RETAIN } ]         [ {1     } ]
               [ {TAPE   } ]
               [ {DISK   } ]
               [ {DISKETTE} ]

[ , {no-skpcode} ] [ , {max-rec} ] [,forms] [ , {NOHDR} ] [ , {NOSTL} ]
   {7          }   {5120    }                {HDR  }       {STL  }
                   {0       }

[,brk-pge] [ , {NOUPD} ] [,NOCMP][,RETAIN][,HOLD][,SECURE]
           {UPD  }
```

Note: *When using the SPL statement for a spooled data-set-label diskette output file, only the* nXm, NOUPD, NOCMP, RETAIN, *and* HOLD *parameters are meaningful. The remaining parameters are ignored.*

The *HOLD* parameter holds the spooled output file (print, punch, or data-set-label diskette) for later processing. Files on hold are released by a BEGIN SPL command or by a CC job control statement specifying a BEGIN SPL command. You'll notice that *HOLD* is also the last parameter of the SPL statement. This is so you can specify *HOLD* (as the first parameter) or choose one of the other options for the first parameter and still specify *HOLD* (last).

With the *RETAIN* parameter, the spooled output file is processed (printed, punched, or placed on data-set-label diskette), but it is also retained in the spool file for processing at a later time. For the same reasons mentioned for *HOLD*, you can specify *RETAIN* as the first or the twelfth parameter.

You use the *TAPE*, *DISK*, and *DISKETTE* parameters to redirect spooled output to tape, disk, or format-label diskette for temporary storage. The output can be processed (printed, punched, or placed on data-set-label diskette) at a later time.

The remaining parameters can be summarized as follows:

- The *nXm* parameter establishes buffers for use only by the spool subfile being created.
- The *no-cop* parameter allows you to specify the number of times (from 0 to 255) you want a spool file processed (printed or punched). Zero indicates no output.
- The *no-skpcode* parameter must be specified if a filed vertical format buffer (requested via // VFB) or the system default vertical format buffer has more than seven skip codes. One skip code for forms overflow and two for home paper position are always included in this count.
- The *max-rec* parameter specifies the number of records the output file can contain before a message is sent to the operator asking if the job should be continued, breakpointed, or cancelled. Value 0 specifies that no limit is put on the number of records that can be entered into the spool file.
- The *forms* parameter identifies any special form or card type (other than the standard paper or cards) needed when the spool file is output.
- The *NOHDR* parameter suppresses the printing of a page header at the beginning of a print file; the *HDR* parameter prints it.
- The *NOTSTL* parameter suppresses the test line request message to the operator when a forms change is required. This request message does not occur for STAND1 forms. The *STL* parameter sends a test lines message to the operator when a forms change is required. If both parameters are omitted, the system default is used.

- The *brk-pge* parameter indicates a specific number of pages or cards to be spooled out before the file is breakpointed and printed or punched.

- The *NOUPD* parameter indicates that the spool file subdirectory entry be updated only when a file is closed. Value UPD enables recovery of spooled data in the event of a system crash.

- The *NOCMP* parameter indicates the system should not attempt to compress data that's directed to the output spool file.

- The *RETAIN* and *HOLD* parameters perform the same function they do when specified as the first parameter. Remember, though, if you specify *HOLD* (as the last parameter) with *RETAIN*; *TAPE*, *DISK*, or *DISKETTE*; or with *RETAIN* and *TAPE*, *DISK*, or *DISKETTE*, the output file is first put on hold. The other parameters are acted upon accordingly when the file is released. If you specify *RETAIN* (the twelfth parameter) with *TAPE*, *DISK*, or *DISKETTE*, the output is redirected to the appropriate device and a copy of the file is also retained (in the spool file) for later use.

- If specified, the *SECURE* keyword parameter indicates that the workstation to which the auxiliary workstation printer is connected must be logged on before the output file can be printed. If the workstation is not logged on and this keyword parameter is specified, the file will not be printed.

Just as described in "Controlling Spooled Output with a JPROC Call" in Section 5 for the SPOOL JPROC, the SPL job control statement must be placed in the device assignment set for the spooled file.

Sending Spooled Output to Remote Batch Processing Terminals

The DST job control statement is used to send spooled output (print or punch) to RBP (remote batch processing) terminals in your ICAM network. The format of the DST statement is:

```
//[symbol] DST dest-1[,dest-2,...,dest-16]
```

The *dest* parameter is one to six alphanumeric characters and defined by RBP. The keywords OS3CTR or CENTRAL can be used to specify the local site's printer.

The DST statement must appear within the device assignment set for the print or punch file. When specifying multiple destinations, you can list several destinations in one // DST statement or use several // DST statements each listing one or more destinations. For example:

```
// JOB REMOTE
      .
      .
      .
// DVC 20
// DST A,OS3CTR,C,D
// LFD PRINT
      .
      .
      .
// EXEC PROG1

/&
```

or

```
// JOB REMOTE
      .
      .
      .
// DVC 20
// DST A
// DST OS3CTR

// DST C,D
      .
      .
      .
// EXEC PROG1
/&
```

For more information on remote batch processing, see the *Integrated Communications Access Method (ICAM) Utilities Programming Guide*, 7004 4565.

Note: *RBP output (specified by / / DST) and DDP and auxiliary printer output (specified by / / ROUTE) cannot be mixed for any one job. For any job, all output must be of one type or the other.*

Sending Spooled Output to DDP Sites and Auxiliary Workstation Printers

The ROUTE job control statement routes print or punch output to printers and punches at DDP sites and to auxiliary printers connected to a load or remote workstation or terminal. You place the ROUTE statement in the device assignment set for the file to be routed.

Notes:

1. *Output can be routed to the site central printer and up to seven auxiliary printers.*
2. *The* // ROUTE *and* // DST *statements cannot be mixed in the same job.*

The ROUTE statement format is:

```
//[symbol]ROUTE destination-1,...,destination-8
```

You can specify up to eight destinations for non-DDP destinations, or one DDP site destination. These destinations are the central printer or punch at a local or DDP site, a workstation auxiliary printer at a DDP site, or an auxiliary printer that is locally or remotely connected to your system.

You specify the destination as follows:

```
[host-id:]user-id-1,...,user-id-8
```

The host-id identifies a particular host in a DDP network. It is one to four alphanumeric characters long and identical to the label-id of the LOCAP macroinstruction in an ICAM network. You can also use $HOST to indicate the host that initiated the job (the originator/master). The host in this case may be remote or local. A host-id is optional but must be followed by a user-id if specified. Whenever you omit a host-id, the local host (the processor on which the job is executing) is assumed.

To identify an auxiliary workstation printer, specify a 1- to 6-alphanumeric character workstation user-id. You can also use YMAS to indicate an auxiliary printer at the master workstation. The keyword CENTRAL in place of a user-id indicates a central printer or punch. Any destinations that specify a user-id (other than CENTRAL) or YMAS denote auxiliary workstation printers and are valid only for print files.

Consider the following destinations:

- CENTRAL or OS3CTR

  The output goes to the central printer or punch at the local site.

- host-id:CENTRAL

  The output goes to the central printer or punch at a DDP site (identified by host-id).

- host-id:user-id

  The output goes to an auxiliary workstation printer (identified by a user-id) at a remote host (identified by a host-id) at a DDP site. This destination is valid only for print files.

- user-id

  The output goes to an auxiliary workstation printer (identified by a user-id) that is locally or remotely connected to your system. This destination is valid only for print files.

- YMAS

  The output goes to the auxiliary printer if the master (the terminal or workstation that has control of the job when the job is executed) at the local site is a workstation. Otherwise, the destination is the central printer at the local site. The YMAS destination is valid only for print files.

The following statements route output to the central printer:

```
// ROUTE OS3CTR
```

and

```
// ROUTE CENTRAL
```

The following statement routes output to up to eight auxiliary printers:

```
// ROUTE USERID1,...,USERID8
```

The following statement routes output to the central printer and up to seven auxiliary printers:

```
// ROUTE OS3CTR,USERID1,...,USERID7
```

The following control stream contains a device assignment set for a print file which includes the // ROUTE statement.

```
// JOB OUTPUT
     .
     .
     .
// DVC 20                    ]     Print file
// ROUTE A123:CENTRAL        }     device
// LFD PRTFIL                ]     assignment set
     .
     .
     .
// EXEC PROG1
/&
```

The ROUTE statement in the preceding device assignment set routes the print output to the central printer at a remote host whose host-id is A123.

Notes:

1. *RBP output (specified by / / DST) and DDP or auxiliary printer output (specified by / / ROUTE) cannot be mixed for any one job. For any job, all output must be of one type or the other. Also, DDP destinations and local auxiliary printer destinations cannot be used for the same print file.*

2. *When a workstation or terminal starts a job that directs printed output to an auxiliary printer connected to a local or remote workstation or terminal (one that is not the originator), the user at the other workstation or terminal must be logged on with the same user-id as specified in the / / ROUTE job control statement and must issue an RP command to starts printing. See the* Interactive Services Operating Guide, *UP-9972, for more information on the RP command.*

Spooling Input Card Data

A job that reads a large volume of data through the card reader ties up the operating system by using a slow-speed device (card reader) as the means of supplying input to a high-speed processor. You can avoid this by loading the card data into a spool file (high-speed disk device) for later retrieval. In this way, the card reader can be used to transfer data to the spool file while other jobs are being executed in the high-speed processor. High-speed processing, therefore, goes on without interruption from a slow-speed card reader.

The system operator uses the IN command to initiate spooling. You must identify the card file to be spooled to the system operator, precede these cards with a DATA statement, and follow them with a FIN job control statement or another DATA statement. DATA is a control statement that identifies (to the input reader) the card data you want spooled. Its format is:

```
// DATA FILEID=file-identifier[,RETAIN][,IGNORE]
```

When the operator places your cards in the card reader and issues the IN command at the console, the card file is placed in the spool file along with the file-identifier from the DATA statement. The FIN or the final DATA card terminates the card reader. The spooled card file becomes a subfile.

Later, when your job stream is run, the subfile is read in (just as the cards are read in at the card reader, only much faster). Spooled data cards may be read by a job that's entered at a card reader or by a stored job control stream.

Note: *Input data doesn't have to be spooled before your job's processing begins, but it must be spooled by the time your program attempts to open its files.*

The job control stream must contain a device assignment set for a card file. If you've included an LBL job control statement in the device assignment set, the file identifier specified on the DATA card must match the LBL statement's file identifier. If there isn't an LBL statement, the file identifier on the DATA card must be a concatenation of the job's name and the file name from the LFD job control statement. Either way, an association is made between the file you defined in your job control stream and the subfile.

If this is the control stream,

```
// JOB BALANCE
// DVC 30
// LBL SPOOL1
// LFD READ
/&
// FIN
```

you code this DATA statement:

```
// DATA FILEID=SPOOL1
```

If this is the control stream,

```
// JOB BALANCE
// DVC 30
// LFD READ
/&
// FIN
```

you code this DATA statement:

```
// DATA FILEID=BALANCEREAD
```

The *RETAIN* parameter is used to maintain the subfile after it is processed. If you specify *RETAIN*, only the DE SPL,RDR console command can delete the subfile. The following example shows the use of the *RETAIN* parameter:

```
// DATA FILEID=BALANCEREAD,RETAIN
   data cards
      .
      .
      .
// FIN
```

You can, if necessary, place a // RUN/RV job control statement in the card deck. When the deck is spooled, the run processor calls the specified job stream. Only one RUN/RV statement may be placed within a DATA...DATA or DATA...FIN card sequence. If more than one RUN/RV statement is present, only the last statement is used.

The *IGNORE* parameter is used to permit RUN job control statements to be spooled as data. It can be used, for example, for conversion jobs. Suppose you have a card deck of control streams to be converted from another operating system to OS/3 and you have several RUN statements in the deck. When you spool the card deck, you don't want the RUN statements to call stored control streams; you want them converted to OS/3 RUN job control statements.

Let's assume we are running a conversion job named CNVT with an input card deck to be spooled named CARDIN. The DATA FILEID job control statement is coded like this:

```
// DATA FILEID=CNVTCARDIN,,IGNORE
```

Because we specified *IGNORE* in this example, RUN statements in the card deck are spooled as data cards.

Spooling Diskette Files

Just as card input can be spooled, so can input from data-set-label diskette. The operator uses the IN command to initiate the spooling and the data is placed in the spool file. It remains there as a subfile and is retrieved by either a control stream entered at a card reader, or by a prefiled job control stream. The data set label from the diskette provides the label for your spool file, while the /* statement indicates the end of the data file.

Whenever you're using input that's spooled from data-set-label diskette, specify the *I* parameter of the DVC statement for the diskette. Remember, the format of the DVC statement is:

```
//[symbol] DVC  {nnn[(n)]}  [,  {addr    }]
                {RES     }      {OPT     }
                {RUN     }      {IGNORE  }
                                {ALT     }
                                {I       }
                                {O       }
                                {REQ[(n)]}
                                {REAL    }
```

The *I* parameter tells job control that your data is in the spool file. The data cannot be retrieved from this file unless the *I* parameter is specified as follows:

```
// DVC 132,I
```

The *O* parameter is used when you want the spooled output to go to data-set-label diskette.

Equating Logical Unit Numbers to Device Type Codes

Since logical unit numbers can be changed at SYSGEN time, the possibility exists that, when running your control stream on a system other than the one it was designed for, one of your logical unit numbers may indicate a different device on the other system. For example, the system your control stream was designed for might have logical unit number 64 associated with an 8416 disk subsystem. But on the system you are running under, logical unit number 64 may be an 8419 disk subsystem - wrong device. A way to get around this is to use the EQU job control statement, which equates logical unit numbers to specific device type codes. (This device type code is always associated with this device.)

The format of the EQU job control statement is:

```
//[symbol] EQU lun-1,type-1[,lun-2,type-2,...,lun-n,type-n]
```

The *lun* parameter indicates the logical unit number you have on the DVC job control statement in the control stream. The *type* parameter is the 4- to 8-hexadecimal-character device type code for the device you are using. See Table A-1 of the *Job Control Programming Reference Manual*, UP-9984, for the codes.

The EQU job control statement, which you must place before the device assignment sets in the control stream, is effective for the entire job.

Let's assume that a job is being run on a system other than the one it was written for and that there's a possibility the logical unit numbers in the second system were changed at SYSGEN time. On your system, logical number 64 is the 8416 disk subsystem. To ensure that we get an 8416 on the other system, we insert an EQU job control statement coded as follows (the device type code - 2002 - was obtained from Table A-1 of the *Job Control Programming Reference Manual*, UP-9984).

```
// EQU 64,2002
// DVC 64
// VOL DISK01
// LBL XYZ
// LFD TEST
```

You can also use the EQU statement to specify additional logical unit numbers for virtual readers, printers, or punches. See "Using Multiple Devices, SYSRES, or the Job's YRUN File" in Section 4.

Specifying Unique Load Codes

A load code buffer controls what characters are printed by your printer. Codes corresponding to the characters on your print band or cartridge are placed in the buffer and whenever a particular code is encountered, the character equated (via the load code buffer) with that code is printed. (To simplify this discussion, we'll use the term *print cartridge* from here on to mean *print band* or *cartridge*.)

For non-SDMA printers (0770 and 0776), the default contents of the load code buffer are set at SYSGEN time; there is a unique buffer for each printer type. One of the uses of the LCB job control statement is to override these specifications - to equate different codes with different characters so that you can change print cartridges. You define a load code buffer by specifying an 8-bit code for each character on the cartridge. Whenever that code is encountered, the corresponding character is printed.

For SDMA printers (class I, II, III), each print cartridge contains its own corresponding load code buffer. Therefore, you don't need to define a unique buffer in an LCB statement when you change cartridges. If you do, it is ignored. As you'll see, though, the // LCB statement has other uses.

There are two formats for the LCB statement, one for non-SDMA printers and the other for SDMA and AP9215-1 printers. The format for non-SDMA printers is:

```
//[symbol] LCB  {X'hex-string-1' }  [ {X'hex-string-2' } ,..., {X'hex-string-n' } ]
                {C'char-string-1'}  [ {C'char-string-2'}       {C'char-string-n'}  ]

              [,CARTNAME=symbol]

              [,NAME= [48-BUS]
                      [48-SCI]
                      {63-STD}
                      [OWNLC1]
                      [OWNLC2]]

              [,CARTID= {X'aa'}]
                        {C'c' }

              [,NUMBCHAR=n]

              [,TYPE=[0770]
                     {0776}
                     [*   ]]

              [,SPACE=[X'aa' ]
                      {C'c'  }
                      [X'40' ]]
```

continued

```
[,MISM= {IGNORE}]
       {REPORT}

[,DUAL= {X'xxyyxxyyxxyyxxyy'}]
       {C'abababab'        }
       {C'bbbb'            }
       {X'yyyyyyyy'        }

[,MISMCHAR= {X'aa'}]
           {C'c' }
           {X'40'}
```

The only parameters that have practical use for SDMA printers are *symbol*, *CARTNAME*, *NAME*, *TYPE*, and *MISM*.

The format of the LCB job control statement for SDMA and AP9215 printers is:

```
// [symbol] LCB [CARTNAME=symbol]

[,NAME= {48-BUS      }]
       {48-SCI      }
       {63-STD      }
       {OWNLC1-OWNLC9}

[,TYPE=SDMA]

[,MISM= {IGNORE}]
       {REPORT}
```

The *symbol* in the label field is a 1- to 8-alphanumeric character name and can have one of the following uses:

- To specify a default cartridge name when you omit the *CARTNAME* parameter
- To specify the name of a filed load code buffer that you're changing via the job SG$PRB

Use of *symbol* will become more clear after we discuss the *CARTNAME* and *NAME* parameters.

For non-SDMA printers, you use the first parameter of the LCB statement to assign the codes for each graphic symbol on the print cartridge by specifying either the *X'hex-string'* (hexadecimal) or *C'char-string'*. You need two hexadecimal characters or one EBCDIC character for every symbol. The position of each in the string of parameters must correspond to its position on the print cartridge. As many

parameters as you need to specify the entire print cartridge may be used; you may intermix the character and hexadecimal strings as required. Since the single quote (apostrophe) and ampersand (&) symbols have special meanings to job control, they must always be coded in hexadecimal. Statement continuation is only allowed between parameters; individual character strings can't be coded on one job control statement and continued on another. When using hexadecimal character strings, the number of digits must be even.

Note: *The character strings for your printer are shown in the appropriate printer subsystem manual.*

The *CARTNAME* parameter specifies the name of the print cartridge to be used. Your installation is responsible for assigning a unique, 1- to 8-alphanumeric character name to each cartridge. SCIENCE, for example, could be used for a scientific character set.

When you provide a cartridge name in the // LCB statement, the operator is requested to mount the cartridge just before the file starts printing. The output is not printed until the operator mounts the cartridge and replies to the message. Remember, if you don't specify a cartridge name, the cartridge that's already on the printer is used. So, to ensure use of the proper cartridge and to avoid printing of the wrong characters, you should specify a cartridge name.

You can use *symbol* in the label field of the LCB job control statement (instead of *CARTNAME*) to specify a cartridge name. If you use both *symbol* and *CARTNAME* to specify a cartridge name, the *CARTNAME* parameter takes precedence.

You specify *NAME* when you want to use one of the filed load code buffers (48-BUS, 48-SCI, 63-STD, or OWNLCn) established at SYSGEN time or by use of the job SG$PRB. There is a unique 48-BUS, 48-SCI, 63-STD, and OWNLCn for each printer type. (There is also a default load code buffer for each printer type when no // LCB statement is specified.) *NAME* indicates that you want a filed load code buffer; you're not establishing your own. Therefore, CARTNAME, TYPE, and MISM are the only other parameters you can specify when you use *NAME*.

The *NAME* parameter specifies the name of the filed load code buffer, which in turn specifies a cartridge name. So, when *NAME* is specified, *CARTNAME* is unnecessary.

As mentioned earlier, you can also use *symbol* for the name of a filed load code buffer. This is done only when you are executing the job SG$PRB to change a filed load code buffer (48-BUS, 48-SCI, 63-STD, or OWNLCn) via the job SG$PRB. If this is the case, you use *symbol* to specify the name of the buffer to be changed. This is the only time symbol indicates a load code buffer name. At all other times it indicates a default cartridge name if you omit the *CARTNAME* parameter. See the appropriate installation guide for more information about the job SG$PRB.

The *CARTID* parameter specifies a cartridge or band identifier. It may be either two hexadecimal digits (X'aa') or one EBCDIC character (C'c'). This parameter is required for non-SDMA printers (0776 and 0770) and must agree with the number found physically on the cartridge.

The *NUMBCHAR* parameter applies only to non-SDMA printers and specifies the total number of graphic symbols expected on the print cartridge. As a safety check to make sure you specified all characters, this number should coincide with the number of characters specified in the character strings. When you omit *NUMBCHAR*, the number of characters specified by the character strings is assumed to be correct.

To identify the printer for which the LCB job control statement is constructed, you use the TYPE parameter. From this, we can see that an LCB job control statement coded for one type of printer cannot be used for another type.

The 0770 and non-SDMA 0776 printers are available only on the models 8/10/15/20/50 systems. For all other non-SDMA printers, enter * for the TYPE parameter and use the logical unit number to specify the type of printer. Specify SDMA for the TYPE parameter if you are a model 7E user with an AP9215-1 printer.

You specify the space, or nonprinting code, for non-SDMA printers through the *SPACE* parameter. This code is not included in either the *X'hex-string'* or *C'char-string'* parameters. It may be either two hexadecimal digits (X'aa') or one EBCDIC character (C'c'). The default value is X'40'.

A mismatch occurs when you try to print a character that is not in the load code buffer or has not been specified as a dualed character. You can use the *MISM* parameter to record character mismatch errors in the system error log by coding *MISM=REPORT*. The default, *MISM=IGNORE*, means that mismatches aren't recorded.

For the 0770 or 0776 printer, you have a choice as to the replacement symbol. If you specify in EBCDIC, you would use the *DUAL=C'abababab'* parameter, with *a* being a character that is on the print cartridge and *b* being the character that *a* replaces. For example, assume that the print cartridge contains the asterisk symbol (*) but not the question mark symbol (?). You could substitute * for ? in the printout by specifying *DUAL=C'*?'*. Every time the program outputs the EBCDIC code for a question mark, an asterisk appears in the printed listing.

If you specify in hexadecimal, you would use the *DUAL=X'xxyyxxyyxxyyxxyy'* parameter, where *xx* is the code for the character printed and *yy* is the code of the character that *xx* replaces.

We've already said that when a character mismatch occurs, you can use the *MISM* parameter to record it in the error log. For non-SDMA printers, you may also specify a character that's to appear on the printed output in case of a character mismatch; otherwise, a blank will appear (the default value X'40'). This is done with the *MISMCHAR* parameter. You can specify any character you want, in either hexadecimal (X'aa') or EBCDIC (C'c'), as long as the character also appears in either the *X'hex-string'* or *C'char-string'* parameter.

Here is an example of how the LCB job control statement is used for a non-SDMA printer:

```
                                                                Column 72
1. // DVC 28
2. // LCB C'/.=*',X'7D',C'+,$'')(-0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',   X
3. //1    NUMBCHAR=48,CARTID=X'02',TYPE=0770,DUAL=C'*?''>+<',              X
4. //2    CARTNAME=SCIENCE
5. // LFD PRINTOUT
```

1. The DVC job control statement has 28 for the logical unit number, indicating that a 0770 printer must be used.

2. This gives the actual character set for the load code buffer. Notice the shaded area; this is where a switch is made from specifying in EBCDIC (C) to hexadecimal (X). We did this because we want to specify a single quotation mark (apostrophe) for the load code. Since a single quotation mark begins and ends each character string, coding the single quotation mark as an EBCDIC character would have terminated the character string, and the remaining characters would be invalid. So, we ended the character string after the last character before the single quotation mark (the asterisk), specified the single quotation mark in hexadecimal (7D), and then continued with the next character (a plus sign) in EBCDIC. The comma character for the load code (after the plus sign) will not end the character string because it's enclosed within single quotation marks.

3. The *NUMBCHAR* parameter indicates that there are 48 characters in the print cartridge. If we missed specifying a character in the character string parameter, this would cause an error, so we'd know that we forgot a character. The *CARTID* parameter indicates a cartridge identifier of 02, and we're using a 0770 printer (*TYPE* parameter). The *DUAL* parameter indicates that three nonprintable characters (?, >, and <) are going to be appearing during the job, and gives the printable characters (*, ", and +) that will replace them.

4. When this print file is opened, the operator receives a message telling him to mount the cartridge named SCIENCE.

5. Provides the file name for the print output file and completes the device assignment set.

Here is a similar example of how the LCB job control statement is used for an SDMA printer:

```
// DVC 220
// LCB TYPE=SDMA,CARTNAME=SCIENCE
```

Some points to remember when coding the LCB job control statement are as follows:

- You can always specify the *CARTNAME* and *TYPE* parameters.
- If you specify *NAME* to indicate a filed load code buffer, you cannot specify any other parameters except *CARTNAME*, TYPE, and *MISM*.
- If you're using the job SG$PRB to change a filed load code buffer, use *symbol* to specify the name of the buffer rather than *NAME*.

Using Forms Control

A vertical format buffer controls a printer's vertical form spacing. This applies to the 0770, 0776, and 0789 printers. Codes corresponding to specific lines on a printer form are loaded into the vertical format buffer. You advance the form to a particular line by issuing a skip command in your program and specifying the code. The default vertical format buffer for each printer type is set at SYSGEN time. You can use the VFB job control statement to specify a unique vertical format buffer for a print file.

You must place the VFB job control statement within the device assignment set for the printer file to which it applies. The // VFB statement becomes effective when your program opens the print file. The format of the VFB job control statement is:

```
//[symbol] VFB  LENGTH=lines
                [,FORMNAME=symbol]
                [,USE= [STAND1       ]]
                       {OWNVF1       }
                       [OWNVF2-OWNVF9]  (SDMA and 9215 printers only)
                [,DENSITY= [6]]
                           {8}
                [,TYPE= [SDMA]  ]
                        {0770}  [,OVF=(line-1,...,line-n)]
                        {0776}
                        [*   ]
                [,OVF2=(line-1,...,line-n)][,CD1=(line-1,...,line-n),...
                [,CD15=(line-1,...,line-n)]]
                [,HP=n]
```

The *symbol* in the label field is a 1- to 8-alphanumeric character name and can have one of the following uses:

- To specify a default form name when you omit the *FORMNAME* parameter.
- To specify the name of a filed vertical format buffer that you're changing via the job SG$PRB.

Use of *symbol* will become more clear after we discuss the *FORMNAME* and *USE* parameters.

The *LENGTH* parameter indicates how many lines are on a form in the range of 1 to 192. You must use this parameter whenever you specify any of the VFB statement parameters for forms overflow (OVF1,OVF2) or vertical line positioning (CD1,...,CD15). *LENGTH* must also be specified whenever you specify *DENSITY*.

The *FORMNAME* parameter specifies the name of the printer form to be used. (This is very useful when you want your output printed on a special form.) Your installation is responsible for assigning a unique, 1- to 8-alphanumeric character name to each form. PAYCHK, for example, could be the name used for payroll checks.

When you provide a form name in the // VFB statement, the operator is requested to place that form in the printer before the file begins printing. The output is not printed until the operator loads the form and replies to the message. Remember, if you don't specify a form name, the form that's already in the printer is used. So, to ensure use of the proper form and to avoid printing on any valuable special forms, you should always specify a form name.

Remember, you can specify a form name using any of the following:

- The SPOOL JPROC (See "Controlling Spooled Output with a JPROC Call" in Section 5.)
- The SPL job control statement (See "Controlling Spooled Output with a Job Control Statement" in Section 6.)
- The *symbol* in the label field of the VFB job control statement (A form name specified this way takes precedence over a form name specified with // SPOOL or // SPL.)
- The *FORMNAME* parameter of the VFB job control statement (A form name specified this way takes precedence over a form name specified with // SPL, // SPOOL, or the // VFB statement's *symbol*.)

You specify the *USE* parameter when you want to use one of the filed vertical format buffers (either STAND1 or OWNVFn) established at SYSGEN time or via the job SG$PRB. There is a unique STAND1 and OWNVFn for each printer type. *USE* indicates that you want a filed vertical format buffer - you're not establishing your own. Therefore, *FORMNAME* and *TYPE* are the only other parameters you can specify when you specify *USE*.

As mentioned earlier, you can use *symbol* for the name of a filed vertical format buffer. This is done only if you are executing the job SG$PRB to change a filed buffer (STAND1 or OWNVFn). If this is the case, you specify either STAND1 or OWNVFn in the *symbol* field. You don't specify the *USE* parameter. This is the only time *symbol* indicates a vertical format buffer name. At all other times it indicates a default form name if you omit the FORMNAME parameter. See the appropriate installation guide for more information about SG$PRB.

DENSITY indicates the number of print lines per inch. (The default is 8.) An 11-inch form, for example, printed at a density of 8 lines per inch has 88 lines; this same form printed at a density of 6 lines per inch would have 66 lines.

Note: *If you change the print density for a print file, the forms mount message is displayed to your operator. This occurs even if the form name remains the same, and the form is not to be changed. Once alerted, your operator must reply to the message before any output printing can occur.*

We refer to the remaining parameters of the VFB job control statement as skipcodes. These codes indicate forms overflow and vertical line positioning. When you specify any one of these, you must also specify the LENGTH parameter because *line* is a decimal number in the range of 1 to whatever amount is specified by the *LENGTH* parameter. When only one line is specified for a code, you may omit the enclosing parameter. If you accidentally repeat a code for the same line, the first one is accepted and the others are ignored. (In this case, a warning message is displayed.)

The *OVF* parameter specifies the forms overflow line indicator. When an overflow code is placed in the vertical format buffer, any space operation (such as print and space) that advances the form to or beyond the overflow position causes the hardware to detect and indicate forms overflow. You can specify multiple overflow indicators. For example, you might indicate a forms overflow routine through your program that prints subtotals, and another overflow routine that goes to the top-of-forms (home paper) position of the next page.

The *OVF2* parameter specifies a secondary forms overflow position for use with the 0770 printer. You can specify multiple overflow indicators. For example, if you're going to print payroll checks and there are only 10 print lines for each check, setting up a vertical format buffer at only 10 lines is impractical. Every time 10 lines are printed, the vertical format buffer is rechecked to find the specifications for the next paycheck (spacing, etc.), even though it's the same form with the same spacing. This takes time. But if you set up the vertical format buffer length for, say, 60 lines, you could define 6 paychecks in one buffer. In this way, the vertical format buffer is checked after every sixth form instead of after every form.

When you design the VFB, bear in mind that lines can be printed (and the form advanced) beyond the overflow position. For printing of assemblies, librarian runs, dumps, etc., you should provide at least four lines between the overflow position and the bottom of the form.

The user should always specify an OVF parameter if the file is to be spooled or if you specify the *LENGTH* parameter.

The CD1 through CD15 parameters are for the device independent control character codes. These codes are used for vertical line positioning. For example, CD1=5 means, every time this code is detected, each page of your report is skipped to the fifth line. Not all codes may be used with all printers. The *Consolidated Data Management Macroinstructions Programming Guide*, 7004 4607, lists the appropriate control character codes for your printers in the section that explains the control printer forms macroinstruction.

The HP parameter specifies the line number location of the home paper position.

Notes:

1. *In a spooling environment, space must be reserved for all lines with assigned skip codes. If you specify a / / VFB statement for a spooled file and provide a full vertical format buffer specification (you do not specify a filed vertical format buffer with* USE *or* symbol*), job control reserves enough space. If, however, you request a filed vertical format buffer (STAND1 or OWNVFn) that has more than seven skip codes, or if you use a system default vertical format buffer having more than seven skip codes, you must specify the number of skip codes using the* no-skpcode *parameter in the / / SPL statement or the* SKIPCODE *parameter in the / / SPOOL JPROC.*

   When you don't use a / / SPL statement or / / SPOOL JPROC, the default is seven skip codes. Three skip codes are automatically included in this count: home position for current page, overflow for next page, and home position for next page. The four remaining are user-specified skip codes. Therefore, the / / SPL statement and the / / SPOOL JPROC specify the total count of lines on your form where a skip code is allowed, plus three.

2. *Repeat occurrences of the same skip code on more than one line are counted as separate skip codes.*

Consider the following. Suppose you want to produce a report on a special 11- by 14-inch form that prints 12 lines of data at 6 lines per inch (lpi), then skips 3 lines; prints another 12, skips 3; and so forth down the page a total of 4 times. Your VFB statement might look like this:

```
// VFB FO=WORKSHT,DE=6,LE=66,OV=61
```

You would have to identify your special printer form (WORKSHT) to the operator so that it can be loaded on an available printer. Specify your desired printing density in terms of lines per inch (lpi). Specify the overall length of the form as a function of the number of lines that could be printed on the form; in this case, 66 (6 lpi x 11 inches). And, finally, specify the line on the form at which you want the printer to advance the paper to the top of the next page, which is called the home paper position. This parameter is sometimes critical because the location of the home paper position depends on where the operator physically aligns the form on the printer. If the home paper position has been set by the operator to line 4 of the form and your program prints before skipping any lines, the first print line will occur on line 4.

If we assume that the form we're using is meant to be loaded at line 4 and that our program prints before skipping, our OV specification would be 61 as shown in the example. This would allow us to print 48 lines and skip 9, before advancing the paper to the next top-of-forms, or home paper position (4 + 48 + 9 = 61). If, however, the operator loads our form at print position 2, instead of 4, our OV specification would have to be 59, instead of 61, to maintain our desired page format. The obvious lesson in this example is that you must tell the operator how to load a special form when your output format is critical. Most of the time, you're not concerned with the exact number of lines that are printed, but only that the printed output not continue beyond a reasonable line on the form.

Some points to remember when coding the VFB job control statement are as follows:

- You can always specify the *FORMNAME* and *TYPE* parameters.
- If you specify *USE* to indicate a filed vertical format buffer (STAND1 or OWNVFn), you cannot specify any other parameter except *FORMNAME* and *TYPE*.
- If you're using SG$PRB to change STAND1 or OWNVFn, use *symbol* to specify the name of the buffer instead of *USE*.
- If you specify *DENSITY*, you must specify *LENGTH*.
- If you specify any codes (*OVF1*, *OVF2*, *CD1* through *CD15*), you must specify *LENGTH*.
- If you specify *LENGTH*, you should specify at least one overflow code (*OVF*).

Controlling Tape Units

You use the MTC job control statement to position a tape volume prior to or after job step execution. With it, you can move the tape volume to a certain block within a file or to a certain file within a multiple volume. A tape volume can also be rewound to a load point, rewound and unloaded, or have tape marks written.

You must insert the MTC job control statement at a point after the device assignment set for that tape unit. The format of the MTC job control statement is:

```
//[symbol] MTC lfdname, [BB,nn]
                        |BM,nn|
                        |FB,nn|
                        {FM,nn}
                        |WM,nn|
                        |RL   |
                        [RU   ]
```

The *lfdname* parameter specifies the same file name that was used in the device assignment set for the tape volume.

The next parameter provides seven choices; they indicate the type of operation you want done. They are:

BB - Backspace a specified number *(nn)* of blocks.

BM - Backspace a specified number *(nn)* of tape marks.

FB - Forward space a specified number *(nn)* of blocks.

FM - Forward space a specified number (*nn*) of tape marks.

WM - Write a specified number *(nn)* of tape marks.

RL - Rewind to load point.

RU - Rewind and unload the tape volume.

The amount, *nn*, must be a decimal number.

The relationship between the number of tape marks to the number of files on a volume is covered in the *Consolidated Data Management Macroinstructions Programming Guide*, 7004 4607.

The following example shows how the MTC job control statement is used:

```
// JOB TAPELIST
// DVC 90
// VOL T123
// LFD TAPEIN
// DVC 20
// LFD PRNT
// MTC TAPEIN,FB,10
// EXEC TPRNT
// MTC TAPEIN,RU
/&
// FIN
```

The first MTC job control statement spaces tape T123 forward 10 blocks prior to job step execution. The second MTC job control statement rewinds and unloads the same tape after the job step is finished. Note that the *lfdname* parameter of both MTC job control statements agrees with the *filename* parameter of the LFD job control statement.

Releasing (Freeing) a Device and Volume

Once a device and a volume are assigned to a job, they remain assigned until the job is finished. This assignment applies to all job steps of the job. But, what if your job has 10 job steps, and a certain device or volume is only used in the first job step? In effect, they can't be used by any other job until this entire job is complete. You can use the FREE job control statement to release the device and volume immediately after it is no longer needed, even though the job is not completed. However, if a device and volume are released during one job step and either one is requested in a later job step in the same job, no release occurs. This protects you from not having a needed device or volume available because it was released too soon. Remember, the entire control stream is scanned before the job begins executing.

The format of the FREE job control statement is:

```
//[symbol] FREE lfdname-1[[(DEV)],...,lfdname-n[(DEV)]]
```

The *lfdname* parameter specifies the same file name used in the device assignment set for the file.

The *(DEV)* parameter indicates that the device and volume are to be released. There is no comma between the *lfdname* and *(DEV)* parameters.

Note: *You should always specify* (DEV) *even though it's shown as optional. Additionally, you must specify* (DEV) *to free unit record devices such as card readers, card punches, printers, and workstations.*

Here's an example of a multiple-job-step job. The first job step needs the card reader. After that, it's not needed.

```
// JOB PAYROL
// DVC 50
// VOL DISK01
// LBL DETAILS
// LFD PRDISC
// DVC 30
// LFD READ01
// EXEC CARDTP
// FREE READ01(DEV)
// EXEC EDIT          ┐
// EXEC BALANC        │   Job steps that
// EXEC NEGBAL        │   don't need the reader
// EXEC WORKP         ┘
/&
// FIN
   data file
/*
```

You can also use // FREE to allow a job to be scheduled that appears to use more volumes or devices than are available.

Suppose, for example, that a job (PAYROL) uses four cataloged tape volumes. Your system has only two tape drives. The system assumes, for cataloged volumes, that each unique volume requires a unique device. Your job is not scheduled because there are not enough unique devices available for each unique cataloged volume. You can get the job scheduled, however, if you use the FREE statement to release the tape drives once the first two volumes have been accessed.

During execution of the job, you are still protected from a needed device or volume being unavailable, because no actual release occurs if // FREE specifies a device or volume needed in a later job step.

Your job stream might look like this:

```
// JOB PAYROL
// LBL FILA
// LFD TAP1
// LBL FILB
// LFD TAP2
// EXEC STEP1
// FREE TAP1(DEV)
// FREE TAP2(DEV)
// LBL FILC
// LFD TAP3
// LBL FILD
// LFD TAP4
// EXEC STEP2
```

Your job is scheduled, even though your system has only two tape devices, because the drives are freed after the first job step. During execution of the job, volumes A and C use one tape drive and volumes B and D use the other tape drive.

// FREE can be used to release a workstation when it's no longer needed by a job. You specify the workstation lfdname as it appears on the LFD statement in the workstation's device assignment set and code the FREE statement as follows:

```
// FREE WRKSTN(DEV)
```

If this statement is specified, all workstations connected to the file are freed.

Scratching Unwanted Files

Once a disk or diskette file is no longer needed, it might as well be scratched, making the space available for some other file. The SCR job control statement does this. Any file or extent specified on this job control statement is scratched as soon as the SCR job control statement is encountered by the job step processor. Therefore, the SCR statement should only be specified after any job steps needing that particular file are executed. Only files on volumes that are currently mounted when the SCR job control statement is encountered are scratched. You can't use the SCR job control statement to delete a file on SYSRES that has Y as the first three characters of the file label, and you can't use it to delete the YRUN file from the RUN volume. Only one volume serial number may be specified for any SCR job control statement.

The format of the SCR job control statement is:

```
//[symbol]SCR lfdname[, {DATE[,yyddd]}]
                      {PRE[,aaaa]   }
```

The *lfdname* parameter specifies the file name (of the file to be scratched) used in a previous device assignment set in the control stream. Within that assignment set, you must specify the volume serial number and the file identifier. But, if you're working with a disk file and the next parameter is either *DATE* or *PRE*, you may omit the LBL job control statement from the relevant device assignment set.

The *DATE* and *PRE* parameters are used only for disk files. When you use the *DATE* parameter, all files on the disk volume that have an expiration date earlier than the current system date are scratched. If you want to use a date other than the current system date, include the *yyddd* parmeter as part of your specification (where *yy* is the year and *ddd* is the day - leading zeros must be specified). When you specify a date, all the files on the disk volume dated earlier than the date specified are scratched.

The *PRE* parameter indicates that all files on a disk volume with a certain prefix are to be scratched. You specify this prefix as the next 1-to-8-character parameter. The first three characters of this prefix, however, cannot be Y if the volume is SYSRES. If you omit the *aaaa* parameter, the first four characters of the file identifier from the associated LBL job control statement are used as the prefix.

If this parameter (*DATE* and *PRE*) is omitted, the entire file specified by the *lfdname* parameter is scratched.

Here are three examples:

```
// DVCVOL DSP028
// LBL PAYROLLDETAILS
// LFD PRDET
// SCR PRDET
```

In this first example, the entire file identified as PAYROLLDETAILS is scratched. The *filename* parameter of the LFD job control statement and the *lfdname* parameter of the SCR job control statement must agree.

```
// DVCVOL DSP028
// LFD DELETES
// SCR DELETES,DATE,76002
```

In this example, all files on disk volume DSP028 that have an expiration date earlier than the second day of the year 76 are scratched. Notice the absence of an LBL job control statement. When you use either the *DATE* or *PRE* parameter, an LBL job control statement isn't needed.

```
// DVC 130
// VOL DKT001
// LBL ADDRFILE
// LFD ADDR1
// SCR ADDR1
```

Our last example shows an entire file being scratched on our format-label diskette. Remember, the *filename* parameter of the LFD job control statement and the *lfdname* parameter of the SCR job control statement must agree.

Notes:

1. *The file to be scratched should not be in use by another job.*

2. *After an SCR job control statement is processed, the file is no longer available. You can't even refer to this file with another SCR job control statement or a FREE statement.*

File Cataloging

The file catalog (YCAT) is a system resident file. It contains entries consisting of file information about tape, disk, and diskette files in the system. The catalog enables easy access to this file information for jobs and can also restrict files only to authorized users.

The CAT, DECAT, and QUAL job control statements and a special form of the LBL job control statement are used to create, access, and decatalog cataloged files. Their use and a complete description of the OS/3 file cataloging facility are contained in the *File Cataloging Technical Overview*, 7004 4615. The catalog manipulation routine (JC$CAT) is also described there.

Selecting Optional Features

Unisys Operating System/3 provides optional features you can select whenever you want. As you'll see, some options (such as DUMP and GO) are only effective during the job step in which they are specified, some (such as GABRDUMP, GDUMP, and GSYSDUMP) are effective from the time the option is encountered until end of job, while others (such as ACN, BUF, OFT, LOG, and SCAN) are in effect for the entire job because they are acted upon when the run processor prepares the control stream for execution.

This is the format of the OPTION job control statement:

```
//[symbol] OPTION p-1[,...,p-n]
```

As you can see, you can specify as many features as desired, as long as they're separated by commas (there can't be any spaces). The features available are:

- ABRDUMP

  Provides a main storage dump in the immediate vicinity of the current TCB PSW address. Displays current registers and buffers of all open DTFs.

- ACN=account-number

  Overrides the *acct-no* specified in the JOB control statement.

- BOF

  Your program is given control with binary overflow interrupt-enabled.

- BUF=nXm

  Overrides the *nXm* parameter specified in the JOB control statement.

- DOF

  Your program is given control with decimal overflow interrupt-enabled.

- DUMP

  Provides a job region dump at execution time in hexadecimal, if job step termination is requested, or a snapshot dump in response to a SNAP macroinstruction.

- EOD=xx

  Supplies substitute characters for the end-of-embedded-data (/*) job control statement. Used when embedded data is DSL source code. The first character specified must be a slash (/). The second character can be anything but a slash, an asterisk, an ampersand, or a currency symbol (/, *, &, $).

- GABRDUMP

  Specifies that OPTION ABRDUMP is in effect for every job step from the time *GABRDUMP* is encountered to end of job.

- GDUMP

  Specifies that OPTION DUMP is in effect for every job step from the time *GDUMP* is encountered to end of job.

- GJOBDUMP

  Specifies that OPTION JOBDUMP is in effect for every job step from the time GJOBDUMP is encountered to end of job.

- GO

  Automatically executes a load module after link editing is completed. An OPTION job control statement with the GO feature is generated automatically by the ASMLG JPROC call statement, for example.

- GSUB

  Provides symbol substitution for all embedded data sets in the job stream. This is a global SUB option.

- GSYSDUMP

  Specifies that OPTION SYSDUMP is in effect for every job step from the time GSYSDUMP is encountered to end of job.

- HDR= { NOHDR / HDR }

  NOHDR suppresses the printing of page separators. HDR allows page separators to be printed. OPTION HDR overrides the page separator specification in the JOB control statement.

- HOLD

  Places a job containing it in hold status while the job is in the job queue table. A job containing this option is not released until a BEGIN operator command is issued, or until a CC job control statement with BE specified is encountered in a subsequent control stream. CC BE cannot be used to release a HOLD within the same job.

- IMMOVE

  Prevents memory consolidation movable shuffle in this job step.

- JOBDUMP

  Provides an edited version of a dump if a dump is requested and is in effect only for the job step that contains this option.

- LDA

  Directs job control to set up a 256-byte user local data area (LDA) in the job prologue. This option is provided primarily for IBM® System/34 compatibility and is used if RPG programs or the assembler uses the LDA.

- LINK

  Automatically executes the linkage editor once the object module is created. This allows you to compile and link edit without intervention from job control.

- LOG= { logical-unit-number / ORIGINATOR / CENTRAL }

  Directs the job log to a specific printer or a magnetic tape. The keywords ORIGINATOR and CENTRAL refer only to printers. If you specify ORIGINATOR, the log goes to the printer at the job's originator (this includes an auxiliary workstation printer if the job's originator is a workstation). If you specify CENTRAL, the log goes to the local site's control printer. Only LOG=CENTRAL can be specified in RBP initiated jobs. The default log destination for RBP is the originator.

  For security purposes, the file passwords are not entered in the job log file.

Note: *In nonburst mode, the job log is normally printed first (on the first available printer) followed by the output file. If the DVC statement for the output file indicates a specific printer (e.g., DVC 28), you should include the OPTION LOG statement with the same logical unit number (e.g., OPTION LOG=28) so that the job log will be directed to the same printer as the output file. If OPTION LOG isn't included, the job log will be printed first (on the first available printer) and the output file will be printed on DVC 28, provided the device is available. If the device is not available, the output file will not be printed.*

If the DVC statement for the output file indicates any printer (e.g., DVC 20) and you include an OPTION LOG statement indicating a specific printer (e.g., OPTION LOG=28), both the job log and the output file will be printed (in that order) on DVC 28. For more information on nonburst mode, output spooling, and job logs, see the Spooling and Job Accounting Operating Guide, *7004 4581.*

IBM is a registered trademark of International Business Machines Corporation.

- MASTER=destination (where destination=[host-id:]user-id)

  Assigns the specified workstation (at the specified host) as the job's master - the workstation that has control of the job when the job goes into execution. (By default, the originator has control of the job so that master and originator are usually the same unless you use this option. See OPTION ORI for a definition of the originator.) The assignment as master takes effect when the job name is entered in the job queue and this assignment does not change.

  Specify OPERATOR as the user-id to designate a system console as the master. If your system has DDP, you can use a host-id to specify a particular host. If you omit the host-id, the local host (the processor on which the job is executing) is assumed. The host-id is optional but if specified, must be followed by a user-id. If you include this option in a saved translated control stream, the option will be effective when the stream is restored.

- MASTER=destination(EXEC) (where destination=[host-id:]user-id)

  Functions the same as MASTER=destination but takes effect only when the job is in execution. The originator has control when the job is in the job queue.

- MAX=maximum-main-storage-size

  Overrides the *max* parameter specification in the JOB control statement. The *max* value is interpreted as a hexadecimal value when you simply code the number or *X'number'*. You can also indicate that the *max* value be interpreted as a decimal by coding *MAX=D'number'*. If more than one maximum value is specified (via the // JOB statement or multiple // OPTION statements), the largest value is used.

- MERGE=NO

  Used to create a separate identifier for a job's log in the spool LOG file (when spooling and log accumulation are configured for the system). By including MERGE=NO, you can determine if your job log is present in the accumulated LOG file.

- MIN=minimum-main-storage-size

  Overrides the *min* parameter specification in the JOB control statement. The *min* value is interpreted as a hexadecimal value when you simply code the number or *X'number'*. You can also indicate that the *min* value be interpreted as a decimal by coding *MIN=D'number'*. If more than one minimum value is specified (via the // JOB statement or multiple // OPTION statements), the largest value is used.

- MXT=maximum-time

  Overrides the *max-time* parameter specified in the JOB control statement. The maximum time can be specified in minutes, or you can specify SUP or DEF. MXT=SUP suppresses the max-time function. MXT=DEF specifies that the system default is to be used for the *max-time* value.

- NOSCHED

  Saves a job control stream in its translated state (in YSAVE), but prevents the job from being scheduled and executed. See the SAVE option for information about subsequent runs of the saved, translated job stream.

- NOSCHED: (alt-miram-lib [, {RES / RUN / vsn}]) [,write-password]

  Functions like // OPTION NOSCHED but is used when you want the saved translated control stream placed in your own MIRAM library.

  You must use the *alt-filename* parameter to specify a 1- to 44-character file identifier. The file identifier must not be hyphenated.

  Optionally, you can specify the volume to contain the job control stream. *RES* identifies the SYSRES volume, *RUN* identifies the RUN pack, the *vsn* identifies the volume serial number of a disk pack or format-label diskette. Keep the following in mind:

  - If the file is cataloged, the volume you specify here (*RES*, *RUN*, or a *vsn*) is used instead of the volume indicated (for that file name) in the catalog.

  - If the file is cataloged and you don't specify *RES*, *RUN*, or a *vsn*, the volume indicated (for that file name) in the catalog is used.

  - If the file is not cataloged and you don't specify *RES, RUN,* or a *vsn,* the SYSRES volume is used. When you omit *RES*, *RUN*, or a *vsn*, the parentheses are optional and you can simply code // OPTION NOSCHED:*alt-filename*.

  If the file is cataloged with a 1- to 6-character write-password, you must specify the password in the last parameter.

  See the SAVE option for information about subsequent runs of the saved, translated job stream.

- NSCAN

  Resets the SCAN facility. It should be used only with embedded data of a job step for which SCAN has been specified. Subsequent job control statements normally removed by SCAN are not removed. The OPTION NSCAN statement itself is removed. When NSCAN is specified, SCAN cannot be used again in the same job step.

- NSRCH

  Only the library named on the EXEC job control statement is searched for the load module; the job run library file (YRUN) and the lead library file (YLOD) are not searched.

- NSUB

  Resets the SUB facility. It should be used only within the embedded data of a job step for which both SUB and SCAN have been specified. Set symbols in embedded data are not substituted until another SUB is encountered.

- NULL

  Specifies a no-operation for the OPTION statement.

- OFT=+n

  Tells the run processor to reserve space for an additional number (n) of files in the open file table. The *n* parameter must be in the range 1 through 16 and must be preceded by a plus sign. For IMS users, *n* is the number of terminal classes used to dynamically create files.

- OPL=option-list

  Overrides *print-option-list* specifications on the JOB control statement. Any of the options available through the *print-option-list* parameter of the JOB control statement may be specified via OPTION OPL.

- ORIGINATOR=destination (where destination=[host-id:]user-id)

  The originator is that workstation (and host) that physically initiates a job and subsequently has control of the job. OPTION ORI allows you to designate another workstation as the originator regardless of the physical originator. This option takes effect (changes *user-id*) immediately when it is encountered in the job control stream. That is, the run processor immediately changes the *user-id* to that specified by the ORI parameter. You have the option of specifying more than one OPTION ORI statement in the same job stream. In such cases, the last OPTION ORI statement encountered in the job stream designates the workstation that controls processing at execution time.

Specify OPERATOR as the *user-id* to designate a system console as the originator. If your system has DDP, you can use the *host-id* to specify a particular host. If you omit the *host-id*, the local host (the processor on which the job is executing) is assumed. The *host-id* is optional but, if specified, must be followed by a *user-id*. If you included this option in a saved translated control stream, the option will be effective when the stream is restored.

- OUT= { ORIGINATOR / CENTRAL / [host-id:]user-id }

Note: *When a workstation initiates a job that directs printed output to an auxiliary printer connected to another workstation (one that is not the originator), the user at the other workstation must issue an RP command to initiate printing. See the* Interactive Services Operating Guide, *UP-9972, for information about this command.*

Directs all job output (print files, punch files, and job logs) to the specified destination as follows:

- ORIGINATOR

  Directs all printed output to the printer at the job's originator. Directs all punch output to the central punch at the job's originator.

- CENTRAL

  Directs all print or punch output to the local site's central printer/punch.

- [host-id:]user-id

  Directs all printed output to the specified destination and all punch output to the central punch at the specified host.

  The *host-id* identifies a particular host in a DDP network, is 1 to 4 alphanumeric characters long, and identical to the *label-id* of the LOCAP macroinstruction in your ICAM network. Use $HOST to indicate the job's originator (the host that initiated the job). If the *host-id* is omitted, the local host is assumed. The *host-id* is optional but, if specified, must be followed by a *user-id*.

  A 1- to 6-alphanumeric character workstation *user-id* identifies an auxiliary workstation printer. The keyword CENTRAL in place of a *user-id* identifies the central printer or punch. Any destinations that specify a *user-id* or YMAS are valid only for print files. CENTRAL is valid for print and punch files.

This option is effective for all of the job's print and punch output, but it can be changed for individual print or punch files by specifying // ROUTE or // DST in the device assignment set for that file.

- PRI=switch-priority

  Establishes an overall task switching priority that applies to each program specified on subsequent EXEC statements in that job. This priority can be changed for particular programs by specifying a relative priority (e.g., +3 or -3) or an absolute priority (e.g., 3) on the EXEC job control statement.

- PRO

  Allows procs in embedded data.

- PRT= { ACT | LOG | NOACT | NOLOG | NONE | BOTH }

  Overrides the print option specified in the JOB control statement.

  - ACT forces the printing of accounting records.
  - LOG forces the printing of job log records.
  - NOACT suppresses the printing of account records from the job log file.
  - NOLOG suppresses the printing of log information from the job log file (including main storage dumps).
  - NONE suppresses the printing of both accounting records and log information from the job log file.
  - BOTH forces the printing of accounting records and job log information.

- PSYSDUMP

  Terminates the job immediately if abnormal termination occurs. SYSDUMP is executed as a separate job. This allows immediate rerunning of the terminated job.

- QUERY

  The OPTION QUERY job control statement is for workstation users. It allows you to change control stream execution by dynamically skipping parts of the control stream at run time. To use this facility, specify an OPTION QUERY job

control statement when you create your control stream. Then, when you run the control stream (key in RV job name) and the OPTION QUERY statement is processed, the following messages are displayed at the workstation screen:

```
JC 36 ENTER SKIP PARAMETER (DISPLAY, CANC, STEP=,LABEL=,OFF,NONE)

JC 37 UPSI=xxxxxxxx QUERY LABEL=yyyyyyyy
```

If you enter a null response to the message, the system assumes you want to proceed without a skip.

The type of skip you want is specified by keying in one of the following options:

Option	Meaning
NONE	Discontinue this function in the job step
CANC	Cancels the job
STEP=	Resume processing at the specified job step (program name)
LABEL=	Resume processing at the label specified on the NOP QUERY job control statement
OFF	Discontinue this function in the job
DISPLAY	Display all labels and job steps names in the the control stream. Step names are preceded by an asterisk (*) to distinguish them from labels.
X=	UPSI setting
Y=	Label of QUERY job control statement

To use the label skipping facility of OPTION QUERY, you must specify // NOP QUERY job control statements in the stream as targets for the skips. The NOP statement is discussed in "Providing Targets for Branching" in Section 7.

- REPEAT

  The currently executing program is automatically restarted upon termination until all embedded data files are exhausted. This gives you the ability to execute stacked assemblies or compilations without job control intervention. The REPEAT option does not clear the job region between executions, unless the ZRO option is used when linking the program.

- SAVE

  Saves a job control stream in its translated state and schedules the job to be run. A copy of the control stream as it appears in YRUN is stored in the system file YSAVE. Subsequent runs of the job are initiated through the SC/SI system

command or through the // CC SC/SI job control statement. If you elect to use this option, do not hyphenate the job name. Otherwise, an error condition will result and your job will not be saved. Saving a job control stream with a large number of JPROCS in its translated state eliminates the time-consuming chore of JPROC expansion by the run processor on subsequent runs. Information about the SC/SI system command is found in the appropriate operations guide and the appropriate workstation user guide for your system. This option is available only if your system is configured with consolidated data management.

- SAVE: (alt-filename [, { RES / RUN / vsn }] [,write-password])

  Functions like // OPTION SAVE but is used when you want the saved translated control stream placed in your own permanent MIRAM library.

  You must use the *alt-filename* parameter to specify a 1- to 44-character file identifier. (The file identifier must not be hyphenated.)

  Optionally, you can specify the volume to contain the job control stream. *RES* identifies the SYSRES volume, *RUN* identifies the RUN pack, the *vsn* identifies the volume serial number of a disk pack or format-label diskette. Keep the following in mind:

  - If the file is cataloged, the volume you specify here (*RES*, *RUN*, or a *vsn*) is used instead of the volume indicated (for that file name) in the catalog.
  - If the file is cataloged and you don't specify *RES*, *RUN*, or a *vsn*, the volume indicated (for that file name) in the catalog is used.
  - If the file is not cataloged and you don't specify *RES*, *RUN*, or a *vsn*, the SYSRES volume is used. When you omit *RES*, *RUN*, or a *vsn*, the parentheses are optional and you can simply code // OPTION SAVE:*alt-filename*.

  If the file is cataloged with a 1- to 6-character write-password, you must specify the password in the last parameter.

- SCAN

  Acts upon and removes selected job control statements (CR, GBL, GO, IF, JSET, NOP, and OPTION) from embedded data files. If this feature isn't selected, only the terminators (FIN, END, /$, and /*) are detected.

- SERIAL= { A / C }

  Allows you to have a list of RU/RV commands which are to be run serially rather than concurrently, i.e., the first job must terminate before the second job is run. The list of RU/RV commands following the option serial statement is referred to as the controller job.

The controller job can contain any number of RU/RV statements as well as other JCL statements excluding DVC-LFD sequences and EXEC statements. The only statements allowed after the last RU/RV statement are SKIP and OPR.

If SERIAL=C and a job terminates abnormally, the controller is also terminated and the following message is displayed:

```
JC61 CONTROLLER JOB JOBNAME TERMINATED ABNORMALLY
```

SERIAL=A specifies that the controller is not to be terminated. The UPSI byte is set to X'80' so the controller can skip to the abnormal path.

When the last job terminates normally, the following message is displayed:

```
JC60 CONTROLLER JOB JOBNAME TERMINATED NORMALLY
```

Multiple controllers can be active simultaneously. If you delete (DE) any job in the list, the entire process is terminated.

- SEVERE

  Specifies that the run processor is to be terminated (the job is not to be scheduled) if warning errors occur. Normally, warning errors would not terminate the job.

- SIG

  Program is given control with floating-point significant exception interrupt-enabled.

- SUB

  Scans embedded data for parameters for set symbol substitution.

- SYSDUMP

  A complete edited system dump is provided if job step termination is requested.

- TEST

  Specifies that the job is not to be queued or run.

- TRACE

  Fetches the monitor routine to record the effect of variable instruction parameters. Optional monitor tasks may be selected as described in the *Supervisor Macroinstructions Programming Reference Manual*, UP-8832.

- TSK=number-of-tasks

  Overrides the *tasks* parameter specified in the JOB control statement. From 1 to 255 tasks can be active within any job step.

- UNDEFINED

  Specifies that from the time this option is encountered to the end of job, a warning error message is to be generated whenever an undefined SET symbol is detected.

- UNEQUAL

  Specifies that a warning error message is to be generated whenever two character strings of unequal length are compared.

- XUF

  Your program is given control with exponent underflow exception interrupt-enabled.

If no dumps are requested for a job step (JOBDUMP, DUMP, or SYSDUMP), a NODUMP feature is generated, which prohibits snapshot dumps, end-of-job-step dumps, and abnormal termination dumps. This feature, NODUMP, is not to be specified on an OPTION job control statement; job control assumes this feature.

The OPTION job control statement is generally inserted as the first job control statement for the job step (unless, of course, this is the first job step, in which case the JOB control statement is first). The OPTION statement may also be used in embedded data when the NSCAN, NSUB, SCAN, or SUB features are specified.

In this example,

```
// OPTION JOBDUMP,TRACE
```

all the executed instructions of the program in this job step will be recorded. If the job step terminates abnormally or a DUMP macroinstruction is encountered, an edited dump is provided.

OPTION should not be placed between these job control statements:

- EXEC and /$
- EXEC and PARAM
- PARAM and PARAM
- PARAM and /$
- /* and /$ (where they delimit two separate embedded data sets)

Using the SET Job Control Statement

The SET job control statement modifies certain control fields in the job preamble and establishes a local data area (LDA) in the job prologue. The three fields that can be modified are: the date, user program switch indicator (UPSI), and the communications region. The SET job control statement does not alter the contents of the system information block; for this purpose, use the SET system console command.

The LDA is a 256-byte user area that follows the job accounting area in the job prologue. It is provided primarily for IBM System/34 compatibility.

We use different formats of the SET job control statement to accomplish the aforementioned functions, so we'll look at each separately.

Changing the Date

To temporarily change the date field of the job preamble until the end of the job, use this format of the SET job control statement.

```
//[symbol] SET DATE,yy/mm/dd[,t-date][,d-date]
```

The *yy/mm/dd* parameter is the date you want stored in the job preamble in place of the current date. It's specified as year, month, day.

The *t-date* parameter specifies a 5-digit date for tape files, in the form *yyddd* (2-digit year, 3-digit day). This date is stored, right-justified, in a 6-position field in the job preamble, with the leftmost position set to a blank. This *t-date* parameter is flexible. You can specify six digits, with the leftmost digit indicating the quarter of the year, and the remaining five digits indicating the date. You use this parameter when you want to compare the creation date of the first file header label (HDR1) against a date different from the date in the system information block.

The *d-date* parameter is the 5-digit date for disk files, also in the form *yyddd*. You use this if you want the format 1 label to be compared against a date different from the one stored in the system information block. If you omit the *d-date* parameter, the date specified in the *t-date* parameter is used. If you also omit the *t-date* parameter, then the date from the system information block is used.

In this example,

```
// SET DATE,90/09/14
```

the date used for the job is September 14, 1990.

Setting the UPSI

The SET UPSI job control statement allows you to set indicators that can be tested during program execution. This UPSI area is one byte long (eight bits). You can assign a specific meaning to any or all of the bits. For instance, say a program will run

with either card or tape input (two different sets of instructions defining the input device). You could code the program such that when the first bit of the UPSI byte is 1, the program instructions for card input are used; when the first bit is 0, the program instructions for tape input are used. Then, through the SET UPSI job control statement, you set the first bit of the UPSI byte to indicate which type of input is being used.

The format of the SET UPSI job control statement is:

```
//[symbol] SET UPSI,switch-setting
```

The *switch-setting* parameter is the 8-bit UPSI byte. The allowable characters are:

0 - The bit is set to off.

1 - The bit is set to on.

X - The bit is unchanged.

Unspecified rightmost bit positions are assumed to be X (unchanged). Initially, the UPSI byte is set to all zeros.

More than one SET UPSI job control statement may be specified for a job. However, you must reset conditions you don't want that have been set by a previous SET UPSI job control statement. For example, on the first SET UPSI job control statement, you want to set bits 0, 1, and 7. Code it like this:

```
// SET UPSI,11000001
```

If, on a subsequent SET UPSI job control statement in the same job, you want to set bits 0, 1, and 2, it would be coded like this:

```
// SET UPSI,XX1XXXX0
```

Since bits 0 and 1 were already set by the first SET UPSI job control statement and we want them left on, we code an X in these positions, and code a 1 to set bit 2. Since bit 7 is to be turned off, we code a 0 in this position; otherwise the 1 from the first SET UPSI job control statement would still be effective.

The Communications Region

The communications region is a 12-byte field in the job preamble that passes information from one job step to the next. For instance, assume your job has two job steps. The first job step generates input for the second. But, if this input is incorrect, you don't want to run the second job step. In the program for the first job step, you insert a routine that checks the validity of the output, and if it's incorrect, writes a code in the communications region. Then, in the program for the second job step, you insert another routine that checks the communications region. If the code is there, control is transferred directly to the end of the job.

Once you place these routines in your programs, they are there permanently unless you remove the routines and recompile the programs. It may just happen that sometimes you want to run the second job step even if the first job step was wrong (a test). Here is where you would use the SET COMREG job control statement. This allows you to change the code in the communications region.

The format of the SET COMREG job control statement is:

```
//[symbol] SET COMREG,char-string
```

The *char-string* parameter specifies the 1 to 12 EBCDIC characters or the 2 to 24 hexadecimal characters (even amounts only) to be stored in the communications region. It is stored left-justified, and any unspecified rightmost characters remain unchanged. Specify hexadecimal characters as *X'ccc...cc'* and EBCDIC characters as *C'ccc...cc'*.

At the beginning of the job, the communications region is set to 0's.

Let's say you wanted the hexadecimal code of E2 E3 D6 D7 to be stored in the first four bytes of the communications region; it would be coded as:

```
// SET COMREG,X'E2E3D6D7'
```

The User Local Data Area (LDA)

Job control support for the user local data area in the job prologue is primarily for compatibility with the IBM System/34 LDA feature. You can, at your option, use this area as a larger, more versatile communications region (COMREG). When specified, this statement automatically sets the OPTION LDA, which sets up the LDA in the job prologue. It also allows you to store character strings in the LDA. The format of the SET LDA job control statement is:

```
// SET LDA,n,m, {char-string  }
                {'char-string'}
```

The character string specified is stored left-justified in the LDA. If the character string contains blanks, it must be enclosed by single quotes.

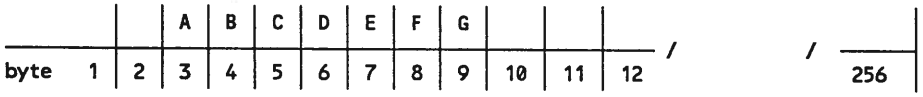
The *n* parameter specifies the byte at which the character string starts in the LDA. The lowest value for this parameter is 1.

The *m* parameter specifies the total number of bytes occupied by the character string. This value must be equal to or greater than the length of the string. It cannot exceed the length of the LDA or be a value that, in conjunction with the *n* parameter specification, extends beyond the end of the LDA.

Assume you want to insert a 7-byte string in the LDA. You want the string to begin at byte 3 and the string doesn't contain blanks; your statement is coded as:

```
// SET LDA,3,7,ABCDEFG
```

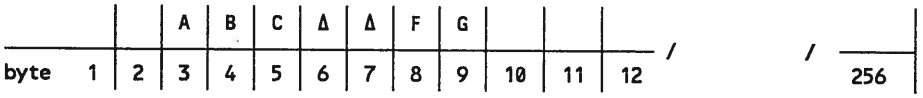
the entry in the LDA appears as:



If the string contained blanks (Δ), then your statement would be coded as:

```
// SET LDA,3,7,'ABCΔΔFG'
```

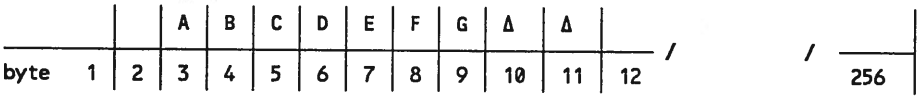
The entry in the LDA appears as:



If the total number of bytes specified (*m* parameter) exceeds the actual length of the string, the entry is left-justified and padded with trailing blanks. For example:

```
// SET LDA,3,9,ABCDEFG
```

results in the following entry into the LDA:



Keep in mind not to specify a string that exceeds the limit of the LDA (256 bytes) or to specify a string length that extends beyond the end of the LDA. For example:

```
// SET LDA,250,8,ABCDEFGH
```

This statement is invalid because you are attempting to insert an 8-byte string into the LDA beginning at byte 250. This extends beyond byte 256, the upper limit of the LDA.

Restarting a Job

In "Restarting a Job" in Section 2, we mentioned that you can restart a job that stopped running because of a computer malfunction, without rerunning the entire job from its beginning. To be specific, we provide you with a restart facility that lets you resume execution of your job from a particular job step or from a particular checkpoint record reached when the job stopped. In both cases, the job control restart (RST) statement is used to initiate the restart process. The RST statement defines the criteria for restarting the job. You simply complete the RST statement, insert it as the first statement of the job control stream for the job being restarted, and rerun that control stream. If the job is on cards, make the RST statement the first card in the deck and rerun the job deck. In cases where the job is prefiled, submit only the RST statement for the job through the card reader. If you are a workstation user and the control stream is prefiled, use the general editor (EDT) or the librarian to place the RST statement in the control stream.

The format of the RST statement varies slightly depending on whether you are restarting from a job step or a checkpoint record. There is also a certain amount of file preparation required if you want to set up restart from checkpoint records. This is discussed later in the section. The general things you should keep in mind when using the restart facility are:

- You may submit only one RST statement per job.
- Card files cannot be repositioned.
- If a multifile tape is to be repositioned, the file sequence number must be included on the LBL job control statement.
- Tapes previously positioned via an MTC job control statement are not positioned to the proper point in the restarted job.
- If a restart is to take place after the job has terminated (normally or abnormally), the restarted job step must not have originally requested temporary work areas.
- Scheduling may be delayed if all the resources needed by all job steps in the job are not available, even if those needed only by the job step to be restarted are available.
- Mount messages to the operator may be produced for volumes that were not needed for the original run because the SKIP job control statements are ignored.
- If the file containing checkpoint records is a disk file, it cannot contain any of your data.
- If the job being executed at the time a checkpoint was recorded was in the job's YRUN file (output to the linkage editor), the job being restarted will not run to normal completion if a program overlay is called after the job is restarted.

Restarting a Job from a Job Step

Restarting a job from a job does not require any special preparation other than preparing an RST statement, inserting it in the control stream for the job being restarted, and rerunning the job stream. The step processor skips to the job step specified in the RST statement and execution of the job resumes at that step. The format of the RST statement for restarting a job at a job step is as follows:

```
//[symbol] RST,,step-number[,jobname[(rename)][,pri][,key-1=val-1,...,key-n=val-n]
```

The *step-number* is the only required parameter. It specifies the number of the job step at which you want to restart the job.

The *jobname* parameter should only be used if the RST statement is submitted from a card reader.

The *rename* parameter allows you to specify an alternate name for a prefiled job that you want to restart.

You can also override the priority level originally defined for the job (JOB statement) by including the *pri* (priority) parameter. Valid entries for this parameter are P for preemptive, H for high, N for normal, or L for low. If omitted and no priority is specified on the JOB statement, the priority defaults to normal.

The *key=val* parameter represents keywords and their values that may be referenced like the parameters of a GBL job control statement. (See "Local Status Set Symbols" in Section 7.) The effect of these parameters is as if a GBL job control statement were inserted as the first job control statement of the job. The total length of the value for the parameters cannot exceed 44 characters.

The following coding of the RST statement restarts the job MYJOB at the beginning of job step 3. The control stream can be assumed to be prefiled since the job name is given, and the priority will be established by the original job statement, if specified; otherwise, it will be run at a normal priority.

```
// RST ,,3,MYJOB
```

This same statement can define an alternate name for the job MYJOB by adding the *rename* parameter. In this case, MYJOB is renamed NEWNAME.

```
// RST ,,3,MYJOB(NEWNAME)
```

Likewise, you can change the priority level of the restarted job by including the *pri* parameter. In the coding example shown, the priority for the job is changed to high (H).

```
// RST ,,3,MYJOB(NEWNAME),H
```

Restarting a Job from a Checkpoint Record

To restart a job from a checkpoint record requires that you first establish checkpoint records in your program. In a BAL program, this is done with the CHKPT macroinstruction. In a COBOL program you use the RERUN clause. You must also define a file (through use of the CHKPT macroinstruction and RERUN clause) into which the checkpoint records are written as they are encountered during your program's execution. Therefore, you must provide a device assignment set for this file in the control stream for the job. Once you have completed this, you may use the RST job control statement to restart the job if it stops. The process is the same as that described for restarting from a job step. That is, prepare the RST statement, insert it as the first statement in the control stream for the job, and rerun the job control stream. (See the previous subsection, "Restarting a Job from a Job Step".) However, the format for the RST statement requires two additional parameters when used to restart a job by checkpoint records; they are *filename* and *checkpoint-id*. All of the remaining parameters are the same as those described in "Restarting a Job from a Job Step".

```
// [symbol] RST filename,checkpoint-id,number[,jobname[(rename)[,pri]
          [,key-1=val-1,...,key-n=val-n])
```

The *filename*, *checkpoint-id*, and job step *number* are required parameters and must be specified in the order shown.

The *filename* parameter identifies the file into which the checkpoint records were written. Therefore, the file name you specify in the RST statement must agree with the file name specified in the LFD control statement of the device assignment set for the checkpoint record file. A word of caution; the LFD job control statement for the checkpoint file must not contain the *INIT* parameter because the use of this parameter will begin writing at the beginning of the file. (See "Specifications for Existing Files" in Section 4.)

The *checkpoint-id* parameter specifies the particular checkpoint that you want to restart the job from. You obtain this number from the screen of the system console. A checkpoint number is displayed on the console screen each time a checkpoint record is written by your program.

The following example shows how you would code the RST statement to restart a job named POCO. The job is to resume execution from checkpoint 6 (the number displayed at the system console at the time the job stopped) in job step 2. The file containing the checkpoint record is identified as CHKPTLOG. This is the same name as that specified on the LFD job control statement used in defining the file earlier in the job control stream.

```
// RST CHKPTLOG,6,2,POCO
```

Suppose there is a possibility that another job named POCO is scheduled for execution. To be safe, you can rename the job to be restarted as follows:

```
// RST CHKPTLOG,6,2,POCO(NEWNAME)
```

And, by including the *priority* parameter, you can redefine the priority for the restarted job. For example:

```
// RST CHKPTLOG,6,2,POCO(NEWNAME),L
```

Issuing System Commands

The CC job control statement allows you to issue OS/3 system console and workstation commands, with their associated parameters, from within a job control stream. Because there are many system commands, we will not attempt to discuss each one here. You can find the formats and descriptions of system console commands in the appropriate operations guide. Workstation commands are described in the *Interactive Services Operating Guide*, UP-9972. The format of the CC statement is:

```
//[symbol] CC  {command                }
               {'command and parameters'}
```

When enclosed in single quotes, any system console or workstation command and parameters can be specified in the CC statement. When the command has no associated parameters or when you do not specify any parameters, the quotes are not used.

Let's say you want to release a job (JOB1) that's being held as the result of a HOLD system command. If you specify the BEGIN command in a CC job control statement, you can include this statement in the job you're going to run. JOB1 will be released when this statement is processed (at your job's execution time). You would code the CC statement as follows:

```
// CC 'BE JOB1'
```

Suppose you wanted to initiate the general editor from a job control stream. The workstation command for the general editor is simply EDT. Because there are no parameters, you'd code the CC statement as follows:

```
// CC EDT
```

Whenever parameters are specified with a command, the total number of characters within the quotes cannot exceed 60.

The CC statement is examined for syntax errors by the run processor during job stream validation. If no syntax errors are found, the job is queued. The command and its associated parameters are sent to the system when the CC statement is encountered by the job step processor. The command is validated by the system independently of your job, so errors associated with satisfying commands do not terminate a job stream. If no EXEC statement follows a CC statement, the specified commands are acted upon prior to job termination.

Notes:

1. *The following system console commands cannot be specified in the CC job control statement: MIX, SWITCH, AVR, REBUILD, SHUTDOWN, SYSDUMP, and all SET commands.*

2. *When the command string contains no blanks (other than the blank separating the command from its first parameter), you can precede the first parameter with a comma instead of enclosing the command and its parameters in single quotes. For example: // CC BE,JOB1*

3. *Unsolicited input messages (see the* Interactive Services Operating Guide, *UP-9972, and // PAUSE responses cannot be specified in the CC job control statement.*

Calling Control Streams

As we mentioned in "Running Job Control Streams" in Section 1, there are several methods available for calling control streams. System console or workstation commands such as RUN/RV and SC/SI can be used, but we'll discuss only the methods available through job control.

The following job control statements are used to call control streams:

```
// RUN

// RV

// CC SC

// CC SI
```

Note: *The run processor (RUN/RV commands) and restore processor (SC/SI commands) do not allow any volumes for a multivolume, single-mount file to be RES or RUN packs.*

The RUN and RV job control statements are discussed in the next subsection, "Using the RUN/RV Job Control Statements to Call Control Streams." Using the CC SC/SI statement to call saved, translated streams is discussed in "Using CC SC/SI to Call Saved Translated Control Streams" later in this section.

Using the RV/RUN Job Control Statement to Call Control Streams

The RV and RUN job control statements are used in a job control stream to call another job control stream. They both select the stream you name and prepare it for execution. You use RV when you're calling a prefiled control stream that does not need a card reader. RUN is used only when the control stream you're calling needs a card reader.

An input device is unnecessary (// RV is used) when the control stream you want is in YJCS or an alternate library file and doesn't contain a CR job control statement.

A card reader is necessary (therefore, // RUN is used) when the control stream is on cards or when the control stream is stored but contains a CR job control statement. The CR statement in a control stream indicates that data on cards is to be accepted from the input device and inserted into the stream. (See "Adding Cards to a Stored Control Stream" later in this section.)

Although you can use // RUN when an input device is not required, you should use // RV. Using the RUN statement wastes time because your job will not be initiated until the (unnecessary) card reader is available. On the other hand, your job will not be initiated at all if you use an RV statement to call a control stream that needs a card reader.

The format of the RV/RUN statement is:

```
//[symbol] [RV jobname[new-name)]                                          ]
         {RUN[ [jobname[(new-name)]]                                       ]}
              [(new-name)          ]

[ [:alt-filename                                                           ] ]
{ [: (alt-filename, [RES])                                                 ] }
  [                 [RUN]                                                  ]
  [                 [vsn]                                                  ]
  [                                                                        ]
  [: (alt-filename, [ [RES] ] ,read-password)                              ]
  [                 [ [RUN] ]                                              ]
  [                 [ [vsn] ]                                              ]

[ , [PRE ]  [      ]                                                       ]
  [ [HIGH]  [ [time ] ]                                                    ] [key-1=val-1,...,key-n=val-n]
  [ [NOR ]  [ [time+n] ]                                                   ]
  [ [LOW ]  [      ]                                                       ]
```

This statement's parameters are similar to those of the RUN/REV console command and are explained in detail in the appropriate operations guide. They are also explained in the *Job Control Programming Reference Manual*, UP-9984.

Using CC SC/SI to Call Saved Translated Control Streams

Recall from earlier sections that a job control stream can be saved in its translated state (after JPROCS have been expanded) by including an OPTION SAVE or OPTION NOSCHED job control statement in your control stream. When the stream is run, a copy of it is stored in the system file YSAVE. Subsequent runs of the control stream can be initiated through the SC/SI command or the CC job control statement specifying the SC/SI command. We are interested in the CC job control statement here. The format of the CC statement is:

```
//symbol] CC  {command                }
              {'command and parameters'}
```

The format of the command we want to specify is:

```
{ SI [ {(did)          } ]  jobname[(new-name)]
     [ {([did],label)  } ]
     [ {(RDR,label)    } ]

  SC
  [ [ {:alt-filename                                                  } ] ]  [ , {PRE } [ {time   } ] ]
  [ [ {: (alt-filename, {RES} )                                       } ] ]  [   {HIGH}  [ {time+n} ] ]
  [ [ {                 {RUN}                                         } ] ]  [   {NOR }               ]
  [ [ {                 {vsn}                                         } ] ]  [   {LOW }               ]
  [ [ {: (alt-filename, [{RES}] ,read-password)                       } ] ]
  [ [ {                 [{RUN}]                                       } ] ]
  [ [ {                 [{vsn}]                                       } ] ]
```

You use the SI command to initiate a job control stream that requires replacement of embedded data from an input device (card reader, data-set-label diskette, or input spool file). The SC command is used only to initiate a job control stream that does not require the use of an input device to replace embedded data. Consider the following examples:

- // CC 'C MYJOB'

  This statement initiates the translated job control stream called MYJOB.

- // CC 'SI MYJOB(NEWDATA)'

  This statement initiates the translated control stream MYJOB. MYJOB is to be run under the new name NEWDATA. The replacement embedded data for MYJOB is expected to be found on the first available card reader.

Notes:

1. *Further explanation for the SC/SI command and its associated parameters can be found in the appropriate operations guide.*

2. *When substituting embedded data, the DATA STEP statement must be used. It is explained in "Substituting Embedded Data" later in this section.*

3. *When embedded data is submitted on diskette, the diskette must be a data-set-label diskette, and the record size must be 128 bytes or less. The records must be unblocked and unspanned.*

Communicating with the System Operator or Workstations

You can send a message to the system console or specific workstations with the OPR job control statement. The message you specify is displayed at job step processor time. The format of the OPR statement is:

```
//[symbol] OPR comment-line[,destination-1,...,destination-n]
```

You use the *comment-line* parameter for the text of your message, which can contain up to 60 characters and must be enclosed in single quotes if it contains embedded blanks, the slash character, or commas.

The *destination* parameter is provided for those systems with workstations or DDP. If your system has neither, the *destination* parameter is ignored and your messages go to the system console.

A *destination* is actually a host-id, user-id pair:

```
destination=[host-id:]user-id
```

The user-id directs the message to a particular workstation. The host-id allows users who have DDP to direct the message to a workstation or system of console at a particular host. If your system does not have DDP, you'll only be interested in the user-id portion of the *destination*.

The user-id can be any 1- to 6-alphanumeric character workstation user-id. You can also specify the keyword OPERATOR or YCON to denote the console workstation, or YMAS to denote the master workstation. If you omit the destination, YMAS is assumed. (See the OPTION MAS and OPTION ORI statements in "Selecting Optional Features" earlier in this section for more information about originator/master workstations.)

The host-id is 1- to 4-alphanumeric characters and is identical to the label-id of the LOCAP macroinstruction in your ICAM network. The host-id is optional but if specified, you must follow it with a user-id. You can also specify $HOST as a host-id. $HOST simply means that the host of the master (the originator of // JNOTE) is used.

If you specify a user-id but omit the host-id, the local host (the processor on which the job is executing) is assumed. Remember, if you omit a destination entirely, the message goes to the job's master workstation.

Consider this example. Suppose you want to tell the operator an error is going to occur but that the job is to continue processing. You could code the following:

```
// OPR 'AN ERROR WILL OCCUR - DO NOT CANCEL JOB', OPERATOR
```

OPERATOR is the destination, so the message is directed to the console and the local host is assumed. (Without DDP, the processor is always a local host.) The following is a list of other sample destinations you could specify in the // OPR statement:

- USER01

  The message is sent to workstation USER01. (The local host is assumed).

- YMAS

  The message is sent to the master workstation. (The local host is assumed).

- No destination specified

  The message is sent to the master workstation.

- A321:USER01

  The message is sent to workstation USER01 at host A321.

- $HOST:OPERATOR

  The message is sent to the console workstation at the originator/master host.

Messages sent to workstations that are not logged on are not rerouted unless they were intended for the master workstation (YMAS). The system reroutes such messages to the console.

The PAUSE job control statement lets you send messages to the system operator or specific workstations; however, it causes the job's processing to stop until the message is acknowledged. (Processing of other jobs in the system continues without interruption.) Regardless of the PAUSE statement's position within a job step, the message is displayed just before execution of the program within the job step. The PAUSE statement has the following format where the *comment-line* and *destination* parameters are identical to the corresponding parameters in // OPR:

```
//[symbol] PAUSE comment-line[,destination-1,...,destination-n]
```

Suppose you want the operator to check a job's printer listing for errors before the job is run. You might code the PAUSE statement like this:

```
// PAUSE 'CHECK FOR ERRORS - IF NONE, CONTINUE, OTHERWISE CANCEL', OPERATOR
```

Job processing stops until the operator acknowledges the message by cancelling or continuing the job. When multiple destinations are specified, the acknowledgements are requested one at a time, not all at once.

The JNOTE job control statement allows you to send messages to the system operator or a particular workstation. Unlike // PAUSE, however, // JNOTE does not stop job processing and does not require acknowledgement. // JNOTE is like OPR except that it's acted upon by the run processor so you can send messages earlier on in the job's processing - before job execution actually begins. Format of the JNOTE statement is:

```
//[symbol] JNOTE comment-line[,destination-1,...,destination-n]
```

The parameters function the same as // OPR and // JNOTE parameters; however, you cannot specify YMAS as a user-id. You can specify YORI to indicate the originator of the job. This is also the default if no destination is specified. Messages sent (via JNOTE) to workstations that are not connected are not rerouted unless they're intended for the originator workstation (YORI). The system reroutes such messages to the console.

Introducing Processing Options

Some programs are written to perform a variety of functions in addition to their main processing function. These programs must be told what variable functions to perform when the job is run. A good example of this type of program is a language translator, which can produce a series of special services if they are requested, but which are not desirable with every compilation or assembly. You submit these requests with PARAM job control statements.

Since PARAM job control statements are read by the individual program, you design the content and format of the information when you write the program. PARAM statements are prepared and read as embedded data.

There is no limit to the number of PARAM job control statements allowed in the control stream, and each one can contain up to 62 characters of information. However, any information beyond column 71 is ignored. You must place the PARAM job control statements immediately *following* the EXEC job control statement.

The format of the PARAM job control statement statement is:

```
//[symbol] PARAM operand-1[,...,operand-n]
```

The operands are the variable information you want to introduce into the job. If the information contains embedded blanks, it must be enclosed by single quotation marks.

Assume that in a program named LISTX, you set a variable option called LST=, which defines the line spacing on the printer. The values you defined in the program are A for a single space, B for a double space, and C for a triple space. On this running of the program, you want to triple space, so it would be coded as the following:

```
// EXEC LISTX
// PARAM LST=C
```

Defining Software Facilities Needed by Your Job

OS/3 automatically loads all of the shared-code modules needed by your job; you do not need to identify these shared code modules in order for them to be loaded. If, however, you have written your own shared-code modules and they are not on YLOD or the volume that contains your job's YRUN file, you must use the // SFT statement to identify these modules to the system.

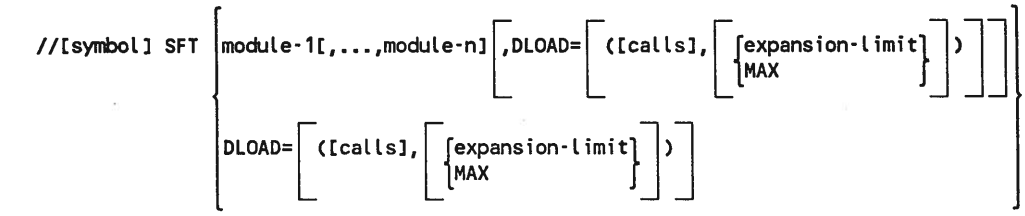
You can also use // SFT to identify data management shared-code modules that you want loaded *prior* to job initiation. This ensures that your job does not have to wait until a particular shared-code module it needs becomes available. The data management shared-code modules loaded prior to job initiation stay resident for the duration of the job.

The // SFT statement may also be used to indicate that dynamic loading is needed or to override the system generation limits for dynamic expansion of the user job region. (This feature is for ANSI'74 COBOL users.)

Let's review the applications for the SFT job control statement. You use // SFT to:

- Identify user-written shared-code modules that are not in YRUN or YLOD
- Identify data management shared-code modules that you want loaded prior to job initiation
- Specify dynamic loading and/or override SYSGEN limits for dynamic expansion of the user job region as established by the SYSGEN parameter DLOADBUFR (ANSI'74 COBOL users only)

The format of the // SFT statement is:



The *module* parameters identify to the run processor the user shared-code modules needed in a job step or the data management shared-code modules that you want loaded prior to job initiation. (User shared-code modules are always loaded prior to job initiation.) The *Supervisor Macroinstructions Programming Reference Manual,* UP-8832, lists all the shared code modules and their functions.

The SFT statement identifies shared-code modules only for the job step in which it appears. If you need the same shared-code modules in three job steps, for example, you must code an SFT statement for each of the three job steps.

Suppose you want to load, prior to job initiation, the data management module that provides for magnetic tape file output in the last step of your job. The module that performs this function is named DD$T1110. You would code

```
// SFT DD$T1110
```

and place it in the control stream for your job. The run processor would detect the SFT statement while scanning the control stream and the shared-code module DD$T1110 would be loaded before your program is executed.

Notes:

1. *When preparing a job, you must not request more shared-code modules than were provided for when your system was generated, or the job will not be scheduled.*

2. *Data management shared-code load modules reside in the system library YSCLOD. You can use the SAT librarian to get a listing of these modules and to obtain information related to each, or, if you have interactive facilities, you can use the FST command and specify YSCLOD.*

3. *There is a system generation parameter (IGNORESFT) that allows you to specify that // SFT job control statements be ignored. This system generation option is useful because you can then take advantage of the dynamic shared-code feature of OS/3 without having to change existing control streams that contain // SFT job control statements. The appropriate installation guide contains more information about this system generation option.*

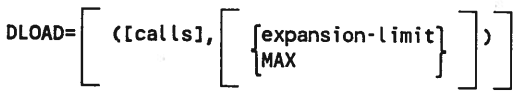
The DLOAD parameter of the SFT job control statement may be used only with ANSI'74 COBOL programs. DLOAD tells the run processor that your job needs the OS/3 dynamic loading facility for externally referenced program modules and indicates the space requirements for dynamic loading.

Normally, the run processor checks the load module and determines from the phase header record whether a job needs dynamic loading of main storage. If it does, the supervisor then allocates space for dynamic loading, immediately following the user job region, according to the limits specified at system generation. In the following instances, however, the DLOAD parameter may be needed:

- If your COBOL program references modules not in YRUN or YLOD that reference other program modules that would make it impossible for the run processor to determine whether these externally referenced modules require dynamic loading.

- If you want to override the SYSGEN-specified limits for dynamic expansion of the user job region.

The format of the *DLOAD* parameter is:



The *calls* specification indicates the maximum number of dynamically loaded modules allowed for a job. The *expansion-limit* specifies the maximum number of bytes (total) that can be added to a job in support of the DLOAD facitily. The number is considered hexadecimal if you code *X'number'* or *number*. It is considered decimal if you code *D'number'*.

If you code

```
// SFT DLOAD=(5,5000)
```

five DLOAD calls will be allowed for in this job, and the job will be allowed to expand a maximum of X'5000' bytes over its initial main storage allocation.

The *MAX* specification indicates that the size of the job is limited only by the amount of main storage in the system.

If you omit the number of calls, the system default number of calls (set by the SYSGEN parameter DLOADTABLE) are allowed. If you omit the expansion limit, the system default for the expansion limit (set by the SYSGEN parameter DLOADBUFR) is used. If you code / / *SFT DLOAD=* then both these defaults apply.

If your job region must be expanded to accommodate the DLOAD facility, the system allocates contiguous main storage immediately following your job. This may involve moving your job to a larger region in main storage. If a large enough region does not exist, an error message is generated - unless your system is generated with the DLOAD facility.

If the DLOAD facility has been specified at system generation and there is not enough contiguous space to accommodate your expanded job, your job is rolled out to disk until the required contiguous main storage is made available through:

- Main storage consolidation
- Roll-out of other lower priority jobs
- Waiting until other jobs terminate, freeing the required contiguous space

Note: *Other jobs can only be moved or rolled out to free main storage for your expanded job after your job has been rolled out.*

There are several points to keep in mind about DLOAD. If the DLOAD facility was not specified at system generation, an error may occur if enough main storage does not exist to dynamically expand your job. On the other hand, if you specified the DLOAD facility at system generation, your job might be rolled out for a long time. Even if you do not need to roll out jobs, the DLOAD facility takes time. One way to avoid this problem is to allow for a larger *initial* main storage allocation for your job through the JOB control statement. Suppose, however, that you do need the ability to dynamically expand your job size and you have specified the DLOAD facility at system generation. To avoid being rolled out for an extended period of time, you can:

- Run your job on a system generated with main storage consolidation
- Run your job with preemptive priority (P) specified on the JOB control statement
- Avoid running your job when other large or long-running jobs are using main storage

Note: *Jobs that use files with locks set cannot be rolled out to accommodate dynamic expansion requirements.*

For more information about specifying the DLOAD facility at system generation, see the appropriate installation guide.

Suppose your job needs the shared code module SINCOS and you want to override the SYSGEN limits for dynamic expansion of your job. You need to allow for six DLOAD calls with a total expansion limit of X'8000' bytes over your initial main storage allocation. Your SFT job control statement would look like this:

```
// SFT SINCOS,DLOAD=(6,8000)
```

Making Temporary Changes to a Load Module

You use an ALTER job control statement to make minor temporary changes in up to eight bytes of a load module to see if the changes have the desired effect before these changes are made permanent. Recompiling and link editing are time-consuming. As many ALTER job control statements as you need to change the module are grouped before the EXEC job control statement.

The format of the ALTER job control statement is:

```
//[symbol] ALTER [phase-name][,address][,change][, {RESET}]
                                                 {ORG  }
```

The *phase-name* parameter is either the 8-alphanumeric-character name of the phase assigned by the linkage editor or the 1- to 6-alphanumeric-character alias name of the phase. If you omit this parameter, the last phase name used on an ALTER job control statement in this job step is used.

The *address* parameter is the 1- to 5-digit starting location address where changed information is to be stored. The number you specify for the address is considered hexadecimal if you code *X'number* or *number*. It is considered decimal if you code *D'number*. This is in relation to the first byte of the phase area. If you omit this parameter and an address is required, an address of zero is used. An address is not required when *RESET* is used as the fourth parameter.

Note: *If the address given is invalid, a change does not take place.*

The actual information to be placed in the phase is specified with the *change* parameter. You can specify it in either EBCDIC or hexadecimal. EBCDIC information takes the form C'c...c'. The maximum number of EBCDIC characters is eight (eight bytes). The maximum number of hexadecimal characters is 16 (eight bytes).

If you omit the *change* parameter, no modification is made for this ALTER job control statement alone, but the information it does contain, such as phase name, is passed to subsequent ALTER job control statements.

The *ORG* parameter indicates that the address specified in the *address* parameter should be added to all the addresses on succeeding ALTER job control statements, until one with a *RESET* parameter or a different phase name is encountered.

Once an ALTER job control statement is encountered, each and every phase of the load module expects an ALTER job control statement. This is the reason for the *RESET* parameter. It indicates that no other ALTER job control statements are in the control stream.

Consider these examples:

```
// ALTER TSTPGM00
// ALTER ,4361,X'FAF3F9'
// ALTER ,4700,X'F8'
// ALTER ,,,RESET
```

If a *RESET* parameter is specified, the information is passed along to the program execution phase. When the phase that had the *RESET* parameter specified is loaded for the first time, the option is reset so that no other phases will be altered. This saves time if a phase that is only loaded once is the only phase requiring alteration.

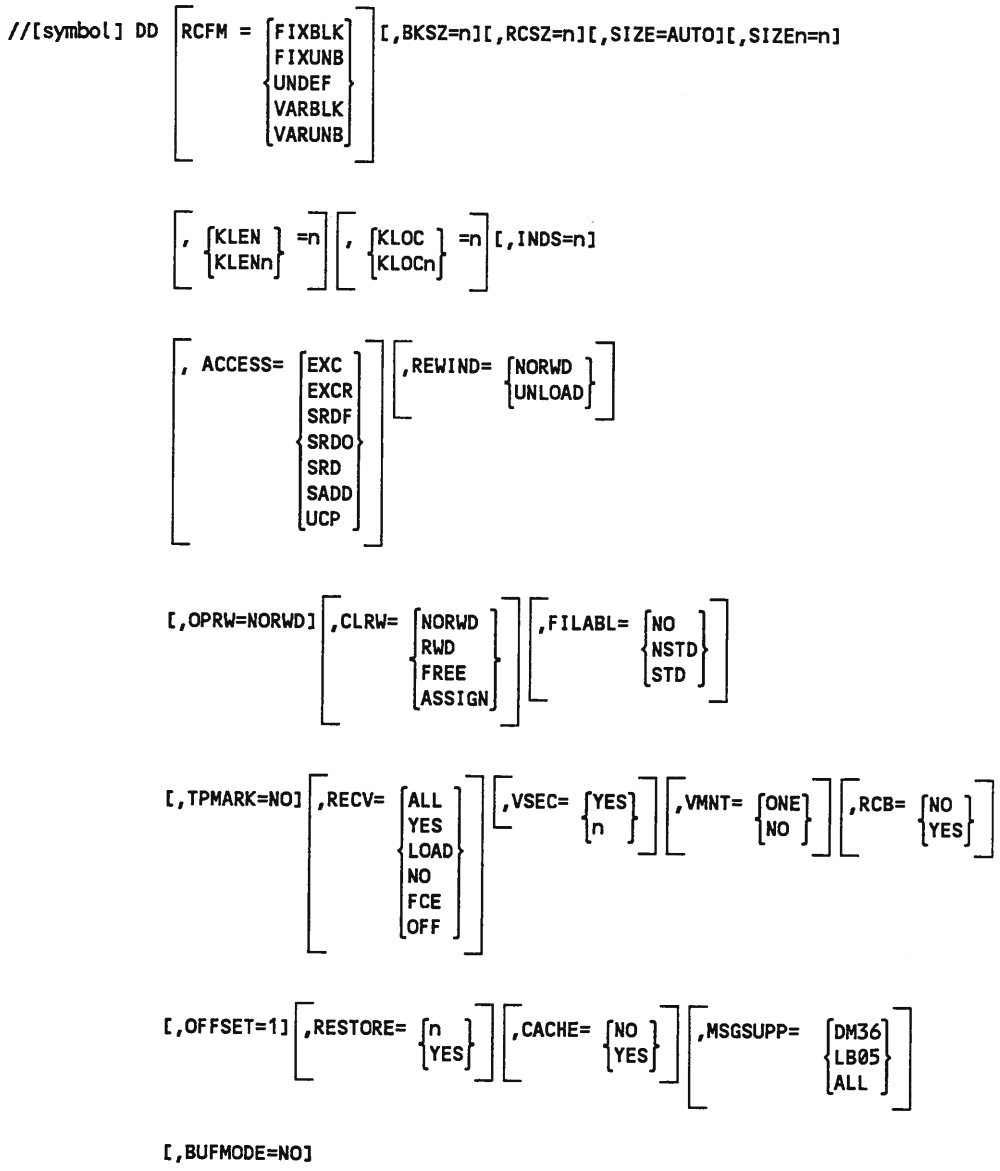
Suppose there is a phase named TSTPGMOO and it constantly needs changes according to weather conditions. The first and last ALTER job control statements could be inserted as needed. In the preceding example, the information contained in addresses 4361 and 4700 is changed.

Changing Your File Definition at Run Time

You may need to change the file definition contained in one of your programs. Regardless of the type of program (COBOL, assembly, and so forth), you would have needed to either reassemble or recompile and relink your program with the updated file definition. Now, using the DD (data definition) job control statement, you can make this change at run time. The changes made using the DD statement are effective only during the execution of the job; if you want to make a permanent change, you must make it in your source program.

You can have only one DD statement in each DVC-LFD sequence, and it must be placed with the assignment set for that device.

The format of the DD statement is:



In the format, we see all the allowable keyword parameters. If a parameter is specified but not allowed, it is ignored. The *n* following the KLEN and KLOC keywords refers to KEYn of a multikey MIRAM disk file. The *n* following the SIZE keyword refers to the partition identifier of a MIRAM disk file types. In Table 6-1, we equate the keyword parameters with their associated file types. For a complete description of all parameters, see the *Consolidated Data Management Programming Guide*, UP-9978. Descriptions of the parameters that are associated with SAT files are found in the *Supervisor Macroinstructions Programming Reference Manual*, UP-8832.

Table 6-1. DD Supported Keywords

Keyword	Format Label Diskette/Disk	Data Set Label Diskette	Tape	Card	Printer
RCFM*			X	X	X
BKSZ*	X	X	X	X	X
RCSZ*	X	X	X		
KLEN1-5*	X				
KLOC1-5*	X				
INDS*	X				
SIZE	X				
SIZE 1-2	X				
ACCESS	X				
VSEC	X				
RECV	X				
VMNT	X	X			
RCB	X	X			
OFFSET		X			
REWIND			X		
OPRW			X		
CLRW			X		
FILABL			X		
TPMARK			X		
RESTORE	X				
CACHE	X				
MSGSUPP	X				
BUFMODE			X		

* Take care when specifying this keyword parameter. If the program accessing the file is dependent on a predefined (e.g., compile time) file or processing characteristics, it may not be prepared for such a change at execution time. You may obtain unexpected results unless the program is a user-written BAL program prepared for this type of specification change or if the user documentation for the product explicitly states that this specification can be changed at execution time.

Legend

X Allowed keyword

Suppose we want to change the following FD entry in a COBOL program:

```
1       8     12                                                    72
        FD    SAVEIT
              RECORDING CODE IS F
              LABEL RECORDS ARE OMITTED
              RECORD CONTAINS 133 CHARACTERS
              BLOCK CONTAINS 10 RECORDS
```

Our FD describes an output magnetic tape file. We want to change the record size from 133 characters to 120 characters. Our DD statement would be:

```
// DD RCSZ=120
```

When a file is cataloged, the DD information does not get cataloged. When you call the file using the catalog, if the DD information is required, you must specify the DD statement in your control stream following the LBL statement. For example, when you cataloged the file, the following assignment set was used.

```
// DVC 60
// VOL DISK01
// DD BKSZ=200
// LBL DISKMAST
// LFD DISKM
// CAT DISKM
```

Now, when you call the file using the catalog, and the DD information is required, you would use the following:

```
// LBL DISKMAST
// DD BKSZ = 200
```

When you use the DD statement with a cataloged file, it must appear following the LBL statement. Otherwise, it can appear anywhere in the DVC-LFD sequence.

Note: *The file cataloging facility is described in the* File Cataloging Technical Overview, 7004 4615.

Adding Cards to a Stored Control Stream

The CR job control statement is used in a stored control stream to indicate that other job control statements or embedded data (on cards, data-set label diskette, or input spool file) is to be accepted from the input device and temporarily inserted into a stored control stream. You indicate the type of input device in the RU command or the // RUN job control statement. The CR job control statement has no parameters, it's just specified as:

```
//[symbol] CR
```

Let's examine one application of the CR statement. Suppose you're constructing a job control stream to execute programs that use low volume card input in the form of embedded data. Assume that you also want to store the control stream in YJCS, but you know that the embedded data will have to be periodically changed. Because the embedded data is part of the control stream, you'll actually be changing the stream when the data is changed. This somewhat defeats the purpose of storing a control stream in the first place.

You could change the programs to accept the data as card files submitted from the card reader (the card files can be changed without disturbing the control stream). Another alternative is to place CR statements in the control stream. When the stored control stream is initiated (with an RU command or a // RUN statement), the run processor will expect to find data in the card reader when it encounters a CR statement. The following example illustrates this:

```
The stored stream is:

// JOB MYJOB
      .
      .
      .
// EXEC PROG1
// CR
      .
      .
      .
// EXEC PROG2
// CR
/&
// FIN
```

```
   In the card reader you've placed:

/$

 embedded data for PROG1

/*
// FIN

/$

 embedded data for PROG2
/*
// FIN (This last FIN statement
        is unnecessary if the
        input is on data-set-label
        diskette or in the spool
        file.)
```

When the first CR statement is encountered, control is directed to the card reader where you've placed the embedded data for PROG1 between the /$ and /* statements. The first FIN statement ends card reader operations and control is returned to the stored stream until the next CR is encountered. Then the embedded data for PROG2 is accepted. Using this method you can place different data in the card reader for each job run if necessary.

Note: *This application of the CR statement cannot be used with saved translated control streams. Embedded data already included in such streams may, however, be replaced using the DATA STEP statement as described in "Replacing Embedded Data Sets in Expanded Control Streams" later in this section.*

As you'll see when we talk about bypassing job control statement, // CR is also used when you want other job control statements temporarily inserted in the stored stream.

Depending upon your application, a CR statement can be placed anywhere in the control stream. If, however, it is placed between a /$ and /* in the stored stream (e.g., for inserting job control statements within embedded data), you must include an OPTION SCAN statement in you control stream. For example:

```
// JOB MYJOB
// OPTION SCAN
      .
      .
      .
// EXEC PROG1
/$
embedded data
// CR
embedded data
/*
/$
// FIN
```

If the OPTION SCAN statement is omitted, the CR statement is ignored.

Note: *Filed control streams should be limited to control information or other low-volume data sequences that remain relatively constant. These control streams and any constant data (not entered from the input reader on each run) are considered permanent and occupy space otherwise available to the system.*

Bypassing Job Control Statements

You use the SKIP job control statement to skip forward in the control stream to another job control statement. SKIP is effective during execution of your program. Here's where the *label* field is used. Put a symbol in this field of the job control statement that's the target of the branch and specify this symbol in the SKIP job control statement. The skip can be conditional or unconditional, depending on the parameters you use.

Note: *Although both are used to bypass job control statements, / / SKIP, which is effective at execution time, must not be confused with / / GO, which is effective at run processor time. See "Unconditional Branching" in Section 7 for an explanation of / / GO.*

Neither the SKIP job control statement nor the target job control statement can be within a device assignment set or embedded data. All the devices assigned within a skipped section are still required before the job can be scheduled; however, skipped devices can't be referenced subsequently in the same control stream because (even though they are available) they aren't completely identified to the system. In view of this, you cannot bypass device assignment sets referenced subsequently in the control stream by REN or SCR job control statements. If you use SKIP to bypass the device assignment set for a cataloged file, you must specify a complete device assignment set for the file, not just the // LBL statement, and skip to a target label beyond the device assignment set for the cataloged file. File cataloging is explained in *File Cataloging Technical Overview*, 7004 4615. The skip function ends following the completion of the advance or upon the detection of a /& job control statement, whichever occurs first.

The format of the SKIP job control statement is:

```
//[symbol] SKIP target-label [,mask [, {ALL}]]
                               [       {ANY}]
                               [       {NONE}]
```

The *target-label* parameter corresponds to the symbol in the label field of the job control statement that's to receive the branch.

The *mask* parameter tests the UPSI byte and makes the SKIP job control statement conditional. (See "Setting the UPSI" earlier in this section.) It's one to eight characters long, and each character is a binary digit that corresponds to the bits of the UPSI byte. The allowable characters are 0 and 1; 0 means not set, and 1 means set. If you use fewer than eight characters, the unspecified rightmost positions are assumed to be zero. If you omit the *mask* parameter, the skip is unconditional.

The *ALL*, *ANY*, and *NONE* parameters are used with the *MASK* parameter to establish the criteria for satisfying the skip condition. For example, *ALL* states that all the UPSI bits indicated by the *mask* must be set to satisfy the skip condition. Only then will the skip be processed. Otherwise, it is ignored and processing continues with the next job control statement in the control stream. The same applies to the *ANY* and *NONE* parameters. If you do not specify one of these conditional parameters, *ANY* is assumed by default.

Let's set up a hypothetical situation. Suppose there's a program like the one described under the SET UPSI job control statement. (See "Setting the UPSI" in this section.) The program accepts input either in the form of cards or tape. In this case, bit 1 set means card input, no bits set means tape input. It edits details for an accounts receivable application and is run many times daily. So, you want to store this control stream in YJCS, rather than have it input through the card reader each time it's run - but then there would be two different device assignment sets for one input file (card or tape). Using the SKIP and SET UPSI job control statements, you could set and test the UPSI byte to see which device assignment set is needed and skip over the unwanted device assignment set. You could code the control stream to be stored as follows:

```
    // JOB BALANCE
    // CR
1.  // SKIP CARD,1
    // DVC 90
    // VOL MAST01
    // LBL DETAILS
    // LFD TAPEIN
2.  // SKIP DOIT
3.  //CARD DVC 30
    // LFD CARDIN
4.  //DOIT EXEC EDIT
    /&
    // FIN
```

and then precede the data cards to be processed with a SET UPSI job control statement that would identify the type of input device required.

In the sample control stream, parameter 1 in the first SKIP job control statement specifies that if the first bit of the UPSI byte is set to 1 (on), go to the job control statement with a symbol of CARD (3). This provides the device assignment set for the card reader, and the device assignment set for a tape is bypassed. If this bit is off, the device assignment set for a tape is processed until the second SKIP job control statement is reached. This causes an unconditional branch to the job control statement with a symbol of DOIT (which is the EXEC job control statement) and bypasses the device assignment set for the card reader.

Now, let's use input. Assume it's in the form of a card file. Look back at the example of the stored control stream. When it's read, the first CR job control statement switches control to the card reader, where we place a SET UPSI job control statement to turn the UPSI byte to on (which indicates card input). It's followed by a FIN job control statement, which terminates the card reader operation - control returns to the stored control stream. Since the UPSI byte is set to on, the tape device assignment set is bypassed, and the card reader device assignment set is used. The load module is then called. Here's what the stream to set the UPSI byte and provide the card input would look like.

```
// SET UPSI,1   ]
// FIN          }    Control statements inserted in the stored
   data cards   ]    stream when // CR is encountered
/*              }    Input card file
```

If the input were on tape, you would place a single FIN job control statement in the card reader. When the first CR job control statement transfers control to the card reader, FIN job control statement transfers it right back. Since the UPSI byte is not set, the device assignment set for tape is used, and the device assignment set for the card reader is bypassed.

Several system programs, such as the assembler, dump/restore, and disk prep, set the UPSI byte when an error occurs. For example, when an error occurs during a disk prep, the prep routine, by its nature, will continue to normal termination. If the error is fatal, you wouldn't want to run any subsequent job steps in the job, as they in turn would also be in error; you'd want to continue processing. The UPSI byte is automatically set on error conditions, and you can test it with the SKIP job control statement. The system programs use the following conventions when errors occur:

- A binary 1000 0000 (X'80') represents a fatal error. If this occurs, you would not want to run the remaining job steps. This can also be specified as a binary 1.

- A binary 0100 0000 (X'40') represents a warning error condition, which means that subsequent job steps can be processed. (However, it's up to you to determine whether the job should be rerun for total accuracy.)

The following two examples show how you can use the SKIP job control statement to check for errors in the system programs. (We're using the disk prep routine, whose control statements are explained in the *System Service Programs (SSP) Operating Guide*, UP-8841.

Example 1

```
1.  // JOB DSKPRP
2.  // DVC 20  // LFD PRNTR
3.  // DVC 50  // VOL DSP028  // LFD DISKIN
4.  // EXEC DSKPRP
5.  /$
6.     SERNR=DSP028,PARTL=V
7.  /*
8.  // SKIP ENDS,1
9.     .           (other
10.    .           job steps
11.    .           go here)
12. //ENDS NOP
13. /$
14. // FIN
```

In example 1, you check the UPSI byte to see if a fatal error has occurred. If the UPSI byte contains bit 1 set (line 8), then all the other job steps are bypassed and control is transferred to the NOP job control statement with the label ENDS (line 12). The NOP job control statement provides you with an address for the skip, with no function being performed. The /& job control statement terminates your job while the FIN job control statement terminates the card reader operation.

Example 2

```
1.  // JOB DSKPRP
2.  // DVC 20  // LFD PRNTR
3.  // DVC 50  // VOL DSP028  // LFD DISKIN
4.  // EXEC DSKPRP
5.  /$
6.     SERNR=DSP028,PARTL=V
7.  /*
8.  // SKIP WARN,01
9.  // SKIP FATAL,10
10. // SKIP EXIT
11. //WARN OPR 'WARNING-A NON-FATAL ERROR HAS OCCURRED'
12. //  SKIP EXIT
13. //FATAL OPR 'FATAL ERROR-JOB TERMINATED-CORRECT AND RERUN'
14. // SKIP ENDOFJOB
15. //EXIT NOP
16.   .           (other
17.   .           job steps
18.   .           go here)
19. //ENDOFJOBNOP
20. /&
21. // FIN
```

In example 2, you check for both the fatal and warning errors and the display of appropriate messages on the system console. If a warning error has occurred, that is, bit 2 set in the UPSI byte (line 8), then you skip to the label WARN on the OPR job control statement. The SKIP job control statement (line 12) is the next job control statement processed. Here, you skip down to the label EXIT on the NOP job control statement (line 15). As mentioned earlier, the NOP acts as an ending point for the SKIP job control statement. The remaining job steps follow the NOP statement and are processed accordingly. Following the last job step, the NOP statement on line 19 is processed, with no action being performed. Your job then terminates normally through the /& and FIN job control statements.

If a fatal error occurs, which is bit 1 set in the UPSI byte (line 9), you skip down to the label FATAL on the OPR statement (line 13) and print the specified message. The SKIP job control statement (line 14) skips down to the label ENDOFJOB on the NOP statement, thus bypassing your remaining job steps and terminating your job.

Bypassing Job Control Statements to Avoid Abnormal Termination

The ABNORM=label keyword parameter of the EXEC statement is used to skip forward in the job control stream if your program contains errors that will cause an abnormal termination. Recall that the format for the EXEC statement is:

```
// [symbol] EXEC program-name[, {[library-name]}][,[±]switch-priority][,ABNORM=label]
                              {$Y$RUN          }
                              {$Y$LOD          }
```

The label that you specify with the *ABNORM* parameter corresponds to the symbol (in the label field) of the job control statement that is the target of the skip. Since *ABNORM* is a keyword parameter rather than a positional parameter it may be coded in any position. For example:

```
// EXEC MYPROG,ABNORM=ERR
      or
// EXEC MYPROG,MYLIB,ABNORM=ERR
```

Now consider the following job control stream:

```
// JOB MYJOB
// DVC 20
// EXEC MYPROG,ABNORM=ERR
// OPR 'MYPROG TERMINATED NORMALLY'
// SKIP EOJ
// ERR OPR 'MYPROG TERMINATED ABNORMALLY'
//EOJ NOP
/&
```

Should MYPROG contain errors that will cause abnormal termination, the *ABNORM* parameter in this example specifies a skip to the job control statement with the label ERR. In this case, the message MYPROG TERMINATED ABNORMALLY will be displayed on the system console. If MYPROG terminates normally, this skip will not occur. Instead, the console message MYPROG TERMINATED NORMALLY will be displayed.

Remember, if the operator issues a cancel instruction for your job, the job still terminates normally, even though you've specified the ABNORM= parameter.

Dynamic Skip Function from a Workstation

The interactive user can change control stream execution from the workstation by dynamically skipping parts of the control stream. This is accomplished through the OPTION QUERY job control statement. (See "Selecting Optional Features" earlier in this section.) When a control stream containing the OPTION QUERY job control statement is processed, a message is displayed at the workstation screen asking you to indicate the type of skip function you want.

Substituting Embedded Data

Data can be embedded within a stored control stream, but there may be times when not all of this is used. For example, you may have a payroll application using a file with the names and pay rates of all the employees. The first quarter of the file may consist of salaried employees, and the remainder is the hourly employees. This job is run every week, but the salaried employees only get paid every two weeks, so you don't need to use their portion of the file on every run.

You can place job control statements within the embedded data to control this. By using the SCAN parameter in the OPTION job control statement, the embedded data is scanned to detect and act upon the job control statements embedded in the data. Thus, the data you do not want is skipped. If the OPTION job control statement is omitted, the job control statements are passed over without action.

The following rules are used by the run processor, and must be followed when placing job control statements in embedded data:

- There can be only one job control statement per card.
- Job control statements cannot be on the same card as data.
- The job control statement must be the target of an IF or GO job control statement.

When scanning embedded data for job control statements, two situations exist:

1. Embedded data is scanned when the OPTION job control statement is not present in the following manner:
    - Data is divided into sets - a particular /* job control statement is paired with its corresponding /$ job control statement in order to determine the true end of embedded data. The number of /* and /$ job control statements must be equal.
    - The FIN job control statement and the END proc definition statement are acted upon when detected.

2. If the SCAN parameter of the OPTION job control statement is used, the following job control statements are also acted upon:

   CR

   GBL

   GO

   IF

   JSET

   NOP

   OPTION

We'll discuss replacing embedded data sets in a saved, translated job control stream next.

Replacing Embedded Data Sets in Expanded Control Streams

Embedded data in a saved translated control stream can be replaced for only one run of the job. The replacement data must be preceded by a // DATA STEP statement and submitted from a card reader, data-set-label diskette, or an input spool file. The format of the DATA STEP statement is:

```
// DATA STEP=nnn
```

The *nnn* parameter is a decimal number in the range 1-255 that specifies the number of the job step within the job for which you're submitting new embedded data. Step 1, for example, is specified like this:

```
// DATA STEP=1
```

The DATA STEP statement is followed by a PARAM statement (if needed), the start-of-embedded-data statement (/$), the new data set, and the end-of-embedded-data statement (/*). If the job step specified in // DATA STEP has more than one data set, you must replace the old data sets in the job step with an equal number of new data sets. If you don't, an error occurs and the function is not performed. For example, let's say you want to replace the embedded data sets (two of them) in job step 3 of your job with new data sets. You would prepare these statements:

```
// DATA STEP=3
/$
   new embedded data
/*
/$
   new embedded data
/*
```

A DATA STEP statement must be submitted for each job step that contains embedded data you want to replace. If your job has four job steps, for example, and you want to replace the embedded data sets in steps 2 and 4 with new data, you would prepare these statements. For this example, assume step 2 has one data set and step 4 has two data sets:

```
// DATA STEP=2
/$
   new embedded data
/*
// DATA STEP=4
/$
   new embedded data
/*
/$
   new embedded data
/*
```

Since the DATA STEP sequence of statements (including the new embedded data) are submitted to the saved, translated stream from a card reader, diskette, or spool file, you must use the SI command or the // CC SI job control statement to initiate the running of the saved, translated stream.

The data sets you submit through the DATA STEP statement last for the duration of the run only because the copy of the job's YRUN file stored in YSAVE contains a copy of the original embedded data. To permanently change a saved, translated stream, submit a new stream to be translated and saved.

Note: *You can also use the / / DATA STEP statement to null existing embedded data by not including any new data between the start-of-embedded-data statement (/$) and the end-of-embedded-data statement (/*). However, bit pointers must be set in the original job control stream when attempting this operation. To null embedded data in a saved translated control stream, your prepared statement appears as follows:*

```
// DATA STEP=nnn
/$
/*
.
.
.
```

Job Control Considerations for Screen Format Services, Menu Services, and Dialog Processing

If you are preparing a control stream for a job that uses screen format services, menu services, or dialog processing, you must include the USE statement in your workstation device assignment set. The USE statement has different formats depending on which of the three interactive components your job uses. Only one USE statement may be specified in each workstation device assignment set.

Notes:

1. *When menu processing is initiated from a BAL user program, the following statements are required for the workstation OPEN RIB:*

   WKFM=VARI

   WAIT=YES

   WORK=YES

   PMODE=WSAM

2. *Menus and screens may not be used together within a user program except when the screens are processed directly by the menu processor (i.e., via the SCREEN and DISPLAY menu function commands).*

The USE Statement for Screen Format Services

When your program needs to use screen format services from a workstation, the USE statement you specify takes this form:

```
//[symbol] USE SFS[, {[format-file-LFD-1]/[format-file-LFD-2]}]
                   {format-file-LFD                         }
                   {$Y$FMT                                  }

      [,initial-screen][, {nnn}]
                         {1  }

      [,screen-format-1=alias-1[,...,screen-format-12=alias-12]]
```

The *symbol* parameter is used as the target of a branching statement. It is one to six alphanumeric characters long, and the first character must be alphabetic.

In the first positional parameter, you can provide an LFD name for up to two screen format files. Any name you use must match an LFD name specified in a previously defined device assignment set for a screen format file. (Screen format files are always MIRAM files.) The *format-file-LFD* is one to eight alphanumeric characters long.

When coding this parameter, remember the following:

- If you omit a format-file-LFD name, it is assumed that all screen formats used reside in the system file YFMT.
- If you code */format-file-LFD-2* alone, YFMT is examined first, then the file indicated by *format-file-LFD-2*.
- If you code *format-file-LFD-1/* alone, the file indicated by *format-file-LFD-1* is examined first, then YFMT.
- If you code *format-file-LFD* alone, only the file indicated by *format-file-LFD* is examined.

The *initial-screen* parameter specifies the name of the first screen format to be used by the application program. It is one to eight alphanumeric characters in length. Use of this parameter depends on the program's language. For more information, see the *Screen Format Services Technical Overview*, UP-9977.

The *nnn* parameter specifies the number of screens to be resident in main storage at one time, in the range 1 to 255. The default value is 1.

The *screen-format=alias* parameter equates a screen format name specified in an application program (alias) to the actual screen format name generated by the screen format generator. A maximum of 12 alias name sets may be specified. The *screen-format* name and *alias* name may each be from one to eight alphanumeric characters in length.

The control stream for a job that uses screen format services could include these job control statements:

```
// JOB YOURJOB
      .
      .
      .
   // DVC 50
   // VOL ABC
   // LBL FRMTFILE                    {   Device assignment set for the screen
   // LFD FORMAT                      {   format file
         .
         .
         .
   // DVC 200
   // USE SFS,FORMAT                  {   Device assignment set for the
   // LFD WORKSTN                     {   workstation
         .
         .                           ->  Screen format file LFD name
         .
   // EXEC PRGRM2
   /&
```

When you run YOURJOB, PRGRM2 is executed. PRGRM2 contains an instruction to open WORKSTN, which opens the screen format file FORMAT.

For more information about screen formats, see the *Screen Format Services Technical Overview*, UP-9977.

The USE Statement for Menu Services

When your program needs to use menu services from a workstation, the USE statement you specify takes this form:

```
//[symbol] USE MENU [, {menu-file-LFD/$Y$FMT}]
                     {$Y$FMT/menu-file-LFD}
                     {$Y$FMT               }

      [,initial-menu][, {nnn}]
                        {1  }

      [,menu-format-1=alias-1[,...,menu-format-12=alias-12]]
```

The USE MENU statement is similar to the USE SFS statement except that the parameters refer to menu formats instead of screen formats.

The *symbol* parameter is used as the target of a branching statement. It is one to six alphanumeric characters long, and the first character must be alphabetic.

The first positional parameter provides an LFD name for up to two menu format files. Any name you use must match an LFD name specified in a previously defined device assignment set for a menu format file. (Menu format files are always MIRAM files.) The *menu-file-LFD* is one to eight alphanumeric characters long. When coding this parameter, remember the following:

- If you omit a format-file-LFD name, it is assumed that all menus used reside in the system file YFMT.
- When you code */menu-file-LFD*, YFMT is examined first, then the file indicated by *menu-file-LFD*.
- When you code *menu-file-LFD/*, the file indicated by *menu-file-LFD* is examined first, then YFMT.

The *initial-menu* parameter specifies the name of the first menu format to be used by the application program. It is one to eight alphanumeric characters in length.

The *nnn* parameter specifies the number of menus to be resident in main storage at one time, in the range 1 to 255. The default value is 1.

The *menu-format=alias* parameter equates a menu format name specified in an application program (alias) to the actual menu format name (given when the menu was created). A maximum of 12 alias name sets may be specified. The *menu-format* name and *alias* names may each be from one to eight alphanumeric characters in length.

The control stream for a job that uses menu format services could include these job control statements:

```
// JOB YOURJOB
      .
      .
      .
   // DVC 50                          ]
   // VOL ABC                         }    Device assignment set for the menu
   // LBL MENUFILE                    }    format file
   // LFD MENU1                       ]
         .
         .
         .
   // DVC 200                         ]
   // USE MENU,MENU1                  }    Device assignment set for the
   // LFD WORKSTN                     ]    workstation
         .
         .                                  Menu format file LFD name
         .
   // EXEC PRGRM1
   /&
```

When you run YOURJOB, PRGRM1 is executed. PRGRM1 contains an instruction to open WORKSTN, which opens the menu format file MENU1.

For more information about menu services, see the *Menu Services Technical Overview,* UP-9317.

The USE Statement for Dialog Processing

When your job needs the dialog processor to manage a dialog session at a workstation, the USE statement you specify takes this form:

```
//[symbol] USE DP,dialog-name[,printer-lfd][,new-audit-lfd][,old-audit-lfd]
```

The files specified in the USE DP statement must have been previously identified (through device assignment sets) in the control stream.

The *symbol* parameter is used as the target of a branching statement. It is one to six alphanumeric characters in length, with the first character being alphabetic.

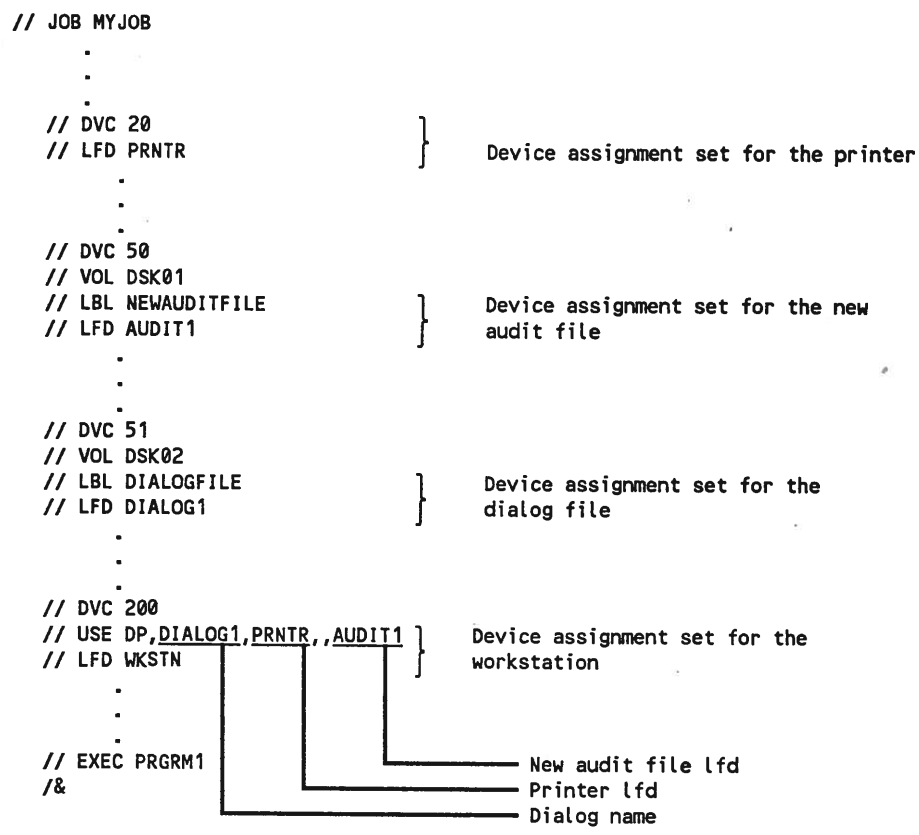
The *dialog-name* parameter specifies the name of the dialog you want to use; it must match the LFD statement of the dialog file's device assignment set. It is one to eight alphanumeric characters in length.

The *printer-lfd* parameter specifies the name of the printer file. It must match the LFD statement of the printer's device assignment set. It is one to eight alphanumeric characters in length. This parameter is specified when you want to produce a printed summary of the dialog session.

The *new-audit-lfd* parameter specifies the name of the new audit file output by the audit version of the dialog processor. It must match the LFD statement of the new audit file's device assignment set. The new audit file contains a record of your responses to a current dialog session. This parameter is one to eight alphanumeric characters in length.

The *old-audit-lfd* parameter specifies the name of the old audit file used as input to the audit version of the dialog processor. It must match the LFD statement of the old audit file's device assignment set. It is one to eight alphanumeric characters in length. The old audit file contains a record of your responses to a previous dialog session.

The control stream for a job that calls the dialog processor could contain these job control statements:



When you run MYJOB, PRGRM1 is executed. PRGRM1 contains an instruction to open WKSTN, which, when processed, causes DIALOG1 to execute at the workstation. Your responses to DIALOG1 are routed back to PRGRM1.

For more information about dialog processing, see the *Dialog Processor Programming Guide*, UP-8858.

Source Module Access via the USE Statement

Your programs can write (create) or read a source module that you identify in the USE LIB job control statement. When included in the device assignment set for a library file, // USE LIB indicates that the file contains source modules and the specified module will be accessed by your program.

The format for // USE LIB is:

```
//[symbol] USE LIB,module-name[,TYPE=[$]]
                               [{M}]
                               [{O}]
                               [{L}]
```

The module name you specify can be from one to eight alphanumeric characters long and the first character must be alphabetic. The following job control stream indicates that PROG1 will access a source module named MODULE1.

```
// JOB READMOD
      .
      .
      .
// DVC 50
// VOL D1234
// LBL SRCLIB1
// USE LIB,MODULE1
// LFD SRCMOD
      .
      .
      .
// EXEC PROG1
/&
```

Note: *Access of a source module by your program is limited to either a sequential read or sequential write operation.*

Section 7
Run-Time Conditional and Set Symbol Job Control Statements

Run-Time Conditional Job Control Statements

GO, IF, and NOP are run-time conditional job control statements. They allow you to branch to other job control statements in the control stream. Unlike SKIP job control statements (effective during execution of your program), they are interpreted and acted upon while the run processor is scanning the control stream, and then stripped from the stream. Therefore, any devices and volumes specified on the bypassed job control statements need not be available. Only forward branches are allowed for run-time conditional statements. Because GO, IF, and NOP are processed only by the run processor and their actions are completed when the run processor has acted upon them, they are very useful when writing job control procedure (JPROC) definitions.

Unconditional Branching

The GO job control statement causes an unconditional branch to another job control statement identified by a symbol. The destination can be a set symbol with a value determined when the job stream is analyzed.

The format of the GO job control statement is:

```
//[symbol] GO destination
```

The *symbol* is only used when this job control statement is the target of another GO or IF job control statement.

The *destination* parameter identifies the target job control statement and must agree with the symbol in the *label* field of that statement.

Like the other run-time conditional statements, the GO job control statement is acted upon by the run processor, before a job is scheduled, and then deleted from the control stream. For this reason, the devices and volumes skipped by a GO statement need not be available when the run symbiont is scanning the control stream.

Note: *Unlike GO, SKIP is effective during the execution of a program. Because a job is not executed until all the devices and volumes it uses are available to the system, devices and volumes bypassed by SKIP must be available or the job won't be scheduled. However, devices and volumes bypassed by a SKIP statement can't be referenced in subsequent job control statements in the control stream because, even though they are available, they have not been completely identified to the system.*

The following is a stored control stream similar to the one shown with the SKIP job control statement. (See "Bypassing Job Control Statements" in Section 6.)

```
// JOB BALANCE
// CR
// DVC 90
// VOL MAST01
// LBL DETAILS
// LFD TAPEIN
// GO DOIT
//CARD DVC 30
// LFD CARDIN
//DOIT EXEC EDIT
/&
// FIN
```

If the input is on cards, you would place the following stream in the card reader:

```
// GO CARD       ]  Job control statements inserted in the stored stream
// FIN           ]  when // CR is encountered.
   data cards    ]
/*               ]  Input card file
```

When the first CR job control statement from the stored control stream is encountered by the run processor, it transfers control to the card reader, where the GO job control statement causes the device assignment set for the tape to be skipped without any processing. The tape volume and the device that would use it do not have to be available. Therefore, they can be used by another job. If the input is on tape, a FIN job control statement is all that's needed in the card reader. The tape device assignment set would be read, and the stored GO job control statement would cause the device assignment set for the card reader to be bypassed.

Conditional Branching

The IF job control statement causes a conditional branch to another job control statement, depending upon certain test conditions. This is similar to using the SKIP job control statement conditionally, except that it's interpreted and acted upon by the run processor, just like the GO job control statement.

The format of the IF job control statement is:

```
//[symbol] IF (a op b)destination
```

The *symbol* is only used when this job control statement is the target of another GO or IF job control statement.

The test for a conditional branch is specified as (a op b), where a and b are the two operands to be compared. You can compare two numeric operands (1 op 2) or two alphabetic operands (a op b) but a run processor error results if you attempt a comparison between one numeric and one alphabetic operand (1 op b).

The *op* in the expression is the relational operator that specifies the type of comparison to be done. The values for *op* are:

EQ - *a* is equal to *b*

NE - *a* is not equal to *b*

GT - *a* is greater than *b*

LT - *a* is less than *b*

GE - *a* is greater than or equal to *b*

LE - *a* is less than or equal to *b*

Remember, whenever you enclose an operand in quotes, the quotes are considered a part of the operand. For example, ('a' EQ a) is an allowable comparison but the operands are not equal because one value is *'a'* and the other is *a*. (See "Specifying Set Symbol Values in Quotes" later in this section for more information.)

The operands are separated from the relational operator by spaces and the entire parameter is enclosed within parentheses.

Note: *If a numeric comparison is made and neither a nor b is numeric, both the greater than and less than conditions are set, resulting in all conditions except equal being allowed to branch. If a character compare is being used and the two operands are not of the same length, then the comparison is made on the number of characters present in each, rather than on the contents of the operands. Thus, a string of five characters will always be less than a string of six characters, regardless of the character content of the comparands. If you have specified // OPTION UNEQUAL, an error message is generated whenever character strings of unequal length are compared. (See "Selecting Optional Features" in Section 6.)*

The *destination* parameter identifies the target job control statement that will receive control if the transfer condition is true. This entry must agree with the *symbol* in the label field of the target job control statement.

When scanning for the target job control statement, only the FIN job control statement is acted upon. Therefore, you cannot branch out of the current job stream; any procedure calls or CR job control statements that are skipped are not acted upon.

The comparand fields may be variable symbols, or dummy arguments, that can be set in a JPROC definition. They're called dummy arguments because the variable symbol can be modified when called by the JPROC call.

Let's look at an example. At first, this example will not be totally clear, but when combined with the explanations of the remaining job control statement in this section and the JPROC definitions in the next section, it will become clearer. The only purpose here is to explain how the IF job control statement functions.

Consider this example:

```
// IF ('&IN' EQ 'N')EXIT
```

This job control statement is in a JPROC definition. When the PROC directive was written, it contained a parameter called *IN*. The ampersand of *&IN* identifies this as a variable symbol; this means, use the value of the *IN* parameter. *EQ* is the relational operator. *N* is a value that can be supplied as a value for *IN*. Thus, if the value specified by the *IN* parameter is equal to *N*, transfer control to the destination supplied by the next parameter, which is *EXIT*. If *IN* is not equal to *N*, control is transferred to the job control statement immediately following the IF job control statement. Note in the example that spaces precede and follow both IF and the operator EQ. Note also the *lack* of spaces between the parentheses and the '&IN' and 'N' terms and the lack of spaces or a comma before the word EXIT.

Providing Targets for Branching

The *symbols* in the label field of the job control statements provide the targets for branching job control statements. But the /$, /*, and /& job control statements don't have a label field. You may also want to branch to the end of a JPROC, which is an END directive. This doesn't have a label field that can be accessed by a branching job control statement. The NOP job control statement allows you to branch to an otherwise unaccessible position in the control stream.

The format of the NOP job control statement is:

```
//symbol NOP [QUERY]
```

This job control statement provides a target for a branching job control statement. The *symbol* must agree with the target defined in the sending job control statement. The optional QUERY parameter is used when you want to take advantage of the label-skipping facility of // OPTION QUERY. This facility is available to workstation users and console operators.

The following is an example based on the IF job control statement example shown earlier in "Conditional Branching" and using the END directive as the target (it's still within a JPROC definition):

```
// EXEC LISTX
// IF ('&IN' EQ 'N')EXIT
// PARAM SPACE=TWO
//EXIT NOP
     END
```

Notice that the IF job control statement was placed afer the EXEC job control statement. This is allowable since it's a run-time conditional job control statement, which is acted upon by the run processor and then stripped from the control stream.

Note: *You can use the NOP statement to place comments in your control stream. The comment is used in place of the QUERY parameter, is separated from the NOP statement by one or more blanks, and is enclosed in single quotes. When used for this purpose, the NOP statement does not have to be the target of a branching statement.*

Run-Time Set Symbols

A set symbol is a type of variable that can be set to a value and used by the run processor as a counter, switch, or value to control a job. Because the run processor is responsible for making set symbols effective, they are called run-time set symbols. There are two types of set symbols:

- GLOBAL

  A global set symbol, once declared, can be referenced anywhere in the basic control stream as well as in any JPROC definition the control stream calls.

- LOCAL

  A local set symbol can only be declared and referenced within a JPROC definition. (If a local and a global set symbol have the same name, the local symbol is used within the JPROC.)

You use the following to declare run-time set symbols.

- // GBL, // QGBL, RUN/RV (command), // RUN/RV

  Declare global set symbols only.

- // JSET

  Declares local set symbols and (if specified in a basic control stream after // GBL or // QGBL) can be used to supply or change the value of a global set symbol (without changing the symbol's status to local).

Global Status Set Symbols

The GBL job control statement can be used to declare global set symbols. This statement may appear anywhere in the control stream, and the symbols are global from the point of declaration forward.

The format of the GBL job control statement is:

```
//[symbol]GBL set-id-1[=init-1][,set-id-2[=init-2],...,set-id-n[=init-n]]
```

The *set-id* parameter specifies the name of the set symbol. The *init* parameter assigns a value to the set symbol, provided a value has not already been assigned. For example:

```
// JOB MYJOB
      .
      .
      .
// GBL PRNTR=20
```

The set symbol defined in the preceding // GBL statement is PRNTR and the value of PRNTR (&PRNTR) is 20. The value 20 is substituted any time you reference this symbol by &PRNTR later in the control stream, or in any JPROC the control stream calls. For example:

```
// JOB MYJOB
      .
      .
      .
// GBL PRNTR=20
      .
      .
      .                    } The value 20 is substituted for &PRNTR when the
// DVC &PRNTR=20           } run processor encounters this statement. The
// LFD PRTFIL              } result is // DVC 20.
      .
      .
      .
/&
```

Note: *The & used when referencing a set symbol is never used when defining the set symbol in the GBL job control statement.*

The value assigned by the *init* (value) parameter is used only if a value is not assigned by a preceding // GBL statement; is not assigned in the RUN/RV command (the RUN/RV job control statement if you're initiating one job from another); is not changed later in the control stream by a // JSET statement. Consider the following:

```
// JOB MYJOB
      .
      .
      .
// GBL PRNTR=20
      .
      .
      .
// GBL PRNTR=26
      .
      .
      .
```

The value assigned by the first GBL job control statement applies for the entire control stream any time &PRNTR is referenced. The second GBL job control statement does not result in an error condition but has no effect on the value of PRNTR. (You can use a JSET job control statement in place of the second GBL job control statement to change the value of PRNTR. JSET is discussed in "Local Status Set Symbols" later in this section.)

The effect of specifying a global set symbol and value in the RUN/RV command is as if // GBL is inserted directly after the // JOB statement in the control stream. If, for example, you use RV MYJOB,,PRNTR=28 to initiate a job, // GBL PRNTR=28 is considered the first statement in the stream. You can reference &PRNTR any place in the job stream and the run processor will substitute the value 28. Consider the following:

```
// JOB MYJOB
      .
      .                The global set symbol PRNTR was defined and given
      .              } a value of 28 in the RUN/RV command.  The run
// DVC &PRNTR         } processor substitutes 28 for &PRNTR resulting in
// LFD PRTFIL           // DVC 28.
   .
   .
   .
```

Note: *Remember to include a // OPTION SUB statement in your control stream if you want values substituted for set symbols referenced in embedded data.*

If you include a // GBL statement for PRNTR in your control stream specifying one value, and initiate that stream with a RUN/RV command specifying another value for the same symbol, the value specified on the RUN/RV command is used. If, for example, you use RV MYJOB,,PRNTR=28 to initiate the following stream:

```
// JOB MYJOB
      .
      .
      .
// GBL PRNTR=20
      .
      .
      .
// DVC &PRNTR
// LFD PRTFIL
      .
      .
      .
```

the value 28 is substituted for &PRNTR. The value 20 is used only if you don't supply a value for PRNTR in the RUN/RV command.

Whenever you specify a set symbol in the // GBL statement without a value (for example, // GBL PRNTR), you must use the RUN/RV command to supply the value, or provide a value using the // JSET statement before the symbol is referenced. Otherwise, the value of the symbol is considered null. This may or may not be desired. Consider the following GBL job control statement:

```
// GBL PRNTR,TOKEN=DKIN
```

This statement declares global status for the set symbols PRNTR and TOKEN. The value of TOKEN is DKIN. The value of PRNTR was previously defined in the RUN/RV command, will be defined later in a JSET job control statement before &PRNTR is referenced, or is a null value.

When coding the GBL job control statement, you cannot use the statement continuation; specify separate // GBL statements.

With the QGBL job control statement, interactive users can declare global set symbols in a job control stream and then specify values for those symbols through the workstation at job run time. The format of the QGBL job control statement is:

```
//[symbol] QGBL set-id-1[=init-1][,set-id-2[=init-2],...,set-id-n[=init-n]]
```

The *set-id* parameter may be a maximum of eight characters and the *init* (value) parameter may be a maximum of 60 characters. When you run a control stream containing a // QGBL statement, the specified set symbol is displayed at the workstation and you're asked to provide a value for the set symbol. A null response may indicate that a (default) value specified in the QGBL statement is valid. Suppose you build a job control stream that includes these statements:

```
// JOB MYJOB
// QGBL DVC=20
// DVC &DVC
// LFD PRNTR
      .
      .
      .
/&
```

When you initiate the control stream (RV MYJOB) and the run processor encounters the // QGBL statement, the following is displayed on the workstation screen:

```
03 ? JOB=MYJOB        SYMBOL=DVC        VALUE=20        *ENTER VALUE
```

If you don't enter a value on the following line (for example, 03 22, indicating a specific printer), the value specified in the // QGBL statement (20) is substituted for &DVC.

Suppose you write the following job control stream to prep a data-set-label diskette:

```
// JOB PREPDSL
// QGBL ADDR,VSN,RCSZ,SPIRL,IPL
// DVC 20
// LFD PRNTR
// DVC 130,&ADDR
// VOL X(NOV)
// LFD DISKIN
// OPTION SCAN,SUB
// EXEC DSKPRP
/$
   SERNR=&VSN,RECSZ=&RCSZ,SPIRL=&SPIRL,IPLDK=&IPL
   VOL1
/*
/$
```

The // QGBL statement declares five global set symbols. One is referenced in the DVC statement for the diskette device. The other four are referenced in the embedded data. (The embedded data consists of keyword parameters whose values provide necessary information for the diskette to be prepped.) When you initiate the job at your workstation using RV PREPDSL, and the run processor begins job processing, the following occurs:

- A workstation screen display asks you to supply values for each of the set symbols declared by the // QGBL statement. For example:

  ```
  JOB=PREPDSL    SYMBOL=ADDR    VALUE IS NULL    *ENTER VALUE
  ```

  (Assume that 320, DK001, 128, Y, and Y are the values you specify for ADDR, VSN, RCSZ, SPIRL, and IPL, respectively.)

- When DVC 130,&ADDR is encountered, the run processor substitutes the value 320 resulting in // DVC 130,320. (Had a null response been entered, then a physical device would not be assigned.)

- When the embedded data is encountered, the run processor substitutes the specified values (provided, of course, you included a // OPTION SUB statement in the job control stream) resulting in

  ```
  SERNR=DK001,  RECSZ=128,  SPIRL=Y,  and  IPLDK=Y.
  ```

For information about prepping diskettes, see the *Systems Service Programs (SSP) Operating Guide*, UP-8841.

If global symbols declared by // QGBL are given values through any other means (a RUN/RV command, a // GBL statement in the control stream, a // JSET statement in the control stream), you won't be asked to submit a value at the workstation even though the stream includes a // QGBL statement.

Local Status Set Symbols

The JSET job control statement can be used to define a local set symbol or to change the value of a global set symbol without changing its status to local.

The format of the JSET job control statement is:

```
//symbol JSET value
```

The *symbol* specifies the name of the set symbol. The value of this set symbol is coded as the *value* parameter. It may be a character string up to eight characters long enclosed by apostrophes, if it contains blanks. For example:

```
// PRNTR JSET 20          (The symbol PRNTR is given a value of 20.)
```

Now, consider the following:

```
//PRNTR JSET &DEVICE
```

In this statement, PRNTR will have whatever value is given to DEVICE. The value for DEVICE can be supplied via the RUN/RV command, in a preceding // GBL or // QGBL statement, in a JPROC call, or even in // JSET statement specified later in the control stream (for example, //DEVICE JSET 20).

The value can also be a simple 2-term expression such as &A+&B. The operations allowed in a 2-term expression are:

Operator	Description
//	Covered quotient, A//B is equivalent to (A+B-1)/B.
/	A/B means arithmetic quotient of A and B.
*	A*B means arithmetic product of A and B.
-	A-B means arithmetic difference of A and B.
+	A+B means arithmetic sum of A and B.
**	A**B means logical product AND of A and B.
++	A++B means logical sum OR of A and B.
--	A--B means logical difference XOR of A and B.

Whenever you're performing an operation using a JSET statement, the operands upon which the operation is to act must be numeric. Look at this example:

```
// GBL M=1,X=2
//MX JSET &M+&X
```

The result of this operation is MX=3.

If both the operands are not numeric, the operation is not performed and the result is a concatenation of the values. If M had been set with the value of A in the preceding example, the result would have been MX=A+2. The operation would not have been performed.

You can also use the JSET control statement to establish a null value. This can be done by specifying either:

```
//symbol JSET
```

or

```
//symbol JSET ''
```

Leading zeros are not maintained for multiple-digit numeric values in a JSET control statement. If a leading zero is required when the symbol is used, it must be created via a second JSET control statement. For example, if you want the value of symbol P to be 08, assign another symbol (K in this example) the value of 0, like this:

```
//K JSET 0
```

Assign symbol P the value of 8, like this:

```
//P JSET 8
```

When P is referenced, it must be prefixed by K. Thus, the value of &K&P is 08.

As mentioned earlier, when you define a set symbol in a JPROC using // JSET, the symbol is considered local and can only be referenced within the JPROC. JSET, however, also allows you to change the value of a global set symbol without changing its status to local. For example:

```
// JOB MYJOB
// GBL PRNTR=20
      .
      .
      .                 This statement changes &PRNTR to 28.
//PRNTR JSET 28         PRNTR is still a global set symbol.
      .
      .
      .
```

If you define a global set symbol in the RUN/RV command or a GBL statement and you don't specify a value (for example, RV MYJOB,,PRNTR or // GBL PRNTR) you can simply use // JSET to provide one or more values for PRNTR.

For example:

```
// JOB MYJOB
      .
      .                  PRNTR was defined in the RUN/RV command or a previous
      .                  // GBL statement.  // JSET assigns 28 for the value of
//PRNTR JSET 28           PRNTR.  Any time the run processor encounters &PRNTR,
      .                  28 is substituted until the next // JSET is encountered.
      .
      .
//PRNTR JSET 20           This statement changes &PRNTR to 20.  Any time the
      .                  run processor encounters &PRNTR, 20 is substituted
      .                  until the end of job.
      .
```

Specifying Set Symbol Values in Quotes

There are certain considerations you should take when assigning a value enclosed in quotes to a set symbol.

Whenever you use // GBL or // QGBL to assign a quoted value to a set symbol, the quotes are always considered part of the value. For example:

```
// GBL X='ABC',Y=XYZ
```

The value of X (&X) in this case is 'ABC' while the value of Y (&Y) is XYZ. This is worth remembering especially if &X will be involved in a comparison using the IF job control statement. (See "Conditional Branching" earlier in this section.) If, for example, the value of X is set to 'ABC' as follows:

```
// GBL X='ABC'
```

the following statement represents a character comparison match:

```
// IF (&X EQ 'ABC')LABEL
```

This statement results in a branch to LABEL because the value of X is 'ABC' and the value you're comparing X to is 'ABC'. Consider the following statement:

```
// IF (&X EQ ABC)LABEL
```

This is not a character comparison match because the value of X is still 'ABC' while the value you're comparing X to is ABC.

A different situation exists when you use // JSET to assign a quoted value because // JSET always removes one level of quotes (if any). Consider the following:

```
//X JSET 'ABC'          The value of X (&X) is ABC.

//X JSET ''ABC''        The value of X (&X) is 'ABC'.

//X JSET ABC            The value of X (&X) is ABC.

//X JSET ''ABC''        The value of X (&X) is 'ABC'.

//X JSET &X             X is now ABC.
```

This should also be considered when specifying a comparison with // IF that involved a quoted value assigned by // JSET.

Using Symbols to Examine Job and System Related Values and Facilities

Through the use of symbols, the INQ job control statement allows you to examine job and system related values (such as jobname, system time, and system date) or to determine the availability of certain facilities (such as DDP and workstations).

The // INQ statement has two formats:

```
//symbol INQ JOB,keyword
//symbol INQ SYS,keyword
```

You use // INQ JOB to examine job related values and facilities and // INQ SYS to examine system related values and facilities. In both formats, *symbol* defines the variable symbol that is set to a value specified by *keyword*.

The keyword ORI, for example, sets the value of the symbol X in the following statement to the user-id of the job's originator (the workstation that initiated the job).

```
//X INQ JOB,ORI
```

If you refer to the value of X (&X) elsewhere in the job control stream, the user-id of the originator will be substituted for that value.

Consider the following:

```
// JOB MYJOB
      .
      .
      .
//X INQ JOB,ORI
// OPR 'DELIVER OUTPUT TO &X',OPERATOR
      .
      .
      .
/&
```

If USER01 initiates the job, the run processor substitutes USER01 for &X so that the operator receives the message 'DELIVER OUTPUT TO USER01'. If USER02 initiates the job, the operator receives the message 'DELIVER OUTPUT TO USER02'.

Suppose you want to execute a program that can receive input either from a workstation or diskette. If the job is initiated from a workstation, then workstation input is preferred. The // INQ JOB statement, used with the keyword WKS, allows you to determine whether the job is initiated from a workstation. This way you can configure a job control stream that assigns a diskette device or a workstation, depending on the situation.

The keyword WKS sets the value of the symbol X in the following statement to either 1 or 0:

```
// X INQ JOB,WKS
```

If the value of X is 0, it means that a workstation is not initiated for this job. If the value of X is 1, a workstation is initiated for the job. With this in mind, consider the following job stream:

```
// JOB MYJOB
      .
      .
      .
//X INQ JOB,WKS
// IF(&X EQ 0)DSKT
// DVC 200
// USE SFS
// LFD INFO
// GO NEXT
//DSKT DVC 130
// VOL A123
// LBL FILE1
// LFD INFO
//NEXT EXEC PROG1
      .
      .
      .
/&
```

This job stream is configured so that the device assignment set for the workstation is skipped if the job is not initiated from a workstation and the device assignment set for the diskette is skipped if the job is initiated from a workstation. Table 7-1 lists all of the keywords that you can use with // INQ JOB and // INQ SYS.

Table 7-1. Keywords and Symbol Values for // INQ JOB and // INQ SYS

	Keyword	Value of Symbol
For // INQ JOB	NAME	The job name
	ORI	The user-id of the originator
	HOST	The host-id of the originator (null if none)
	ORID	The device-id of the originator if a local workstation
	WKS	0 if job is not initiated from a workstation 1 if job is initiated from a workstation
	JBNO	A 4-byte job number
	DDP	0 if remote DDP is not initiated 1 if remote DDP is initiated
For // INQ SYS	RES	SYSRES volume serial number
	RUN	SYSRUN volume serial number
	DATE	The system date (yy/mm/dd)
	DAY	The day of week (Sunday = 1, Monday = 2, etc.)
	MM	The month (01 through 12)
	DD	The day of month (01 through 31)
	YY	The year (specify last two digits)
	TIME	The system time (hh.mm.ss.)
	HOST	The system's own host-id
	REL	The system release-id (vv.r.rrr)
	SUP	The supervisor's name
	DDP	0 if DDP is not available 1 if DDP is available

continued

Table 7-1. Keywords and Symbol Values for // INQ JOB and // INQ SYS (cont.)

	Keyword	Value of Symbol
For // INQ SYS	WKS	0 if workstation support is not configured 1 if workstation support is configured
	S80	07 if running on model 7E 08 if running on model 8 10 if running on model 10 15 if running on model 15 20 if running on model 20 50 if running on model 50
	SPL	0 if spooling is not configured 1 if spooling is configured
	JUL	Assigns Julian date YYDDD to specified symbol

Priorities among Set Symbols, Keyword Parameters, and Positional Parameters

External to a JPROC definition, the only possibility of substitution is for set symbols. Inside of a JPROC definition, however, the possibility of a set symbol matching a keyword parameter or positional parameter name does exist.

The positional parameter name is maintained as a separate entity. Global set symbols are maintained in a single list. Keyword parameter names and local set symbols are maintained together, with a new definition replacing the old. The effect of keyword parameter names and local set symbols being maintained together is to force keyword parameter names to local status if they are mentioned in JSET job control statements within the procedure.

When it's determined that substitution should be performed, the following steps occur, in the order given:

1. A comparison is made with the positional parameter name. This test is done first, since there is one name with many values, but it's a relatively fast test. Care must be taken to make the positional parameter name unique with respect to all set symbols and keyword parameter names. A sublisted reference to a keyword parameter cannot be distinguished from a reference to a positional parameter.

2. The list of local set symbols and keyword parameter names is scanned.

3. The list of global set symbols is scanned.

The result is that if a keyword parameter name matches a local or global set symbol within a procedure, the following occurs:

1. A reference to the name obtains the keyword parameter value up until the occurrence of a JSET job control statement for the name.

2. From the point of occurrence of the JSET job control statement to the end of the JPROC definition, the value of the most recent JSET job control statement is used.

3. At the end of the JPROC definition, the value reverts to the value of the global set symbol at the time of entering the procedure.

Note: *Remember that set symbol substitution may increase the number of characters in a value.*

Section 8
How to Write and Call a Job Control Procedure Definition

The Benefit of Procedure Definitions

Section 5 discussed the job control procedure (JPROC) call statements supplied by Unisys. In this section, we'll discuss how to write your own JPROC definitions and how to call them.

A JPROC definition is similar to an assembler procedure definition, which is explained in the *Assembler Programming Guide*, 7004 4532. However, the JPROC definition is a series of job control statements and procedure directives, as opposed to assembler instructions and directives. It consists of a PROC directive, one or more NAME directives, a series of job control statements, and an END directive.

The PROC directive signals the beginning of the procedure, the NAME directive declares a label by which the JPROC can be called, and the END directive signals the end of the JPROC. Each time the series of job control statements is needed, a JPROC call is used. Job control then inserts the necessary job control statements at the point where the JPROC call was placed. The JPROC definition defines the coding and job control statements needed for a particular operation, and the JPROC call specifies the values for the variable parameters of the JPROC definition.

Coding Rules

The directives used in writing JPROC definitions take this form:

```
LABEL       ΔOPERATIONΔ       OPERAND
```

The *label* field extends from column 1 to column 8. At least one space must separate the *label* field from the *operation* field, and also the *operation* field from the *operand* field. Column 72 is used to indicate continuation, and columns 73 through 80 can contain identification or sequence information.

Note: *For compatibility with job control statements, you can precede the label field with two slashes (/ /) in columns 1 and 2. In this case the label field extends from column 3 to column 10.*

The job control statements within a JPROC definition follow the same conventions as regular job control statements. These are listed in Appendix A.

The characters that are allowable in directives and job control statements are as follows:

Letters	A through Z
Special letters	? $ # @
Digits	0 through 9
Special characters	+ - * / , = ' blank ( ) . > < & ! : ;

The terms you can use in the *operand* field of a directive may be symbols or character strings, which are explained in the following paragraphs.

A symbol is a group of up to 240 alphanumeric characters used for parameter identification and as labels. The first character must be alphabetic. Special characters or blanks may not be contained within a symbol. The following are examples of valid symbols:

V CARDAREA

GS279 R$INTRN

DAVE

The $ of R$INTRN is allowable, because it's a special letter, not a special character.

For a symbol to be recognized by job control as a parameter identifier, it must be immediately preceded by an ampersand.

The following are not valid symbols:

READ ONE - embedded blank

SPEC'L - special character

8AGN - first character not alphabetic

The *operand* field in a NAME directive may be obtained by referencing the symbol *p(0),* where *p* is the symbol used to reference any positional parameter in the definition. The zero indicates the parameter of an operand field.

A character string can represent up to 252 valid characters, all of which must be printable. Character strings containing embedded blanks or commas must be enclosed in either quotation marks or parentheses. The enclosing quotation marks or parentheses are considered part of the character string. Embedded quotation marks are not allowed in the character string.

A null character string is represented by two consecutive quotation marks.

All parameter values are evaluated as character strings.

Parameter Types

Parameters are used to pass information from the JPROC call to the JPROC definition. These parameters can be equated to values, symbols, or character strings, and may be used to specify file identifiers, file names, volume serial numbers, etc.

There are two types of parameters: positional and keyword. Positional parameters are identified by their position within the *operand* field of the JPROC call; keyword parameters are identified by the symbols assigned to them in the JPROC directive. The rules for specifying positional and keyword parameters with respect to position, order, omission, and format are covered in Appendix A.

Both positional and keyword parameters may be sublisted. Thus, each operand of the JPROC call may represent one value or a series of values that may be referenced independently. When a parameter is sublisted, the subparameters must be separated by commas, and the entire list must be enclosed by parentheses.

For sublisted positional parameters, an operand would appear as:

```
(val-1,val-2,...,val-n)
```

For sublisted keyword parameters, an operand would appear as:

```
key=(val-1,val-2,...,val-n)
```

An omitted positional parameter in a JPROC call takes the value of a null character string. When a keyword parameter is given a value in the JPROC definition, it takes that value if the keyword parameter is omitted in the JPROC call. When no value is given to a keyword parameter in the JPROC definition, it takes the value of the null character string when omitted.

Now, let's explain the three JPROC directives.

The Start of the JPROC Definition

The PROC directive signals the start of a JPROC definition. It defines the number and type of parameters that may be specified in the JPROC call.

The format of the PROC directive is:

LABEL	ΔOPERATIONΔ	OPERAND
[[//]symbol]	PROC	[pos,n][,k,...,k]

The *symbol* is a dummy label of one to eight alphanumeric characters. It's used as an entry point to the JPROC definition when it's expanded and inserted into the control stream. If the JPROC call also has a *symbol,* it replaces the *symbol* of the PROC directive when the JPROC definition is called. If the JPROC call has no *symbol,* the dummy label is replaced by a null character string. The characters & . ( ) ' , + - / may not be embedded in the symbol.

The *pos* parameter represents the symbol by which any positional parameter in the body of the JPROC definition is referenced. If this parameter is omitted, no positional parameters can be used in the JPROC call. The *n* is a decimal number that represents the total number of positional parameters permitted in the JPROC call. If omitted, zero is assumed. If you omit the *pos* and *n* parameters in this directive (thereby indicating there are no positional parameters), you must still code two commas before you can code any keyword name values.

The *k* parameter represents the name or names used in referencing keyword parameters and their default values (if any).

To preset a keyword value, the *k* parameter takes the form:

```
[,k-1=value,...,k-n=value]
```

In the following example, MOD1 is the symbol used as an entry point. One positional parameter is allowed, and it's referenced by the symbol P in the JPROC definition. There are three keyword parameters allowed in the JPROC call; PRINTER, INPUT, and OUTPUT. If the PRINTER keyword parameter is omitted, it defaults to 20.

```
MOD1 PROC P,1,PRINTER=20,INPUT,OUTPUT
                    or
//MOD1 PROC P,1,PRINTER=20,INPUT,OUTPUT
```

Naming the JPROC Definition

The NAME directive supplies the name by which a JPROC definition is referenced. It must immediately follow the PROC directive. More than one NAME directive can be used, but all must be grouped at the beginning of the JPROC definition. Each such NAME directive specifies a different name for the same JPROC definition. Multiple NAME directives allow you to specify a different parameter in the *operand* field of each directive.

Note: *You may not give a JPROC any valid job control statement names (DVC, QGBL, etc.).*

When you call the particular NAME directive on the JPROC call, you can reference the parameter of the NAME directive with p(0), where p is the symbol used to reference positional parameters. This will be shown in an example, which should make this much clearer.

The format of the NAME directive is:

LABEL	ΔOPERATIONΔ	OPERAND
[//]symbol	NAME	param

The *symbol* specifies the name of the JPROC definition. This is the name that's used on the JPROC call to obtain the JPROC definition. The *param* is a parameter or parameter sublist that may be selected at job execution time.

Here's an example of this procedure:

```
MOD1 PROC P,1                          //MOD1 PROC P,1
DUMPJOB NAME Y                         //DUMPJOB NAME Y
DUMPSYS NAME X                         //DUMPSYS NAME X
// GO LABEL&P(0)              or       // GO LABEL&P(0)
//LABELY OPTION JOBDUMP                //LABELY OPTION JOBDUMP
// GO NEXT                             // GO NEXT
//LABELX OPTION SYSDUMP                //LABELX OPTION SYSDUMP
//NEXT NOP                            //NEXT NOP
```

This JPROC definition has two names: DUMPJOB and DUMPSYS. Positional parameters are referenced by the symbol P, so the parameter of the NAME directive is referenced as P(0). Assume that DUMPSYS is the name used on the JPROC call. The parameter on this NAME directive is X. When the first GO job control statement is interpreted, it would mean go to the job control statement with a symbol of LABEL&P(0). This &P(0) references the parameter of the selected NAME directive. In this case, it's X. So X is added to LABEL, giving the symbol LABELX. This job would have an OPTION SYSDUMP job control statement inserted at execution time. The procedure would then go to the next job control statement, the NOP.

If DUMPJOB is the name used on the JPROC call, the parameter on the NAME directive would be Y. When the GO job control statement is interpreted, it would mean go to the job control statement with a symbol of LABELY (from the LABEL&P(O)). This job would have an OPTION JOBDUMP job control statement inserted at execution time; the GO job control statement means go to the job control statement with a symbol of NEXT. This is the NOP job control statement; the OPTION SYSDUMP job control statement is skipped.

Ending the JPROC Definition

The END directive indicates the end of the JPROC definition. Therefore, it's the last item in a JPROC definition. Everything between the PROC and END directives is considered to be the body of the JPROC definition.

The format of the END directive is:

LABEL	ΔOPERATIONΔ	OPERAND
[//]symbol	END	unused

and, added to the PROC and NAME directives defined in "Naming the JPROC Definition" earlier in this section, looks like the following example:

```
MOD PROC P,1
DUMPJOB NAME Y
DUMPSYS NAME X
        .
        .
        .
      any
  job control
  statements
    needed
        .
        .
        .
   END
```

If you are submitting embedded data as part of a JPROC definition and the embedded data contains the characters END, a special situation arises because the run processor interprets the characters END as the END JPROC directive. To avoid this problem, you must use a // GBL job control statement to replace the END characters in the embedded data. This is an example:

```
        .
        .
        .
// OPTION SUB
// GBL X=END
// EXEC program-name
/$
        .
        .
        .
&X
        .
        .
        .
/*
```

Calling JPROC Definitions

Once you've written and debugged a JPROC definition, use the file symbiont to store it in the job control stream library file (YJCS) or an alternate library file, and then call it when you need it. Until that time, you can test it by placing the JPROC definition within a control stream and issuing a JPROC call containing the name you supplied on the NAME directive. (In this way, you can test a JPROC without having to actually file it.) The JPROC definition is stored temporarily, in the job's YRUN file. We'll explain this in a little more detail in "How JPROC Definitions Are Stored" later in this section.

To call the JPROC, you use a JPROC call in the control stream. When the run symbiont encounters the JPROC call, it searches the job's YRUN file, then searches the specified library file for the named JPROC definition, and then inserts the selected job control statements from the JPROC definition into the control stream at this point.

The format of the JPROC call statement is:

```
//[symbol] procname [p1,p2,...,pn,ki=vi,kj=vj,...,km=vm]
```

The *symbol* is a dummy label and is optional. When used, the *symbol* is substituted for the *symbol* specified in the *label* field of the PROC directive.

The *procname* specifies the name of the JPROC definition. This must be the same as that specified in the *label* field of a NAME directive in the JPROC definition being called.

The *p* represents positional parameters, and the *k=v* represents keyword parameters and their values.

Positional parameters specified in a JPROC call are associated with positional parameters specified in job control statements in the body of the JPROC definition. The PROC directive specifies the number of positional parameters allowed.

All parameters specified in the JPROC call must be separated by commas. Positional parameters must precede any keyword parameters. When a positional parameter is omitted, the comma must be retained to indicate the omission, except in the case of omitted trailing positional parameters. When there are no positional parameters preceding keyword parameters, two commas must precede the keyword parameters to indicate the omission of the positional parameters.

Keyword parameters are identified by name, not by position, so an omitted keyword parameter does not require a comma to indicate its omission. Keyword parameters may be specified in any order.

No more than one JPROC call can be on a single line.

How JPROC Definitions Are Stored

The file symbiont stores JPROC definitions in YJCS or an alternate SAT library file. (See "Building and Storing Job Control Streams and JPROCS" in Section 1.) The values to be used are not substituted for any preset values until the JPROC call is issued. Substitution then takes place, and the JPROC definition is then considered to be expanded.

A JPROC definition may be called as often an necessary, or until it's deleted from the library file.

The reading, verifying, and expanding of the entire control stream is a function of the run symbiont.

A job input directly from a reader device may include JPROC definitions in its control stream. The JPROC definition must appear in the control stream before any reference to it is made. Therefore, if a JPROC definition pertained to assigning devices to a job, it should be placed before any device assignment sets. Such JPROC definitions apply only to that particular job; they aren't stored in YJCS or an alternate library file, they're stored temporarily in the job's YRUN file. They also cannot be embedded within data.

Because the job's YRUN file is the first file searched for a JPROC definition, by placing a JPROC definition in the job control stream you have the ability to test the JPROC definition without storing it permanently. You can also use this facility to temporarily override a JPROC definition that's already stored. Whenever a JPROC definition being called is found in the job's YRUN file, YJCS or the alternate library file is not searched.

A sample job using this facility would look like this:

```
// JOB TESTPROC
MOD PROC P,1
DUMPJOB NAME Y
DUMPSYS NAME X
        .
        .
        .
       any job control statements needed
        .
        .
        .
     END
// DUMPJOB
/&
// FIN
```

In this example, the JPROC named as either DUMPJOB or DUMPSYS would be entered as a temporary JPROC definition, which is referenced later by the JPROC call of // DUMPJOB. Upon encountering the PROC directive, the run processor will file the statements up to the END directive into the job's YRUN file, which is scanned when the JPROC call is encountered.

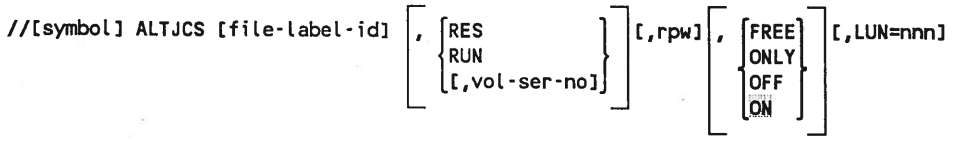
Specifying an Alternate Library File to Be Searched for JPROCS

The ALTJCS job control statement tells the run processor which alternate library file is to be searched for JPROCS. An alternate library file is one other than YJCS. // ALTJCS specifies an alternate library file to be searched for JPROCS only, not job control streams. An ALTJCS job control statement specification overrides an alternate library specification on the RUN/RV command that initiated processing of the job control stream.

You can specify multiple ALTJCS control statements in a control stream; the last library file specified is searched for JPROCS until the next ALTJCS statement is processed.

Note: *The run processor searches only the specified alternate library for JPROCS. Specify ALTJCS ,,,FREE to revert to the prior order of search (alternate library and then YJCS). The FREE option must also be specified before another ALTJCS statement can be specified.*

The format of ALTJCS is:



The *file-label-id* is 1 to 44 alphanumeric characters long. It is optional if you're not searching a new library, but changing the last parameter (FREE, ONLY, OFF, or ON) for an alternate library already defined in a previous ALTJCS statement. We'll discuss these options later. If you don't specify a *file-label-id, don't specify vol-ser-no* or *rpw*.

The *vol-ser-no* parameter specifies the volume serial number of the disk where the alternate library file resides. This parameter can also specify the volume serial number of a format-label diskette. RES, RUN, or the actual volume serial number of the disk or diskette may be specified. If no *vol-ser-no* is specified, the cataloged *vol-ser-no* is used; if it is not cataloged, RES is used.

The *rpw* parameter specifies a read password associated with the alternate library file. It must be specified if the file is cataloged with a read password. It is ignored if no read password exists for the file or if the file is not cataloged.

The *ONLY, OFF*, and *ON* parameters specify order-of-search options. *ONLY* specifies that only the identified alternate library file is to be searched. When OFF is specified, the alternate library file remains open to the run processor and can be searched again by the use of the ON or ONLY options. You specify this option if you no longer want an alternate library file searched for JPROCS. *ON* specifies that the identified alternate library file is to be searched first and then YJCS. ON is the default option. *FREE* is equivalent to *OFF*, except that it also frees the alternate device (from the run processor).

Using the *LUN* keyword parameter, you supply a logical unit number to indicate the device type and characteristics for the alternate library. *LUN* is never specified unless a volume serial number is also specified. It is especially useful where either a disk or format-label diskette can be the alternate library volume.

The volume serial numbers for disk and format-label diskette are syntactically the same. As a result, the system cannot determine if a disk or format-label diskette is required unless the volume is already mounted, or unless you use *LUN* parameter. If you don't specify a logical unit number and if the proper volume isn't already mounted, mount messages suggesting a disk drive, for example, could be directed to the operator when a format-label diskette is actually required. The *LUN* parameter helps avoid such confusion.

Notes:

1. LUN *is used only to determine the device type and characteristics. It has no relationship to logical unit numbers used elsewhere in the job control stream.*

2. *Confusion with mount messages is also avoided if the DVC-LFD sequence for the file is cataloged. (See "File Cataloging" in Section 6.) By simply providing a* file-label-id *in the ALTJCS statement, the correct volume serial number as well as device type is extracted from the catalog (according to the label specified).*

You can identify alternate libraries for control streams and JPROCS through the FILE system console command and the RUN/RV workstation or console command. Workstation commands are explained in the *Interactive Services Operating Guide*, UP-9972. System console commands are explained in the operations handbook for your system.

Parameter Referencing

The parameters of job control statements that require substitute values at execution time must begin with an indicator of &.

For example, if, in the body of a JPROC definition, you have a DVC job control statement in which you wanted to vary the logical unit number, it could be coded as follows:

```
// DVC &P(2)
```

The P is an arbitrary symbol assigned by you in the PROC directive; the (2) indicates that the logical unit number to be inserted is coded as the second positional parameter on the JPROC call. The parentheses around the 2 are required.

In this example,

```
// DVC &P(1,2)
```

the (1,2) indicates that the logical unit number to be inserted is coded as the second subparameter under the sublist for the first positional parameter.

For each character string following the single ampersand, a substitution is made. If the character string is invalid (not defined in the PROC directive), a null character string is inserted.

Any job control statement may be continued between parameters, or between the operation and the first parameter. No job control statement can exceed column 71. This means that the total number of characters cannot exceed column 71, even after substitution. The maximum of 71 characters includes embedded spaces. Column 72 is used to indicate continuation.

The length of a single parameter is 242 characters. For positional parameters, this is the value; for keyword parameters, it is the keyword and the value. If a parameter is sublisted, the maximum length is decreased by 2 for each element of the sublist.

Note: *The maximum length of a single* operand *of a job control statement is 252 characters. Thus, if you have a parameter of 242 characters, there are only 10 characters left for other parameters.*

Here are some examples.

Example 1

In this portion of a JPROC definition, we'll see how a value is given to the DVC job control statement in the body of the JPROC definition.

```
     PROC POS,1
ACTI NAME
       .
       .
       .
// DVC  &POS(1)
       .
       .
       .
     END
```

Let's say that the JPROC call is this:

```
// ACTI 10
```

the DVC job control statement that would be generated and inserted to the control stream would be as follows:

```
// DVC 10
```

Example 2

If part of the JPROC definition looked like this:

```
     PROC KEY1=90
ACT2.NAME
      .
      .
      .
// DVC    &KEY1
      .
      .
      .
     END
```

and the following JPROC call was issued:

```
// ACT2 KEY1=20
```

this job control statement would be generated:

```
// DVC 20
```

If the JPROC call was issued without the KEY1 parameter, the value of 90 set in the JPROC definition would be used.

Example 3

This JPROC definition has one positional and one keyword parameter, and two NAME directives.

```
1.  LAB PROC POS,1,KEY1=10
2.  MASTER NAME 20
3.  DETAIL NAME 30
            .
            .
            .
4.  //&LAB DVC &POS(0)
5.  //     DVC &POS(1)
6.  //     DVC &KEY1
           .
           .
           .
        END
```

When this JPROC call is issued:

```
//L1 MASTER 40,KEY1=50
```

these job control statements are generated:

```
//L1 DVC 20
//   DVC 40
//   DVC 50
```

Line 4 in the JPROC definition means to take the value of the parameter in the NAME directive that matches the name on the JPROC call - MASTER. So the first DVC job control statement has a logical unit number of 20. Line 5 means to take the value of the first positional parameter in the JPROC call; the second DVC job control statement has a logical unit number of 40. Line 6 means take the value of the KEY1 keyword parameter; the third DVC job control statement has a logical unit number of 50. LI is specified by the JPROC call as being the substitute value for the symbol in the PROC directive. Line 4 will use this value. So, the first DVC job control statement has a symbol of LI.

Example 4

A parameter sublist may be referenced. This is done with a secondary level of indexing, which is shown in the following example:

```
     PROC POS,1,KEY=(10,20)
     EXAM NAME (30,40)
      .
      .
      .
1.  // DVC &KEY(1)
2.  // DVC &KEY(2)
3.  // DVC &POS(0,1)
4.  // DVC &POS(0,2)
      .
      .
      .
     END
```

When the following JPROC call is used

```
// EXAM KEY=(50,60)
```

these job control statements are generated:

```
// DVC 50
// DVC 60
// DVC 30
// DVC 40
```

Line 1 of the JPROC definition means use the first subparameter of the KEY keyword parameter. The JPROC call uses this keyword parameter, so its new values (50 and 60) override the values assigned in the JPROC definition (10 and 20). Line 2 means use the second subparameter of the KEY keyword parameter. Line 3 means use the first subparameter on the NAME directive (0,1), and line 4 means use the second subparameter on the NAME directive (0,2).

Example 5

A reference to a parameter may occur anywhere in the body of a procedure definition. If the reference is the only field, and therefore naturally delimited, there is not much likelihood of confusion. If the possibility of confusion exists, the reference may be terminated with a period, which is a concatenation operation. The period is dropped during the expansion of the control stream.

The following JPROC definition has two keyword parameters: KEY1 and LABEL; neither has default conditions.

```
     PROC KEY1,LABEL
COM  NAME
      .
      .
      .
// OPR &KEY1.IS&LABEL.1990
      .
      .
      .
     END
```

If this JPROC call was used

```
 // COM KEY1=TODAY-,LABEL=-OCTOBER-6-
```

this would be generated:

```
// OPR TODAY-IS-OCTOBER-6-1990
```

Section 9
Using the Interactive Job Control Dialog

The Function of the Job Control Dialog

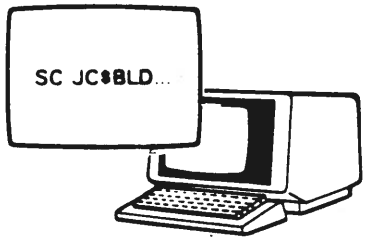
The job control dialog is an interactive facility of OS/3 that guides you through the process of building a job control stream or user JPROC from a workstation. To begin a job control dialog session, key in SC JC$BLD. This activates the dialog processor and opens the job control dialog file. Dialog text is displayed at the workstation screen and your responses to the dialog are entered at the workstation keyboard. The dialog processor passes your responses to the system program JC$BLD, which creates your control stream or JPROC and stores it in the system file YJCS. The functions of the dialog processor, which manages a dialog session, are detailed in the appropriate operations guide.

Note: *If you encounter system errors when keying in SC JC$BLD, key in RV JC$BLD and press XMIT. A short paragraph explaining RUN libraries is then displayed followed by the question DO YOU WANT TO SAVE RUN LIBRARIES? (Y OR N). Key in Y so that you'll be able to enter the SC JC$BLD command without encountering any errors in the future.*

The job control dialog introduces the concept of job control and (if you're building a control stream) presents job control statements in the form of menu items from which you choose the statements you want. If you need a dialog concept or particular statement explained, you can ask for help - by keying in HELP or a choice that generates HELP screens. HELP screens explain the choice or statement parameters to you. When you make a valid choice, the dialog resumes at the point where it was interrupted. The HELP screen facility of the job control dialog can be used selectively (statement by statement) so that you receive detailed explanations only when you need them. More experienced users, then, can execute the dialog session quickly while still being constrained to build syntactically correct statements. Figure 9-1 presents an overview of the process of using the job control dialog to build a control stream or user JPROC.

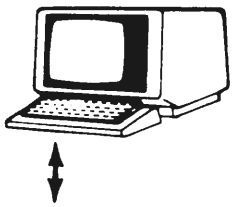
STEP 1

Key in the SC JC$BLD command to initiate a job control dialog session.



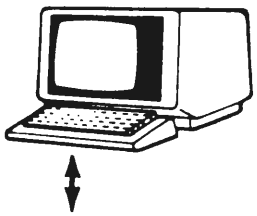
STEP 2

The dialog processor is activated and the job control dialog file is opened in response to the command ... begin executing the dialog.



STEP 3

The dialog processor routes your dialog responses to the system program JC$BLD.



STEP 4

JC$BLD uses your responses to the dialog to build a job control stream or user JPROC and stores it in YJCS.

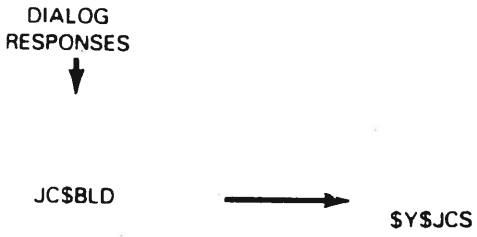


Figure 9-1. Using the Job Control Dialog to Build a Control Stream or User JPROC

Building a Control Stream with the Job Control Dialog

Let's begin a sample job control dialog session. First, you perform the system LOGON procedures described in the appropriate workstation user guide. Then, you key in SC JC$BLD and its associated parameters. The first dialog screen looks like this:

```
                DIALOG FOR JOB CONTROL

PROGRAM=
THIS DIALOG PREPARES A JOB CONTROL STREAM OR PROCEDURE
(JPROC).  FOR AN EXPLANATION OF THE DIALOG  PROCESS, ENTER
'HELP' IN THE SPACE PROVIDED.  HELP
```

If you key in HELP, these screens are displayed:

```
THE DIALOG FOR JOB CONTROL IS A METHOD OF CONSTRUCTING
JOB CONTROL STREAMS AND PROCEDURES (JPROCS) USING COMPUTER
ASSISTANCE.  PROMPTING FOR DATA ENTRY OR SELECTING FROM
AMONG AVAILABLE OPTIONS IS ALWAYS PROVIDED, AND YOU CAN
ASK FOR MORE DETAILED EXPLANATIONS OF STATEMENTS,
PARAMETERS, AND OPTIONS.  AFTER A STATEMENT IS COMPLETED,
THE IMAGE BUILT BY THE COMPUTER AS A RESULT OF YOUR CHOICES
IS DISPLAYED ON THE WORKSTATION SCREEN.  YOU MAY ACCEPT IT
FOR OUTPUT, CORRECT IT, OR REJECT IT ALTOGETHER.
```

Note: *To proceed from one screen to the next, you usually press the transmit key. Whenever necessary, a note will appear at the bottom of the screen reminding you to do this.*

```
THE JOB CONTROL SETS ARE FORMED BY MAKING SELECTIONS FROM
MENUS OF AVAILABLE OPTIONS, AND ENTERING SOME TYPES OF
DATA DIRECTLY.  THIS ALLOWS YOU AS MUCH FREEDOM IN YOUR JOB
CONTROL AS OTHER MEDIA, BUT AT THE SAME TIME PROVIDES
A STRUCTURE TO JOB CONTROL CREATION WHICH HELPS TO
PREVENT MANY COMMON ERRORS.  REMEMBER, HOWEVER, THAT THE
DIALOG DOES NOT RECOGNIZE THE SAME JOB CONTROL ERRORS AS
THE RUN PROCESSOR.  DIALOG ERROR CHECKING IS LIMITED TO
DIALOG OPERATION ERRORS, AND DATA TARGET MISMATCHES
(SUCH AS TRYING TO PUT ALPHABETIC DATA IN A STRICTLY
NUMBER FIELD).
```

The next screen asks what type of module you want to build:

```
JOB CONTROL MODULE TYPES


USE THIS MENU TO SELECT THE TYPE OF MODULE TO BE PREPARED:

1.  JOB CONTROL STREAM

2.  USER WRITTEN JOB CONTROL PROCEDURE (JPROC)

3.  HELP

SELECT ITEM BY ENTERING NUMBER.  ▶_ _
```

If you ask for HELP, these screens are displayed:

```
IN ORDER TO EXECUTE ANY JOB, IT IS NECESSARY TO CONVEY TO
THE COMPUTER EXACTLY WHAT YOU WANT TO DO, AND WHAT RESOURCES
(PRINTER, READER, DISKS, ETC) ARE NEEDED.  THIS IS ACCOMPLISHED
THROUGH THE USE OF JOB CONTROL.  THERE ARE TWO TYPES OF JOB
CONTROL MODULES.  THE COLLECTION OF JOB CONTROL STATEMENTS USED
TO RUN A JOB IS CALLED A JOB CONTROL STREAM, SOMETIMES REFERRED
TO AS THE JOB STREAM OR CONTROL STREAM.  IN IT, THERE MAY BE JOB
CONTROL STATEMENTS, CALLS TO SYSTEM SUPPLIED PROCEDURES, AND THE
SECOND TYPE OF MODULE - USER-WRITTEN PROCEDURES (JPROCS).
```

```
JOB CONTROL PROCEDURES HAVE TWO PARTS - THE DEFINITION
AND THE CALL.  THE DEFINITION IS THE JPROC MODULE CREATED
BY THE DIALOG.  THE CALL IS A STATEMENT IN THE CONTROL
STREAM WHICH HAS THE JPROC NAME AS THE COMMAND, AND
PROVIDES ANY NECESSARY PARAMETERS.  THE JPROC CALL IS USED
AS AN ABBREVIATION TO PREVENT CODING THE DEFINITION MANY
TIMES.  WHEN THE CONTROL STREAM IS PROCESSED, EACH CALL IS
REPLACED BY THE APPROPRIATE DEFINITION WHICH HAS BEEN PUT
AT THE BEGINNING OF THE STREAM OR STORED IN A SYSTEM FILE
($Y$JCS).  THE RESULT IS THE SAME AS IF THE DEFINITION HAD
BEEN CODED INSTEAD OF THE CALL.
```

Once again, you're asked what type of module you want to build.

```
JOB CONTROL MODULE TYPES


USE THIS MENU TO SELECT THE TYPE OF MODULE TO BE PREPARED:

1.  JOB CONTROL STREAM
2.  USER WRITTEN JOB CONTROL PROCEDURE (JPROC)
3.  HELP

SELECT ITEM BY ENTERING NUMBER.  ▶_ _
```

You can ask that HELP screens explaining the choices be displayed again (by keying in 3), but let's assume you want to build a control stream. The next screen displayed is the JOB control statement screen:

```
STATEMENT:  JOB

FORMAT:  //SYMBOL JOB JOBNAME,PRI,MINSTORE,MAXSTORE,TASKS,
         TIME,OPTIONS,ACCT,BUFFERS,LOG,HDR

FUNCTION:  THIS STATEMENT IDENTIFIES A JOB AND INDICATES
           THE BEGINNING OF CONTROL INFORMATION FOR THE
           JOB.  THE SAME NAME IS GIVEN TO THE JOB'S RUN
           FILE ($Y$RUN).

IF YOU WILL NEED HELP WITH THIS STATEMENT, ENTER HELP.
```

What if you didn't need HELP screens? The job control dialog screens vary according to the responses you make to the dialog. The initial screen is the same:

```
                DIALOG FOR JOB CONTROL


THIS DIALOG PREPARES A JOB CONTROL STREAM OR PROCEDURE
(JPROC).  FOR AN EXPLANATION OF THE DIALOG PROCESS, ENTER
HELP IN THE SPACE PROVIDED.  _ _ _ _
```

Because you don't need HELP screens to explain the dialog process, simply press the transmit key to display the next screen. The next screen displayed is:

```
JOB CONTROL MODULE TYPES:


USE THIS MENU TO SELECT THE TYPE OF MODULE TO BE PREPARED:

1.  JOB CONTROL STREAM
2.  USER WRITTEN JOB CONTROL PROCEDURE (JPROC)
3.  HELP

1
```

You key in 1, indicating that a job stream is being prepared. The next screen displayed (since HELP screens weren't requested) is the JOB control statement screen:

```
STATEMENT:  JOB

FORMAT:  //SYMBOL JOB JOBNAME,PRI,MINSTORE,MAXSTORE,TASKS,
           TIME,OPTIONS,ACCT,BUFFERS,LOG,HDR


FUNCTION:  THIS STATEMENT IDENTIFIES A JOB AND INDICATES
           THE BEGINNING OF CONTROL INFORMATION FOR THE
           JOB.  THE SAME NAME IS GIVEN TO THE JOB'S RUN
           FILE ($Y$RUN).


IF YOU WILL NEED HELP WITH THIS STATEMENT, ENTER HELP.
```

As you can see, there is a big difference in the path the job control dialog takes, depending on your responses to the dialog.

Let's take the dialog one step further. If you key in HELP in response to the JOB statement screen, each parameter of the JOB statement is explained. If HELP is not requested, you are simply asked to key in the parametric values, without benefit of prompting screens. When the JOB statement is built, it is displayed and you have a final chance to change the parameters of the statement, with or without HELP screens, or accept the statement as it appears. When the JOB statement is accepted, the next screen presented is the job control statement master menu.

```
          JOB CONTROL STATEMENT MASTER MENU


 1.  ALTER    11. EXEC   21. LFD     31. ROUTE   41. VFB
 2.  ALTJCS   12. EXT    22. MTC     32. RST     42. VOL
 3.  CAT      13. FREE   23. NOP     33. RUN/RV  43. /$
 4.  CC       14. GBL    24. OPR     34. SCR     44. /*
 5.  CR       15. GO     25. OPTION  35. SET     45. /&
 6.  DATA     16. IF     26. PARAM   36. SFT     46. SYSTEM JPROCS
 7.  DECAT    17. JNOTE  27. PAUSE   37. SKIP    47. GENERAL ENTRY
 8.  DST      18. JSET   28. QGBL    38. SPL     48. END OF SESSION
 9.  DVC      19. LBL    29. QUAL    39. UID     49. HELP
 10. EQU      20. LCB    30. REN     40. USE

ENTER SELECTION BY NUMBER _ _
IF YOU WILL NEED HELP WITH THIS STATEMENT, ENTER HELP_ _ _ _
```

The rest of the job control dialog works in the same way as for the initial module-type choice and the JOB statement screens. Each statement you choose from the master menu is displayed and you are asked if you need help to build it. If you do, HELP screens are displayed that explain the parameters of each statement.

Note: *The DD job control statement is not provided on the job control statement master menu. To include this statement in your job control stream, make the GENERAL ENTRY menu selection (47), then enter the statement and its parameters in the space provided.*

The control stream you create is stored in YJCS. A printed summary of the dialog session, organized by sequentially numbered paragraphs, is produced by the dialog processor. The default logical unit number of the printer file (printed summary) output is 20 - any printer. You can accept this default or, during the dialog session, provide a specific printer's logical unit number. Table A-3 of the *Job Control Programming Reference Manual*, UP-9984, lists the OS/3 logical unit numbers for printers.

Building a User JPROC with the Job Control Dialog

The dialog for creating a JPROC guides you through the process of defining your JPROC and building the job control statements and system JPROCS you want to include in the body of the JPROC definition.

The procedure for initiating the dialog is the same as for building a job control stream: perform the system LOGON procedures and key in SC JC$BLD.

When the job control dialog asks you whether you're building a job control stream or user JPROC, key in the choice for user JPROC. The dialog then presents menus for:

- Beginning the JPROC (PROC, NAME)
- Choosing job control statements
- Choosing system JPROCS
- Ending the JPROC (END)

As is the case when you're building a job control stream, these menus generate other menus based on your responses to the dialog.

You can request HELP screens at any point in the dialog where you need choices or parameters explained. After the HELP screens are displayed and you make a valid choice, the dialog returns to the point where it was interrupted.

JC$BLD uses your dialog responses to create a JPROC.

Note: *If you store a JPROC in your own (alternate) library file instead of YJCS, you must include the ALTJCS job control statement in any subsequent job control stream that calls the JPROC. ALTJCS identifies the JPROC and applies* only *to JPROCS.*

Entering Embedded Data

To enter embedded data from a workstation, first choose the /$ (start-of-data) statement from the job control statement master menu. Then, when the master menu is redisplayed, make the GENERAL ENTRY selection (47). Once this is done, you'll be able to enter your embedded data. When all embedded data is entered and the master menu is presented again, choose the /* (end-of-data) job control statement.

If you plan to enter dialog specification language (DSL) source code as embedded data from the workstation, a special situation arises because the characters that denote the start of a DSL comment are the same as the end-of-data job control statement (/*). It's necessary, then, to substitute another set of characters for the end-of-data job control statement. You do this through the OPTION job control statement.

When the OPTION statement menu is displayed at the workstation screen, choose an OPTION EOD statement. The format is OPTION EOD=xx. The first character you select must be a slash (/). The second character can be anything but a slash (/), an asterisk (*), an ampersand (&), or a currency symbol ($). Let's say you choose /Z. Then, when the end-of-data statement is displayed as part of the job control dialog menu, you choose GENERAL ENTRY and key in your substitute characters; /Z in this case. The control stream you create, then, will include these job control statements:

```
      .
      .
      .
// OPTION EOD=/Z
      .
      .
      .
/$          (start of data)
      .
      .     (DSL source code)
      .
/Z          (end of data)
      .
      .
      .
```

You key in your DSL source code when the dialog requests it. By substituting different characters for the end-of-data job control statement, you avoid any conflict with the DSL start-of-comment delimiter.

Changing Dialog Responses

Once you build a control stream or JPROC from a workstation, you may be able to use it for other jobs by making only a few changes to it or, you may discover that you need to correct it. Rather than building a new control stream or JPROC from scratch to incorporate the changes you want, you can use the audit version of the dialog processor to change or edit the responses you made in a previous job control dialog session. The audit version of the dialog processor outputs an audit file containing a complete record of your dialog responses and accepts as input an existing audit file of your responses to a previous dialog. An existing audit file used as input is considered an old audit file. The audit file produced as output of the current dialog session is considered a new audit file.

You begin a dialog session, which uses the audit version of the dialog processor, by performing the system LOGON procedures and keying in RV JC$BLD. When you identify a new and/or old audit file (by volume serial number and file label) during the resulting dialog session, the system loads the audit version of the dialog processor.

Note: *Old and new audit file names cannot be the same when responding to JC$BLD queries.*

The audit version of the dialog processor (Figure 9-2) also outputs a printed summary of a dialog session that is used as a guide to changing dialog responses in a subsequent session. The summary is organized by sequentially numbered paragraphs. When you

use the audit file as input to the dialog processor in a subsequent session, the job control dialog asks you to enter the numbers of the paragraphs you want to change. The summary lists these paragraph numbers.

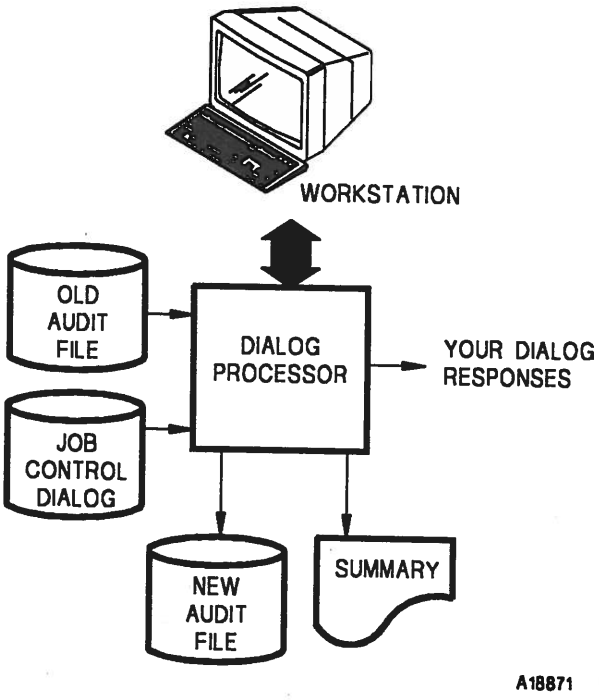


Figure 9-2. Audit Version of the Dialog Processor

Note: *Audit files must be previously allocated MIRAM files.*

The audit version of the dialog processor allows you to present the job control dialog quickly and create a "new" control stream or user JPROC by changing only the responses that need to be changed. Unchanged responses are automatically routed from the old audit file by the dialog processor to JC$BUILD - without your intervention. During the same session, you enter your new responses to the job control dialog. You can also produce a new audit file (if you've specified it in the build command) that contains a mix of responses from the old audit file and responses entered during the current session. This audit file can then be used as input to the dialog processor in a subsequent session.

Note: *Only control streams and user JPROCS created using the job control dialog can be changed in a subsequent dialog session.*

Suppose you build a control stream for a job that runs nearly every day with only a few changes to the control stream. Perhaps you want disk and print output on some days, and disk output only on other days. You first build the control stream on Monday, specifying that a new audit file and a printed summary of the session be produced. You use the audit file as input to Tuesday's dialog session and use the summary report as your guide to changing the appropriate dialog responses.

Figure 9-3 traces the process of changing your dialog responses in a subsequent session.

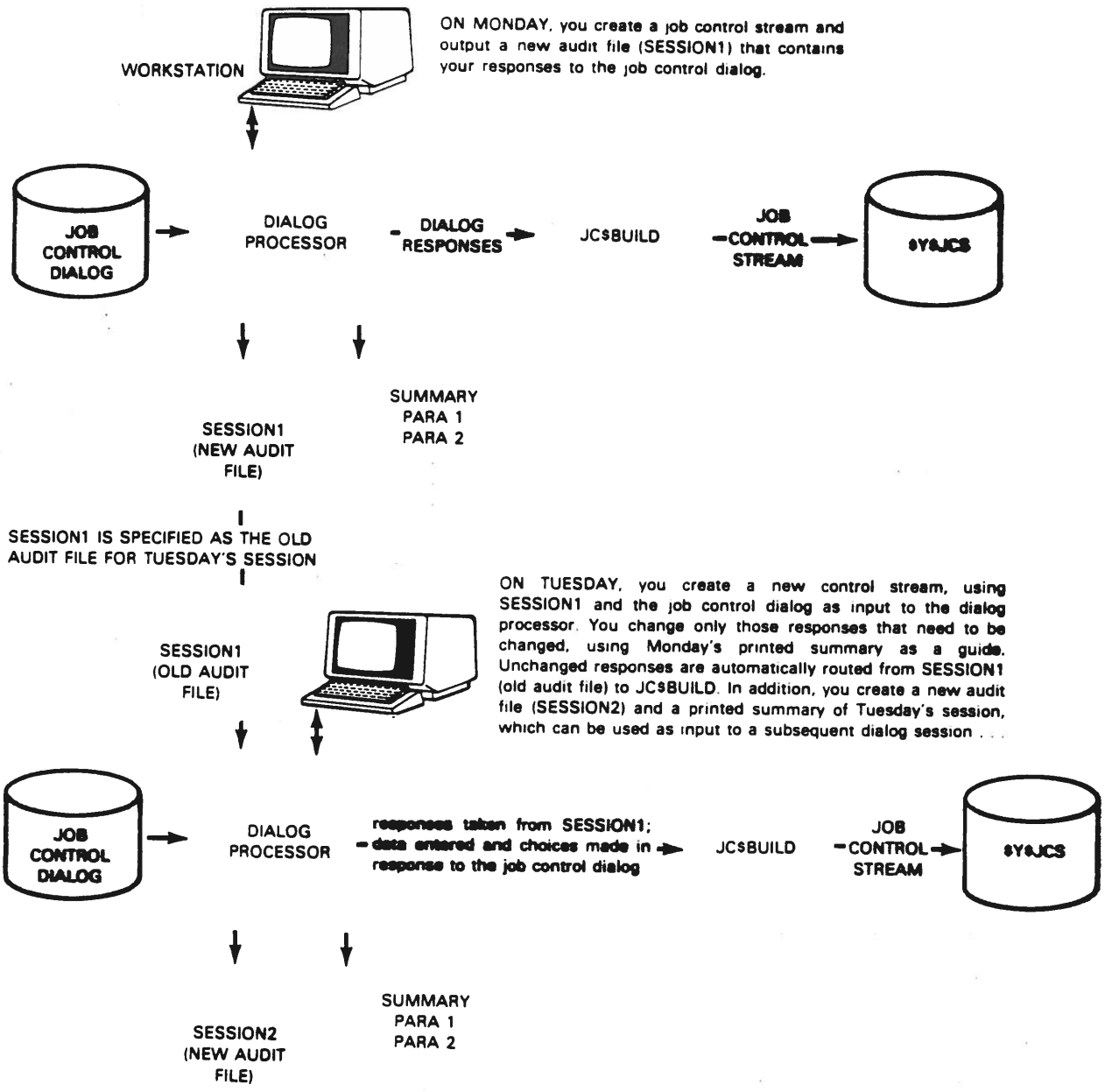


Figure 9-3. Changing Your Dialog Responses

The appropriate operations guide has more information about using the audit version of the dialog processor, including information about breaking off a session and continuing it at a later time - without losing your changed dialog responses.

Appendix A
Statement Conventions

Job Control Statement Format

A job control statement has a maximum of five fields, which must appear in the following order:

1. Indication Field

   Distinguishes job control statements from data. It is required and begins with either //, /&, /$, or /*.

2. Label Field

   Contains a 1- to 8-alphanumeric-character symbol; the first character must be alphabetic. Unless this field is explained in a specified control statement, it is the target address of a SKIP, GO, or IF control statements or the ABNORM=label keyword parameter of the EXEC statement. This field is not separated from the indication field by a space; it immediately follows the //.

3. Operation Field

   Contains the name of the function to be performed. It is required for all job control statements having an indication field of //. At least one space must separate the operation field from the label field.

4. Operand Field

   Contains the specific information concerning the items upon which a job control function is to operate or the manner in which the function is to be performed. At least one space must separate the operand field from the operation field.

5. Comments Field

   Contains any descriptive information desired but not processed. The field must not contain a slash character. For those job control statements in which an operand is not permitted, such as the FIN control statement, all information beyond the operation field is treated as the comments field.

Excluding the indication and label fields, consecutive fields must be separated by one or more spaces. A space may not appear in a field except within apostrophes (hexadecimal code 7D) or parentheses in an operand field.

Example

```
//MYTARGAD     LBL    'MASTER CUST'     NAME FILE
  1    2     6  3  6        4         6     5
```

Notes:

1 Indication field

2 Label field

3 Operation field

4 Operand field: Note that spaces are allowable, because of the use of apostrophes.

5 Comments field

6 Field separation spaces

How Job Control Statements Are Presented

The conventions used to delineate job control statements are:

- Positional parameters must be written in the order specified in the operand field and must be separated by commas. When a positional parameter is omitted, and subsequent positional parameters are being specified, the commas associated with positional parameters must be retained; otherwise, the specified parameters will not be processed as required. If no subsequent parameters are being specified, their associated commas should also be omitted.

  For example, the ALTER job control statement has four optional positional parameters. This is presented in text in the following format:

```
//[symbol] ALTER [phase-name][,address][,change][, {RESET}]
                                                  {ORG  }
```

Then, the statement may be written:

```
// ALTER phase-name,address,change,RESET
// ALTER phase-name,address,change
// ALTER phase-name,address
// ALTER phase-name
// ALTER phase-name,,change
// ALTER ,,,RESET
// ALTER phase-name,,,ORG
```

Note that three commas are required in both the last and next-to-last examples. In the next-to-last example, the three commas are encountered before any parameters and are thus used to imply that the *first*, as well as the second and third parameters, were omitted. In the last example, a parameter is encountered before any commas, and thus the first comma is used to separate the first parameter from the omitted second and third parameters.

```
// ALTER ,,,ORG
```

If the last example used four commas, it would appear that ORG was the fifth parameter. And, because job control only associates four parameters with the ALTER job control statement, the ORG parameter specification would be invalid.

- A keyword parameter consists of a word or a code immediately followed by an equal sign, which is, in turn, followed by a specification. Keyword parameters can be written in any order in the operand field. Commas are required only to separate parameters.

The VFB job control statement has the following format:

```
//[symbol] VFB LENGTH=lines[,DENSITY={6|8}] [,FORMNAME=symbol]
        [,USE={STAND1|OWNVF1|OWNVF2-OWNVF9}]  [,TYPE={SDMA|0770|0776|*}] [,OVF=(line-1,...,line-n)]
    [,OVF2=(line-1,...,line-n)][,CD1=(line-1,...,line-n),...]
    [,CD15=(line-1,...,line-n)]
    [,HP=n]
```

However, for the purpose of explaining the use of keyword parameters, we'll use only the first four parameters. Thus, we arrive at the following format:

```
//[symbol] VFB LENGTH=lines[,DENSITY={6|8}] [,FORMNAME=symbol][,USE=STAND1]
```

Then, this job control statement may be written as:

```
// VFB LENGTH=lines,DENSITY=6,FORMNAME=symbol,USE=STAND1
// VFB USE=stand,FORMNAME=symbol,DENSITY=6,LENGTH=lines
// VFB DENSITY=6,LENGTH=lines
// VFB LENGTH=lines
// VFB FORMNAME=symbol,USE=STAND1
```

- A job control statement may consist of a group of positional parameters followed by a keyword parameter (as the last parameter).

  For example:

```
//[symbol] EXEC program-name[, [library-name]  ] [,[±]switch-priority][,ABNORM=label]
                             [ $Y$RUN        ]
                             [ $Y$LOD        ]
```

  Since the last parameter is a keyword (not the last positional) parameter, this statement may be written as follows:

```
// EXEC program-name,ABNORM=label
// EXEC program-name,library-name,ABNORM=label
```

  Commas for the omitted positional parameters may be retained if desired. For example:

```
// EXEC program-name,,,ABNORM=label
// EXEC program-name,library-name,,ABNORM=label
```

  The conventions for coding commas when a positional parameter is omitted and subsequent positional parameters are being specified still apply. When the second positional parameter is omitted, for example, the EXEC statement must be coded as follows:

```
// EXEC program-name,,switch-priority,ABNORM=label
```

- A positional or keyword parameter may contain a sublist of parameters called subparameters, which are separated by commas and enclosed in parentheses. The parentheses must be coded as part of the list. The subparameters within the parentheses may be positional, in which case the associated commas must be retained if a parameter is omitted, except for the case of trailing parameters, or they may be nonpositional. The description of the subparameters will indicate whether or not they are positional or nonpositional.

  For example:

```
[,OVF=(line-1,...,line-n)][,OVF2=(line-1,...,line-n)]
```

- Capital letters, commas, equal signs, and parentheses must be coded exactly as shown.

```
CMcc

X'aa'

NUMBCHAR=n

(NOV)
```

- Lowercase letters and words are generic terms representing information that must be supplied by the user. Such lowercase terms may contain hyphens and acronyms (for readability). For example:

```
phase-name

max-time

destination

filename
```

- Information contained within braces represents mandatory entries of which one must be chosen, such as:

```
{BB,nn
 BM,nn
 FB,nn
 FM,nn
 WM,nn
 RL
 RU}
```

- Information contained within brackets represents optional entries that (depending upon program requirements) are included or omitted. Braces within brackets signify that one of the specified entries must be chosen if that parameter is to be included. For example:

```
[sched-priority]
[{addr}]
[{ALT }]
[{OPT }]
[{IGNORE}]
```

- An optional parameter having a list of optional entries may have a default specification that is supplied by the operating system when the parameter is not specified by the user. Although the default may be specified with no adverse effect, it is considered inefficient to do so. For easy reference, when a default specification occurs in the format delineation it is printed on a shaded background. If, by parameter omission, the operating system performs some complex processing other than parameter insertion, it is explained in the parameter description.

```
[ [library-name]  ]
[ {$Y$RUN       } ]
[ {$Y$LOD       } ]
```

```
[TYPE= {0770} ]
       {SDMA} ]
```

- An ellipsis (series of three periods) indicates the presence of a variable number of entries.

- When a portion of a parameter is *underlined*, only that portion need be specified. For example:

```
FORMNAME=symbol
```

can be coded as:

```
FO=symbol
```

Coding Conventions

All the job control statement information starts in position 1 and is not permitted to extend for more than 71 positions. Job control statements begin with either one or two slashes. In those with only one slash, no space is permitted between the slash and the next character. However, one space must appear between this character and the operand field. In job control statements beginning with two slashes, at least one space must appear between the last slash and the operation field (except when using the continuation statement (//n) or the label field).

More than one job control statement of the type beginning with two slashes may be written on a card, but must not extend beyond position 71. At least one space must precede the slashes denoting the beginning of the second job control statement; this is referred to as multistatement coding.

Numbers required for particular parameters can be expressed in decimal or hexadecimal. Numbers preceded by D' are considered decimal. Numbers preceded by X' are considered hexadecimal. (A trailing quote may optionally be specified.) All of the following represent the same value:

```
X'FF

X'FF'

D'255

D'255'
```

Numbers not preceded by X' or D' are automatically considered decimal except in the following cases when they default to hexadecimal:

- Main storage sizes specified on the JOB statement (*min* and *max* parameters)
- Memory sizes specified on the OPTION MIN and OPTION MAX job control statements
- Absolute disk addresses specified on the EXT statement (*addr* or *Tccc:hh* parameters)
- Address on the ALTER statement (*address* parameter)
- Expansion limit on the SFT statement's DLOAD option (*expansion-limit* parameter)

Character strings on the ALTER, LCB, and SET job control statements must be specified as shown in their formats.

Statement Continuation

A continuation line is not considered to be a job control statement in itself. It is a line that contains the continuation of a job control statement in a preceding line. A nonblank character must appear in position 72 of the line containing the statement to be continued. Continuation may be used with any job control statement that contains at least the first two fields.

A continuation statement must begin with either the 3-character sequence //n, or just a simple //, which then must be separated by one or more blanks from the continued portion of the job control statement. The continued statement takes the form:

```
//[n] param-1...param-n
```

The *n* is a decimal number in the range of 1 through 9. The numbers do not need to be consecutive; however, each number must be greater than or equal to the preceding number used in the control stream. This is an optional field and may be left blank, or numbers can be used so you can keep a visual record of the amount of continuation statements used.

For example, you could code the continuation as either

```
                                                    Column 72 ──┐
// parameters                                                 X <┘
//1 parameters                                                X
//2 parameters
```

or

```
// parameters                                                 X
//1 parameters                                                X
//1 parameters
```

or

```
// parameters                                                 X
// parameters                                                 X
// parameters
```

The *param-1...param-n* are the parameters required to continue the immediately preceding job control statement.

Continuation can only occur at the blanks following the operation or operand fields, or after the comma following a parameter in the operand field. When you continue job control statements, the positions between the last used position and position 72 must be blank. Any information you intended as a comment for this line would be treated as data.

An error message occurs if:

- Column 72 contains a nonblank character and the card is not a valid continuation
- Comments extend past column 71
- A parameter list is not delimited by a comma

An example of the continuation of a multistatement line of coding is as follows:

```
// DVC 50  // VOL ABC123,T12345,T57341  // EXT ST,C,3,                X
//1 CYL,1  // LBL MASTER  // LFD FILEX
```

Software Conventions

The following rules and conventions apply to the processing of job control statements and directives:

- Data cannot be contained on a job control statement.
- Embedded data is normally assumed to be 80 characters long; when input from diskette, data can be 80 or 128 characters long.
- Comments cannot contain a slash.
- Job control does not scan past position 72; however, embedded data of up to 128 bytes is passed through.
- The CR job control statement, and a JPROC call when used, must be the last statement on the card.
- The following job control statements and JPROC directives cannot be part of a multistatement line:

  ```
  // JOB
  // FIN
  // PROC
  // NAME
  // END
  ```

  The // need not start in column 1, but must be first on the card. The // is optional for PROC, NAME, and END.

- The following job control statements cannot be part of a multistatement line. They need not start in column 1, but must be first on the card.

  ```
  /*
  /&
  /$
  ```

Appendix B
Operation Considerations

System Libraries

There are five primary system program libraries stored on the system resident device (SYSRES). The format of these libraries conforms to the standards established by the librarian. For a description of these standards, see the *System Service Programs (SSP) Operating Guide*, UP-8841. As in all disk files, an entry for each library file is maintained in the volume table of contents (VTOC) on SYSRES. These files may be accessed by your program without specifying a DVC-LFD sequence, provided the file name you use in your program is the same as the file identifier. For example: YLOD.

The five library files are:

- System Load Library File

  This file contains the load modules that are generated as output from the linkage editor or the librarian. This includes system software load modules. This file is used as the default input file to the system loader.

  The file identifier for this file is YLOD.

- System Object Library File

  This file contains the object modules generated as output from the language translators. This includes system software object modules. This file is the default input file to the linkage editor.

  The file identifier for this file is YOBJ.

- System Macro Library File

  This file contains the standard system macro definitions, and is used as the default input file for these definitions by the assembler.

  The file identifier is YMAC.

- System Source Library File

  This file provides permanent storage for source modules consisting of source coding processed by the language translators. This file is used only when specifically referenced in the control stream. It's never used as a default input or output file.

The file identifier is YSRC.

- System Job Control Stream (JCS) Library File

  This file provides for the permanent storage of control streams and JPROCS. It's used as the default output file by the file symbiont and as the default input file by the run symbiont.

  The file identifier is YJCS.

Volume Table of Contents

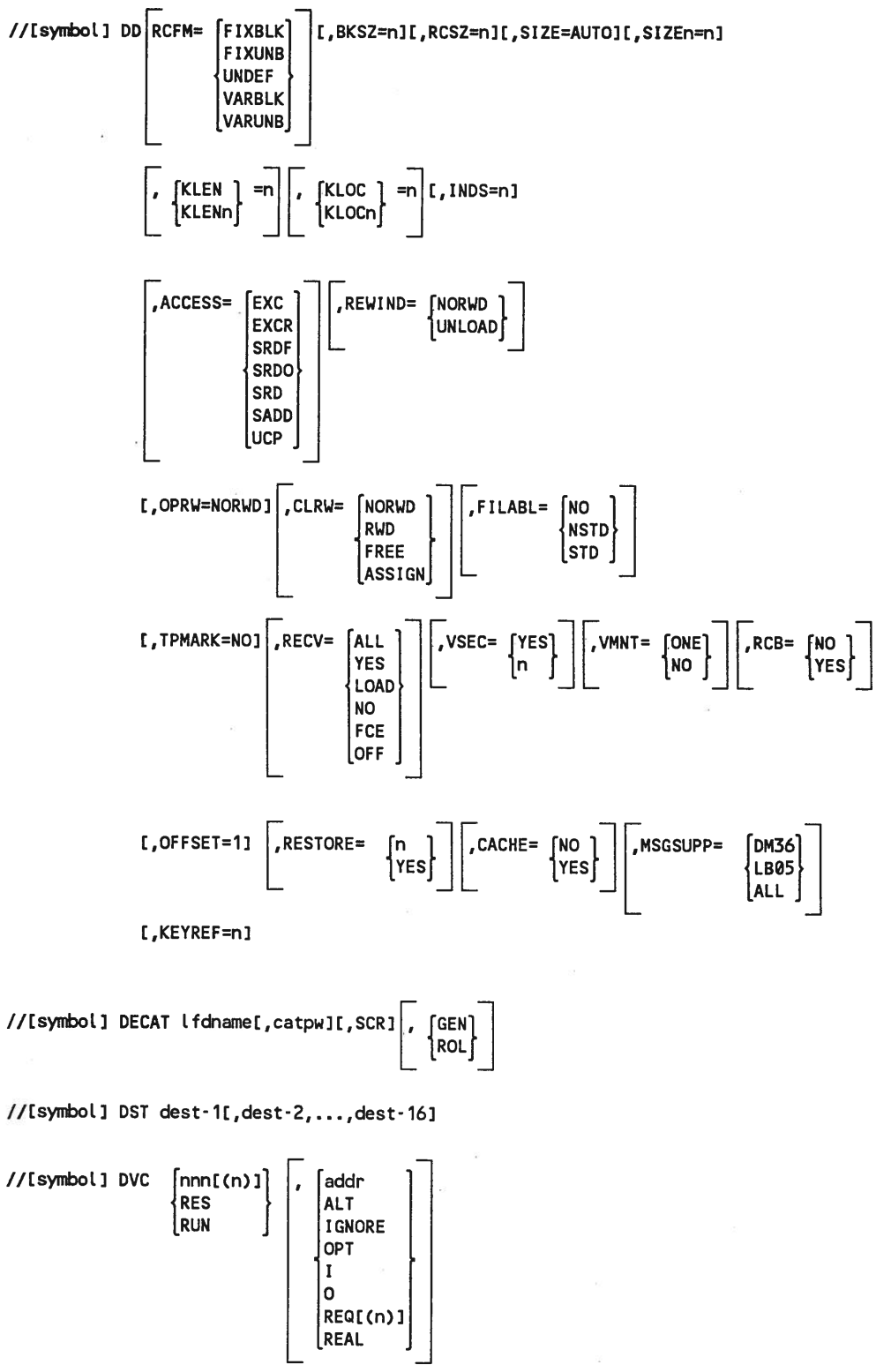
For each file on a direct access volume, there exists a set of control blocks in the VTOC area of the volume. Each set indicates the attributes and extents of the file, and may contain up to two control blocks. The information contained in these blocks is used by data management to control access to files. In case of multivolume files, there is a set of control blocks for the file in the VTOC of each volume.

For a complete description of these control blocks, see the *Consolidated Data Management Macroinstructions Programming Guide*, 7004 4607.

Appendix C
Job Control Statement Formats

Job Control Statements

```
//[symbol] ALTER [phase-name][,address][,change][, {RESET}]
                                                 {ORG  }

//[symbol] ALTJCS [file-label-id][, {RES       }][,rpw][, {FREE}][,LUN=nnn]
                                   {RUN       }         {ONLY}
                                   {vol-ser-no}         {OFF }
                                                        {ON  }

//[symbol] CAT lfdname[,catpw][,SCR][,GEN=nn]

//[symbol] CC  {command                  }
               {'command and parameters'}

//[symbol] CR

// DATA FILEID=file-identifier[,RETAIN][,IGNORE]

// DATA STEP=nnn
```


//[symbol] DD [RCFM= [FIXBLK / FIXUNB / UNDEF / VARBLK / VARUNB]] [,BKSZ=n][,RCSZ=n][,SIZE=AUTO][,SIZEn=n]
[, [KLEN / KLENn] =n] [, [KLOC / KLOCn] =n] [,INDS=n]
[,ACCESS= [EXC / EXCR / SRDF / SRDO / SRD / SADD / UCP]] [,REWIND= [NORWD / UNLOAD]]
[,OPRW=NORWD] [,CLRW= [NORWD / RWD / FREE / ASSIGN]] [,FILABL= [NO / NSTD / STD]]
[,TPMARK=NO] [,RECV= [ALL / YES / LOAD / NO / FCE / OFF]] [,VSEC= [YES / n]] [,VMNT= [ONE / NO]] [,RCB= [NO / YES]]
[,OFFSET=1] [,RESTORE= [n / YES]] [,CACHE= [NO / YES]] [,MSGSUPP= [DM36 / LB05 / ALL]]
[,KEYREF=n]
//[symbol] DECAT lfdname[,catpw][,SCR] [, [GEN / ROL]]
//[symbol] DST dest-1[,dest-2,...,dest-16]
//[symbol] DVC [nnn[(n)] / RES / RUN] [, [addr / ALT / IGNORE / OPT / I / O / REQ[(n)] / REAL]]

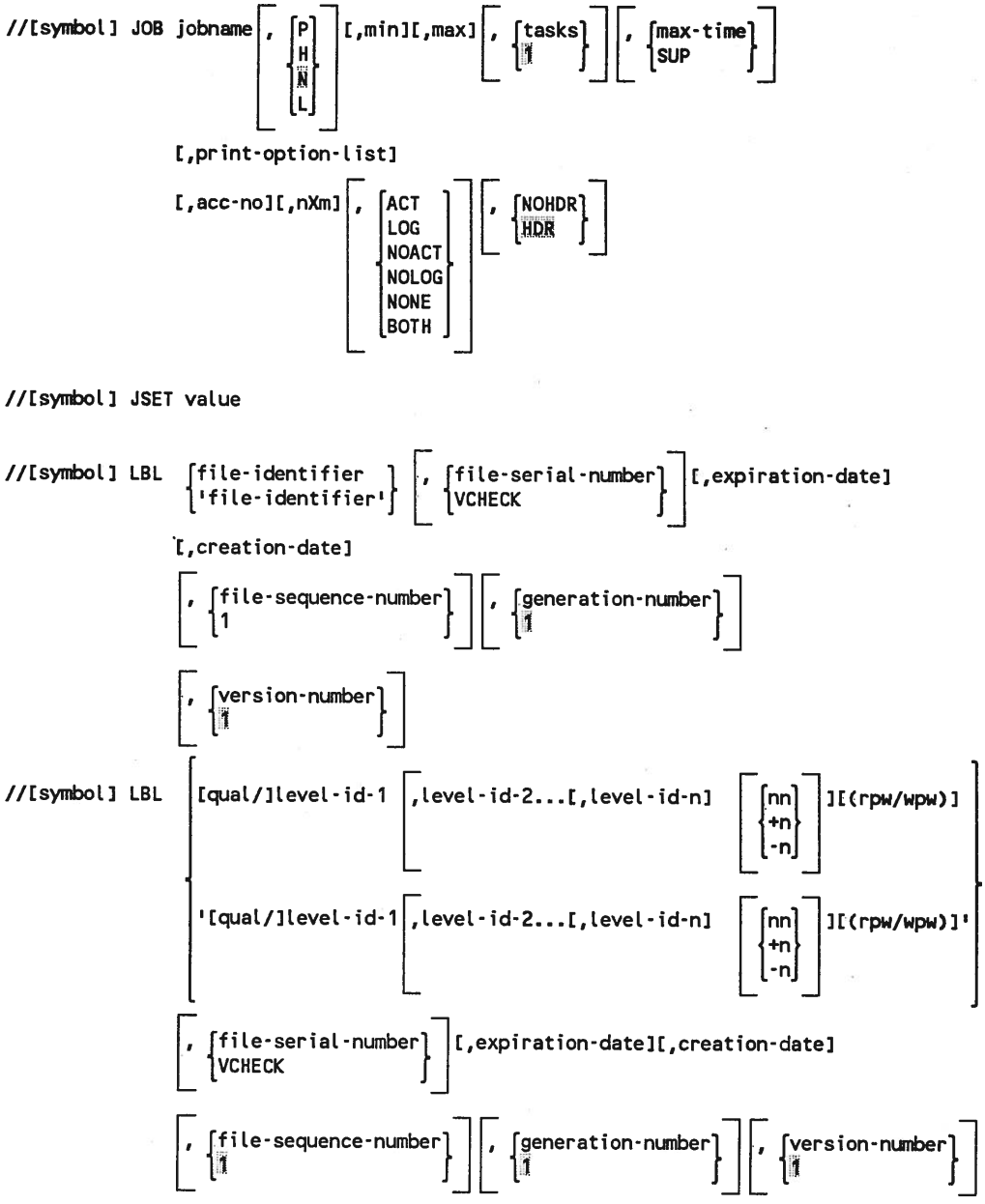
```
//[symbol] DVC PROG[,job-name][,HOST=host-id]

//[symbol] EQU lun-1,type-1[,lun-2,type-2,...,lun-n,type-n]

//[symbol] EXEC program-name[, {library-name | $Y$RUN | $Y$LOD}][,[±]switch-priority][,ABNORM=label]
```

EXT (For disk and format-label diskette)

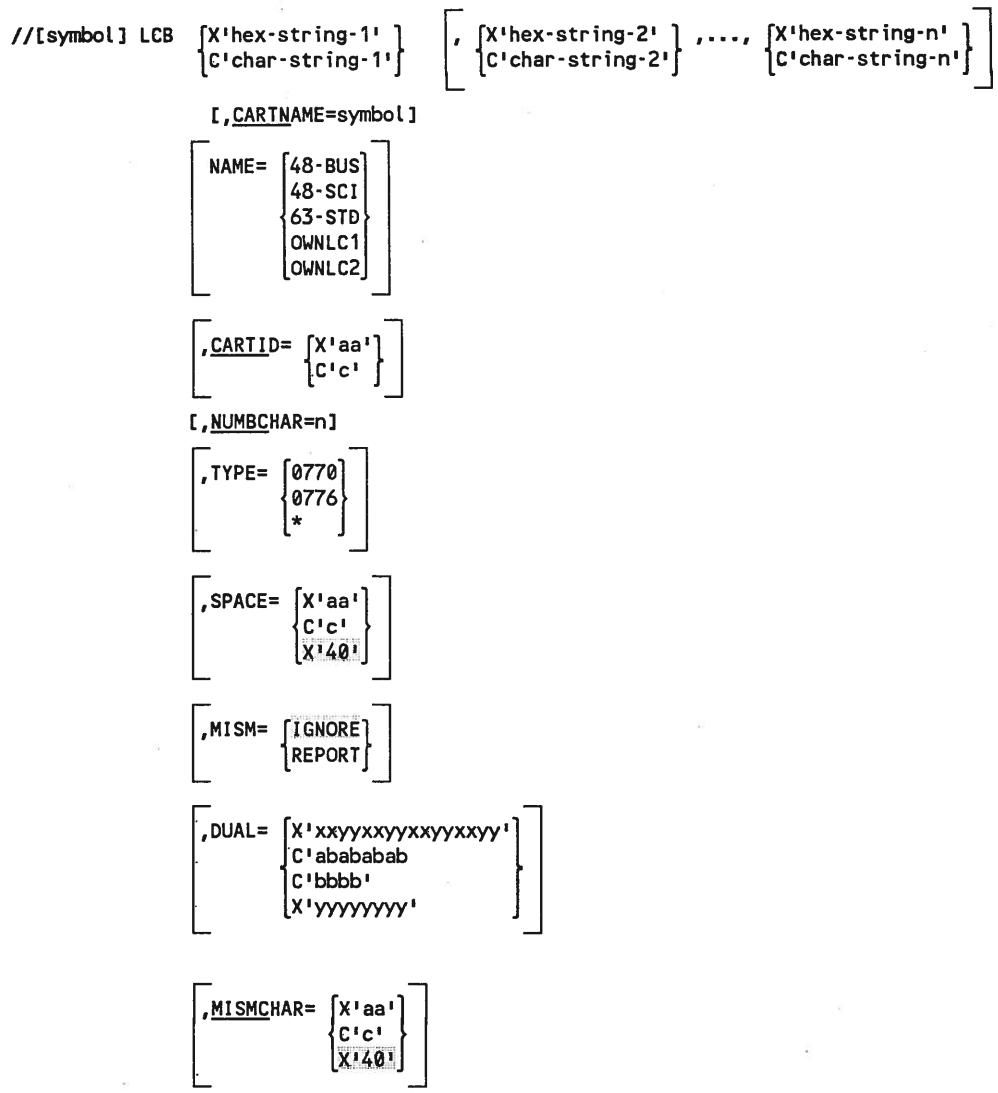
```
//[symbol] EXT {MI | ST}[, {C | CF | F}][, {inc | 0 | 1}][, {addr | Tccc:hh | BLK | TBLK | CYL | TRK | OLD}][, {mi | (bi,ai)}]

         [, {mj | (bj,aj)} ,...][,OLD][,FIX][,NTERM]
```

EXT (For data-set-label diskette)

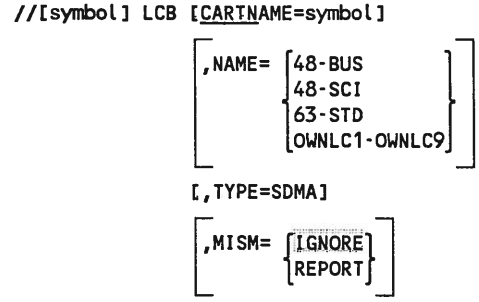
```
//[symbol] EXT MI,C,0,BLK,(bi,ai)[,NDI]

//[symbol] FIN

//[symbol] FREE lfdname-1 [[(DEV)],...,lfdname-n[(DEV)]]

//[symbol] GBL set-id-1[=init-1],set-id-2[=init-2],...,set-id-n[=init-n]]

//[symbol] GO destination

//[symbol] IF (a op b) destination

// [symbol] INQ JOB, keyword

// [symbol] INQ SYS, keyword

//[symbol] JNOTE comment-line[,destination-1,...,destination-n]
```


//[symbol] JOB jobname [, P | H | N | L] [,min][,max] [, tasks | 1] [, max-time | SUP]
[,print-option-list]
[,acc-no][,nXm] [, ACT | LOG | NOACT | NOLOG | NONE | BOTH] [, NOHDR | HDR]
//[symbol] JSET value
//[symbol] LBL file-identifier | 'file-identifier' [, file-serial-number | VCHECK] [,expiration-date]
[,creation-date]
[, file-sequence-number | 1] [, generation-number | 1]
[, version-number | 1]
//[symbol] LBL [qual/]level-id-1 [,level-id-2...[,level-id-n] [nn | +n | -n]][(rpw/wpw)]
'[qual/]level-id-1[,level-id-2...[,level-id-n] [nn | +n | -n]][(rpw/wpw)]'
[, file-serial-number | VCHECK] [,expiration-date][,creation-date]
[, file-sequence-number | 1] [, generation-number | 1] [, version-number | 1]

LCB (For Non-SDMA printers)



LCB (For SDMA and AP9215-1 Printers)



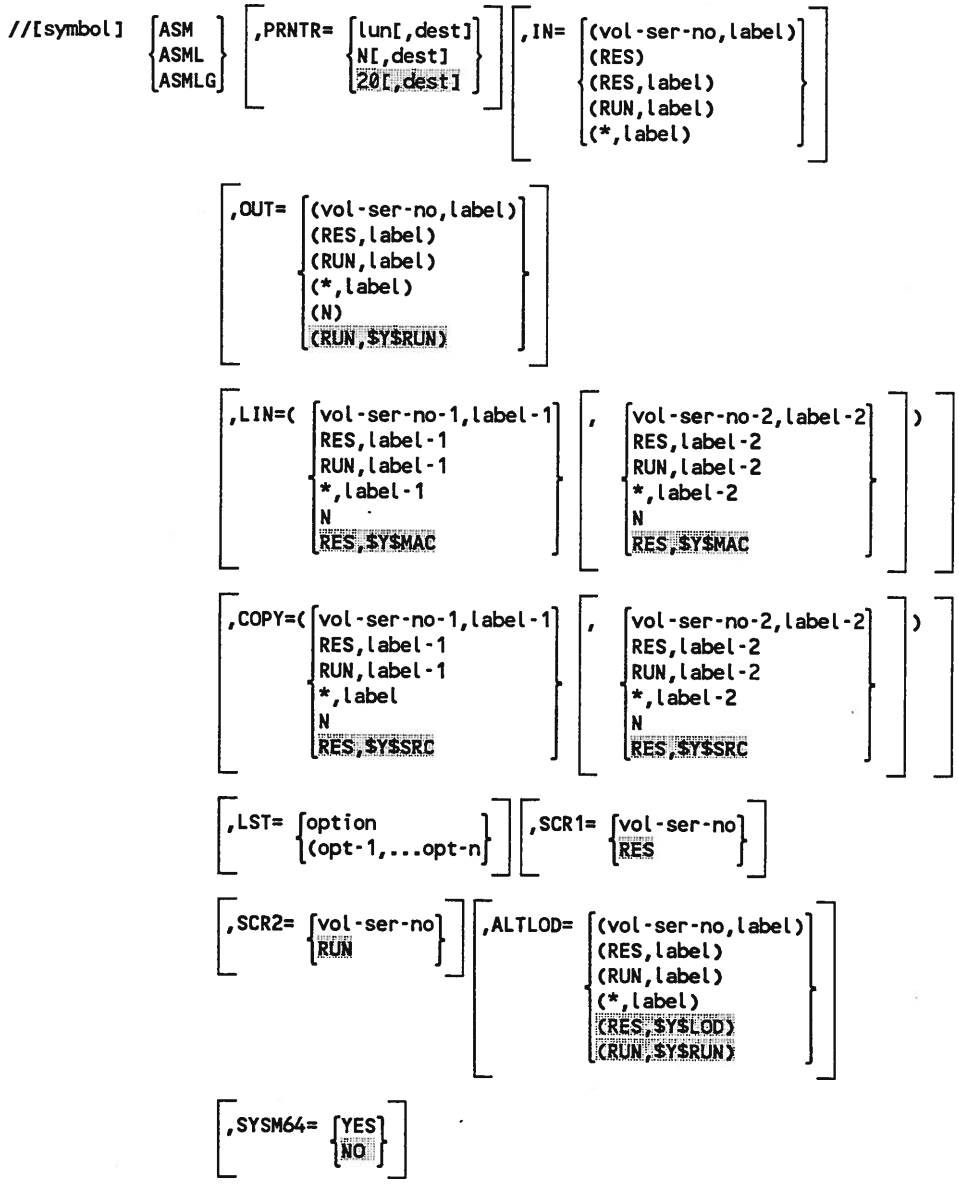
```
//[symbol] LFD  {filename  } [, [n]] [, [EXTEND]]
               {*filename }  [, [8]]  [, {INIT  }]
                                        {PREP  }
                                        {ID    }
                                        {IGNORE}

//[symbol] MTC lfdname, {BB,nn}
                        {BM,nn}
                        {FB,nn}
                        {FM,nn}
                        {WM,nn}
                        {RL   }
                        {RU   }

//[symbol] NOP [QUERY]

//[symbol] OPR comment-line[,destination-1,...,destination-n]

//[symbol] OPTION p-1[,...,p-n]

//[symbol] PARAM operand-1[,...,operand-n]

//[symbol] PAUSE comment-line[,destination-1,...,destination-n]

//[symbol] QGBL set-id-1[=init-1][,set-id-2[=init-2],...,set-id-n[=init-n]]

//[symbol] QUAL [qualname]

//[symbol] REN lfdname, {new-label  } [,NTERM]
                        {'new-label'}

//[symbol] ROUTE destination-1,...,destination-8

//[symbol] RST filename,checkpoint-id,step-number[,jobname[(rename)]][,pri]
             [,key-1=val-1,...,key-n=val-n]

//[symbol] {RUN [{jobname[(new-name)]}]}
           {    {(new-name)         }  }
           {RV jobname[(new-name)]     }

          [{:alt-filename                                  }]
          [{:(alt-filename, {RES})                         }]
          [{                {RUN}                          }]
          [{                {vsn}                          }]
          [{:(alt-filename, [{RES}] ,read-password)        }]
          [{                [{RUN}]                        }]
          [{                [{vsn}]                        }]

          [, {PRE }] [{time  }] [,key-1=val-1,...,key-n=val-n]
          [  {HIGH}] [{time+n}]
          [  {NOR }]
          [  {LOW }]
```

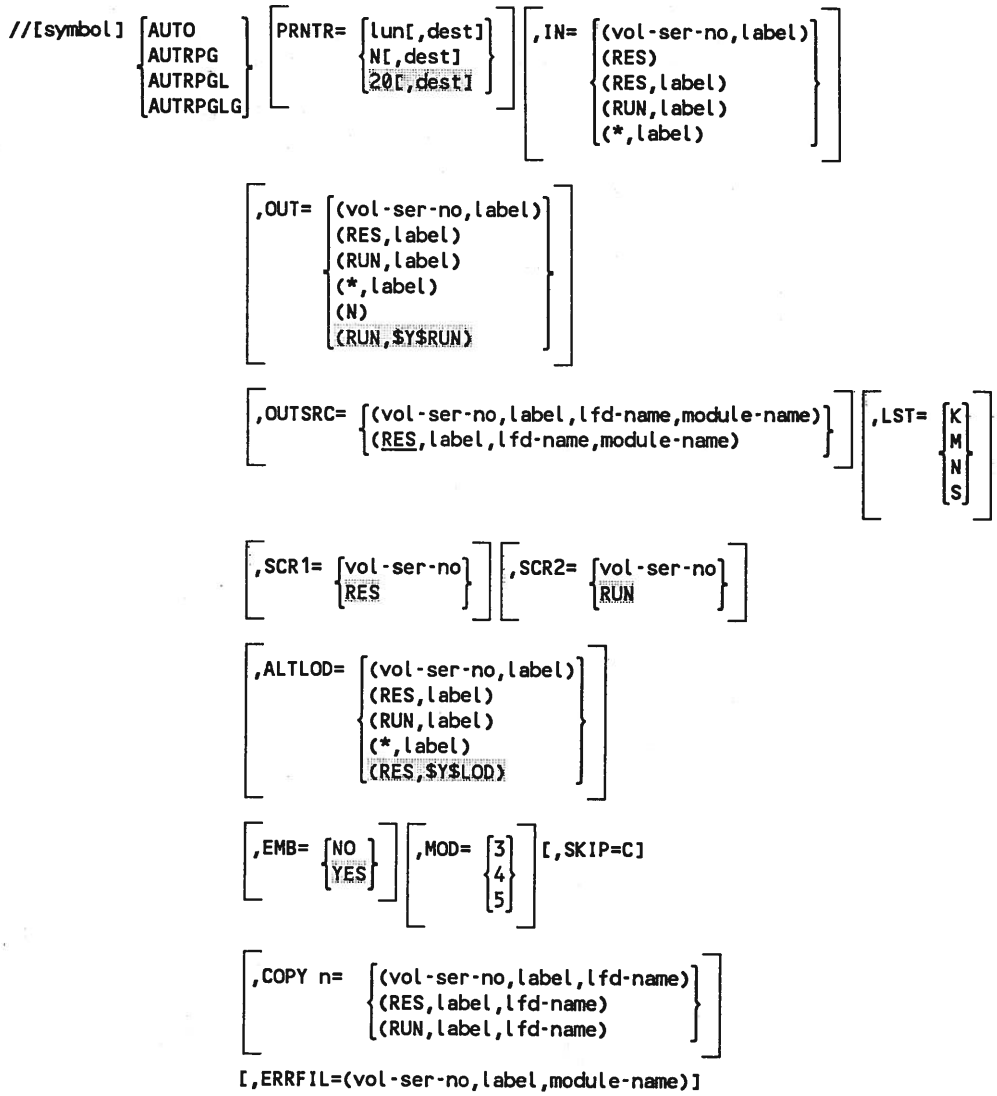
```
//[symbol] SCR lfdname[, {DATE[,yyddd]}]
                       {PRE[,aaaa]   }

//[symbol] SET COMREG,char-string

//[symbol] SET DATE,yy/mm/dd[,t-date][,d-date]

//[symbol] SET UPSI,switch-setting

//[symbol] SFT  {module-1[,...,module-n)][,DLOAD=[([calls],[{expansion-limit}])  ]]}
              {                                        {MAX             }        }
              {DLOAD=[ ([calls],[ {expansion-limit}])                            ]}
              {                   {MAX             }                             }

//[symbol] SKIP target-label[,mask[, {ALL }]]
                                     {ANY }
                                     {NONE}

// [symbol] SET LDA,n,m, {string }
                         {'string'}

//[symbol] SPL[  {HOLD   }][,nXm][, {no-cop}][, {no-skpcode}][, {max-rec}][,forms]
              {RETAIN  }          {1       }   {7            }   {5120   }
              {DISK    }                                        {0      }
              {TAPE    }
              {DISKETTE}]

             [, {NOHDR}][,NOTSTL][,brk-pge] [,NOUPD][,NOCMP][,RETAIN][,HOLD][,SECURE]
                {HDR  }                     [,UPD ]

//[symbol]UID {user-id-1          }[,..., {user-id-255          }]
              {(addr-1)             }      {(addr-255)           }
              {user-id-1(addr-1)    }      {user-id-255(addr-255)}

//[symbol]USE DP,dialog-name[,printer-lfd][,new-audit-lfd][,old-audit-lfd]

//[symbol] USE LIB,module-name[,TYPE={L}]
                                     {S}
                                     {M}
                                     {O}
```

```
//[symbol] USE MENU [ , {menu-file-LFD/$Y$FMT} ] [,initial-menu] [ , {nnn} ]
                        {$Y$FMT/menu-file-LFD}                      {1  }
                        {$Y$FMT              }

                   [,menu-format-1=alias-1[,...,menu-format-12=alias-12]]

//[symbol]USE SFS [ , {[format-file-LFD-1/[format-file-LFD-2]} ]
                      {format-file-LFD                       }
                      {$Y$FMT                                }

               [,initial-screen] [ , {nnn} ]
                                     {1  }

               [,screen-format-1=alias-1[,...,screen-format-12=alias-12]

// [symbol] VFB  LENGTH=lines
                 [,FORMNAME=symbol]

                 [ ,USE= {STAND1       } ]
                 [       {OWNVF1       } ]
                 [       {OWNVF2-OWNVF9} ]     (SDMA printers only)

                 [ ,DENSITY= {6} ]
                 [           {8} ]

                 [ ,TYPE= {0770} ] [,OVF=(line-1,...,line-n)]
                 [        {0776} ]
                 [        {SDMA} ]
                 [        {*   } ]

                 [,OVF2=(line-1,...,line-n)][,CD1=(line-1,...,line-n),...]
                 [,CD15=(line-1,...,line-n)]
                 [,HP=n]

//[symbol]VOL {Mcc                          } [ , {volsn-1 [ {(S)   } ]} ] [ , {volsn-2 [ {S)    } ]} ] ,...
              {N                            } [   {        [ {(NS)  } ]} ] [   {        [ {(NS)  } ]} ]
              {NMcc                         } [   {        [ {(NOV) } ]} ] [   {        [ {(NOV) } ]} ]
              {volsn-1 [ {(S)   } ]         } [   {        [ {(PREP)} ]} ] [   {        [ {(PREP)} ]} ]
              {        [ {(NS)  } ]         } [   {volsn-2 [ {(S)   } ]} ] [   {volsn-3 [ {S)    } ]} ]
              {        [ {(NOV) } ]         } [   {        [ {(NS)  } ]} ] [   {        [ {(NS)  } ]} ]
              {        [ {(PREP)} ]         } [   {        [ {(NOV) } ]} ] [   {        [ {(NOV) } ]} ]
              {  SCRATCH                    } [   {        [ {(PREP)} ]} ] [   {        [ {(PREP)} ]} ]
              {                             } [   {SCRATCH             } ] [   {SCRATCH             } ]

/$

/*

/&
```

Job Control Procedures

```
//[symbol] procname [p1,p2,...,pn,ki=vi,kj=vj,...,km=vm]

//lfdname ACCESS  {lblname                                    }  [ , {DVC=nn,VOL= {volsn} } ]
                  {(lblname [ , {n} ] [ , {EXTEND} ] )         }  [   {            {RUN  } } ]
                  {             {8}       {INIT  }             }  [   {            {*    } } ]
                                                                  [   {VOL= {volsn}        } ]
                                                                  [   {     {RUN  }        } ]
                                                                  [   {     {*    }        } ]

//lfdname ALLOC   {lblname                                    }  [ , {DVC=nn,VOL= {volsn} } ]
                  {(lblname [ , {n} ] [ , {EXTEND} ] )         }  [   {            {RUN  } } ]
                  {             {8}       {INIT  }             }  [   {            {*    } } ]
                                                                  [   {VOL= {volsn}        } ]
                                                                  [   {     {RUN  }        } ]
                                                                  [   {     {*    }        } ]

   [,EXT= ( [{ST}] [ , {C }] [ , {inc}] [ , {addr    }] [ , {mi     }]                  ]
   [        [{MI}]     {CF}      {0  }       {Tccc:hh }       {(bi,ai)}                   ]
   [                   {F }      {1  }       {BLK     }                                   ]
   [                                         {TBLK    }                                   ]
   [                                         {CYL     }                                   ]
   [                                         {TRK     }                                   ]
   [                                         {OLD     }                                   ]
   [                                                                                      ]
   [          [ , {mj     } ,...][,OLD][,FIX][,NDI])                                      ]
   [              {(bj,aj)}                                                               ]
```


//[symbol] [ASM / ASML / ASMLG]
[,PRNTR= [lun[,dest] / N[,dest] / 20[,dest]]]
[,IN= [(vol-ser-no,label) / (RES) / (RES,label) / (RUN,label) / (*,label)]]
[,OUT= [(vol-ser-no,label) / (RES,label) / (RUN,label) / (*,label) / (N) / (RUN,YRUN)]]
[,LIN=([vol-ser-no-1,label-1 / RES,label-1 / RUN,label-1 / *,label-1 / N / RES,YMAC] [, [vol-ser-no-2,label-2 / RES,label-2 / RUN,label-2 / *,label-2 / N / RES,YMAC]])]
[,COPY=([vol-ser-no-1,label-1 / RES,label-1 / RUN,label-1 / *,label / N / RES,YSRC] [, [vol-ser-no-2,label-2 / RES,label-2 / RUN,label-2 / *,label-2 / N / RES,YSRC]])]
[,LST= [option / (opt-1,...opt-n)]]
[,SCR1= [vol-ser-no / RES]]
[,SCR2= [vol-ser-no / RUN]]
[,ALTLOD= [(vol-ser-no,label) / (RES,label) / (RUN,label) / (*,label) / (RES,YLOD) / (RUN,YRUN)]]
[,SYSM64= [YES / NO]]

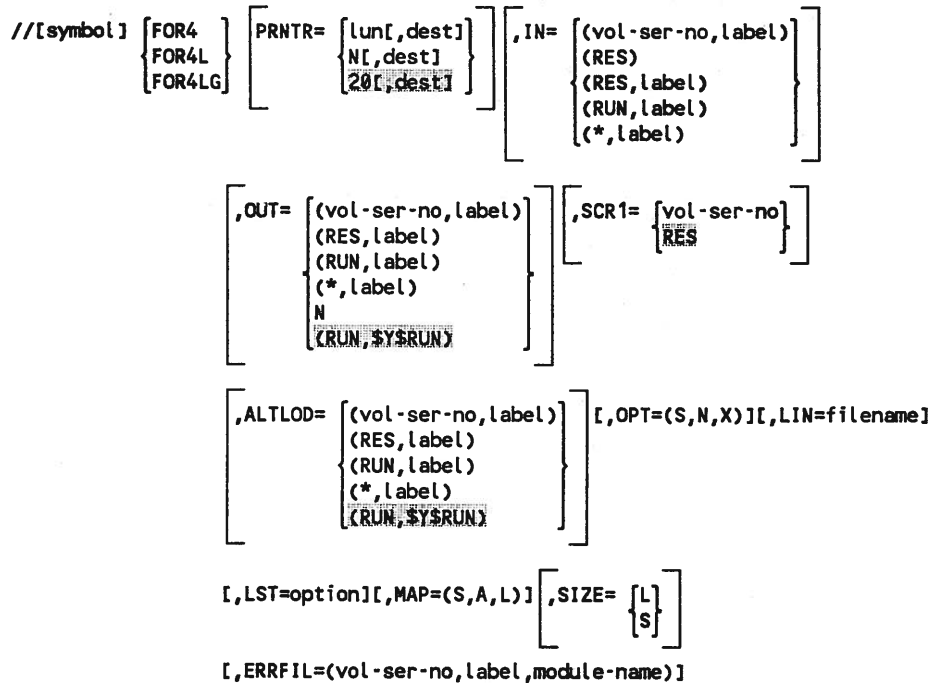

//[symbol] [AUTO | AUTRPG | AUTRPGL | AUTRPGLG]
[PRNTR= [lun[,dest] | N[,dest] | 20[,dest]]]
[,IN= [(vol-ser-no,label) | (RES) | (RES,label) | (RUN,label) | (*,label)]]
[,OUT= [(vol-ser-no,label) | (RES,label) | (RUN,label) | (*,label) | (N) | (RUN,YRUN)]]
[,OUTSRC= [(vol-ser-no,label,lfd-name,module-name) | (RES,label,lfd-name,module-name)]]
[,LST= [K | M | N | S]]
[,SCR1= [vol-ser-no | RES]]
[,SCR2= [vol-ser-no | RUN]]
[,ALTLOD= [(vol-ser-no,label) | (RES,label) | (RUN,label) | (*,label) | (RES,YLOD)]]
[,EMB= [NO | YES]]
[,MOD= [3 | 4 | 5]]
[,SKIP=C]
[,COPY n= [(vol-ser-no,label,lfd-name) | (RES,label,lfd-name) | (RUN,label,lfd-name)]]
[,ERRFIL=(vol-ser-no,label,module-name)]

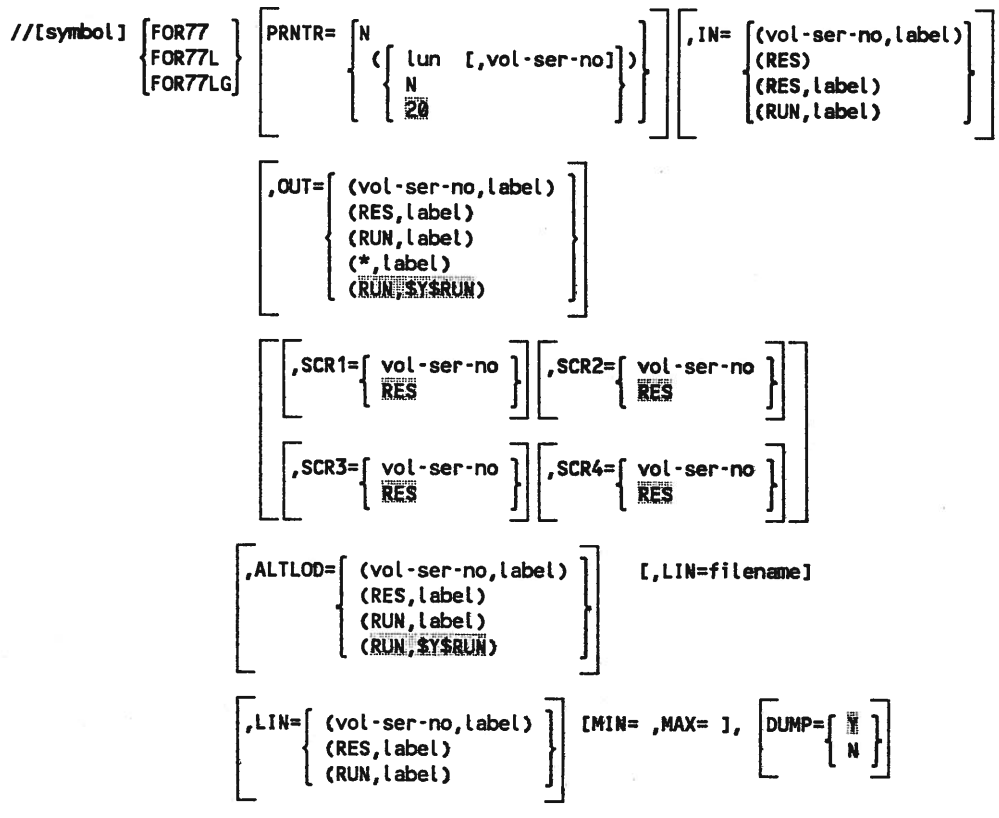
```
//symbol {CCOMP | CCOMPL | CCOMPLG} IN=(label,vol-ser-no)
                    [PRNTR=({N|lun|20})] [,LIB1=(label,vol-ser-no)]
                    [,LIB2=(label,vol-ser-no)] [,LIBC=(label,vol-ser-no)]
                    [,OUT=({label,vol-ser-no|N})] [,DEBUG={N|NO|Y|YES}]
                    [,DEFINE=(define_symbol[=initial_value]{, ... })]
                    [,INTERACT={N|NO}] [,DUMP={N|NO}] [,LIST={N|NO|Y|YES}]
                    [,SOURCE={N|NO|Y|YES}] [,NRENT={N|NO|Y|YES}]
                    [,ALTLOD=(label,vol-ser-no)] [,MAIN={N|NO|Y|YES}]
                    [,PP={N|NO|Y|YES}]

//[symbol] {COBL74  } [PRNTR= {lun[,dest]}]
           {COBL74L }         {N[,dest]  }
           {COBL74LG}         {20[,dest] }

                    [,IN= {(vol-ser-no,label)}]
                          {(RES)             }
                          {(RES,label)       }
                          {(RUN,label)       }
                          {(*,label)         }

                    [,LIN=( {(vol-ser-no,label)}) ] [,LINn=({vol-ser-no,label}) ]
                           {(RES,label)         }          {RES,label       }
                           {(RUN,label)         }          {RUN,label       }
                           {(*,label)           }          {*,label         }
                           {(RES,$Y$SRC)        }

                    [,OBJ= {(vol-ser-no,label)}] [,SCR1= {vol-ser-no}]
                           {(RES,label)       }          {RES       }
                           {(RUN,label)       }
                           {(RUN,$Y$RUN)      }
                           {(*,label)         }

                    [,SCR2= {vol-ser-no}] [,SCR3= {vol-ser-no}]
                            {RES       }          {RUN       }

                    [,ALTLOD= {(vol-ser-no,label)}] [,option=specification]
                              {(RES,label)       }
                              {(RUN,label)       }
                              {(*,label)         }
                              {(RES,$Y$LOD)      }
                              {(RUN,$Y$RUN)      }

                    [,ERRFIL=(vol-ser-no,label,module-name)]
```

```
//[symbol] { COBL85  } PRNTR= { lun[,vol-ser-no] }
           { COBL85L }       { N                 }
           { COBL85LG}       { 20                }

     ,IN=  { (vol-ser-no,label) } ,LIN= { (vol-ser-no,label) }
           { (RES)              }       { (RES,label)       }
           { (RES,label)        }       { (RUN,label)       }
           { (RUN,label)        }       { (*, label)        }
           { (*,label)          }       { (RES,$Y$SRC)      }

     ,LINn= { (vol-ser-no,label) } ... ,OBJ= { (vol-ser-no,label) }
            { (RES,label)        }          { (RES,label)       }
            { (RUN,label)        }          { (RUN,label)       }
            { (*,label)          }          { (*,label)         }
                                            { (RUN,$Y$RUN)      }

     ,SCR1= { vol-ser-no } ,SCR2= { vol-ser-no } ,SCR3= { vol-ser-no }
            { RES        }       { RES        }       { RES        }

     ,ALTLOD= { (vol-ser-no,label) } [,option=specification]
              { (RES,label)       }
              { (RUN,label)       }
              { (*,label)         }
              { (RES,$Y$LOD)      }
              { (RES,$Y$RUN)      }
```

```
//[symbol] { COBOLB  } [PRNTR= { lun[,dest] }] [,IN= { (vol-ser-no,label) }]
           { COBOLBL }        { N[,dest]    }       { (RES)              }
           { COBOLBLG}        { 20[,dest]   }       { (RES,label)        }
           { COBOL   }                              { (RUN,label)        }
           { COBOLL  }                              { (*,label)          }
           { COBOLLG }

                [,OBJ= { (vol-ser-no,label) }] [,LIN= { (vol-ser-no,label) }]
                       { (RES,label)        }        { (RES,label)       }
                       { (RUN,label)        }        { (RUN,label)       }
                       { (*,label)          }        { (*,label)         }
                       { (RUN,$Y$RUN)       }        { (RES,$Y$SRC)      }

     [,OUT=(p-1[,...,p-n)][,LST=(p-1,...,p-n)]

     [,SCR1= { vol-ser-no }] [,SCR2= { vol-ser-no }]
             { RES        }          { RES        }

     [,SCR3= { vol-ser-no }] [,ALTLOD= { (vol-ser-no,label) }]
             { RUN        }            { (RES,label)       }
                                       { (RUN,label)       }
                                       { (*,label)         }
                                       { (RES,$Y$LOD)      }
                                       { (RUN,$Y$RUN)      }
```

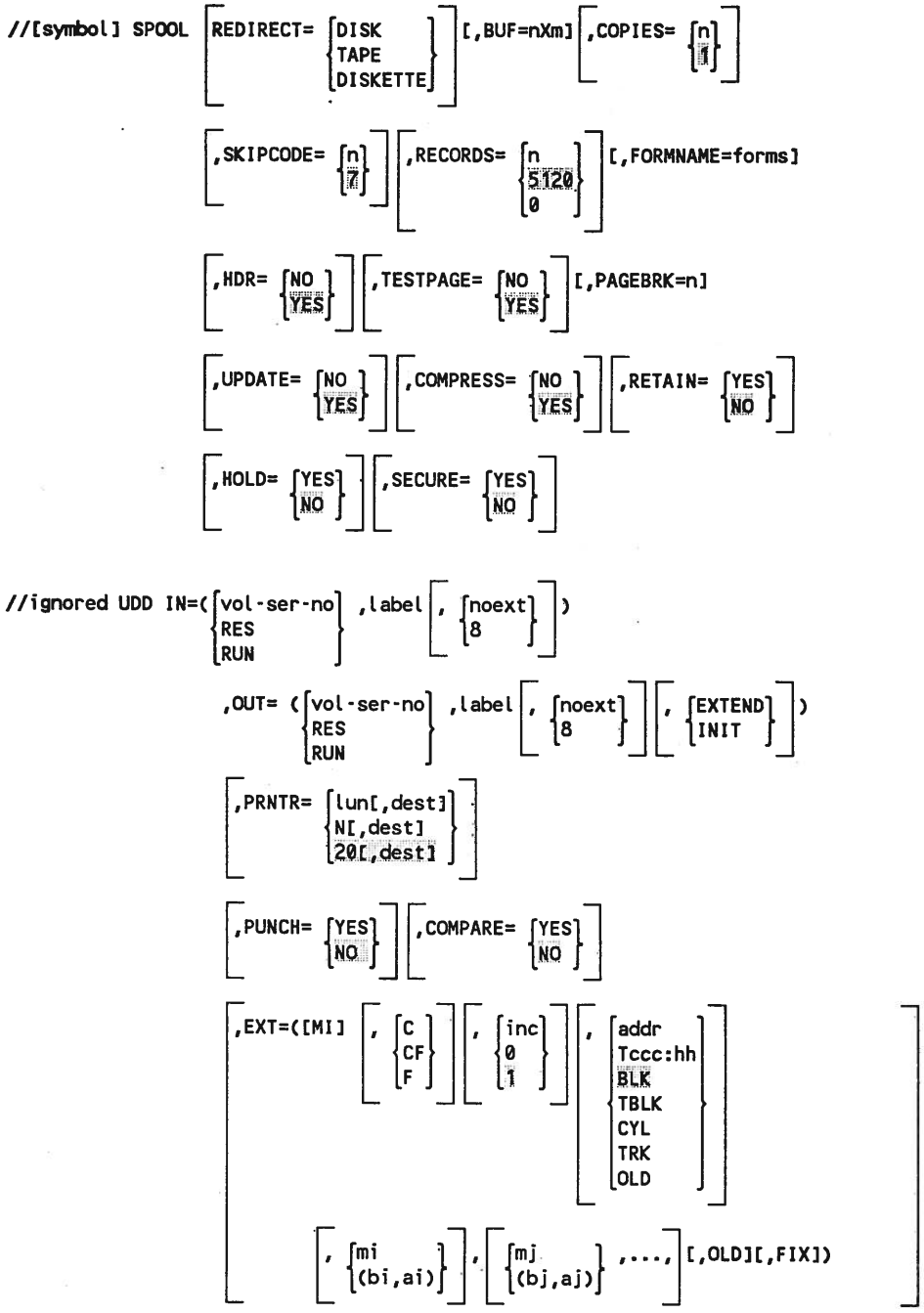
```
//[symbol] DVCDKT vol-ser-no[,lun] [,NOVOL= {Y}]
                                         {N}

//[symbol] DVCVOL {vol-ser-no} [,lun] [,NOVOL= {Y}]
                 {RES       }              {N}
                 {RUN       }

//[symbol] DVCVTP vol-ser-no[,lun] [,PREP= {Y}] [,NOVOL= {Y}]
                                         {N}           {N}

//[symbol]  {FORT  }  [PRNTR= {lun[,dest]}]  [,IN= {(vol-ser-no,label)}]
            {FORTL }         {N[,dest]  }         {(RES)            }
            {FORTLG}         {20[,dest] }         {(RES,label)      }
            {FOR   }                              {(RUN,label)      }
            {FORL  }                              {(*,label)        }
            {FORLG }
            {FOR4  }
            {FOR4L }
            {FOR4LG}

            [,OUT= {(vol-ser-no,label)}]
                   {(RES,label)      }
                   {(RUN,label)      }
                   {(*,label)        }
                   {NO               }
                   {(RUN,$Y$RUN)     }

            [,SCR1= {vol-ser-no}]  [,ALTLOD= {(vol-ser-no,label)}]
                    {RES       }             {(RES,label)      }
                                             {(RUN,label)      }
                                             {(*,label)        }
                                             {(RUN,$Y$RUN)     }

            [,OPT=(D,N,X)][,MDE=I][,STX=options]

            [,CNL=k] [,LIN= {filename}]  [,LST= {k     }]
                            {lib1    }           {option}

            [,MAP=(S,A,L)] [,SIZE= {L}]
                                   {S}

            [,ERRFIL=(vol-ser-no,label,module-name)]
```

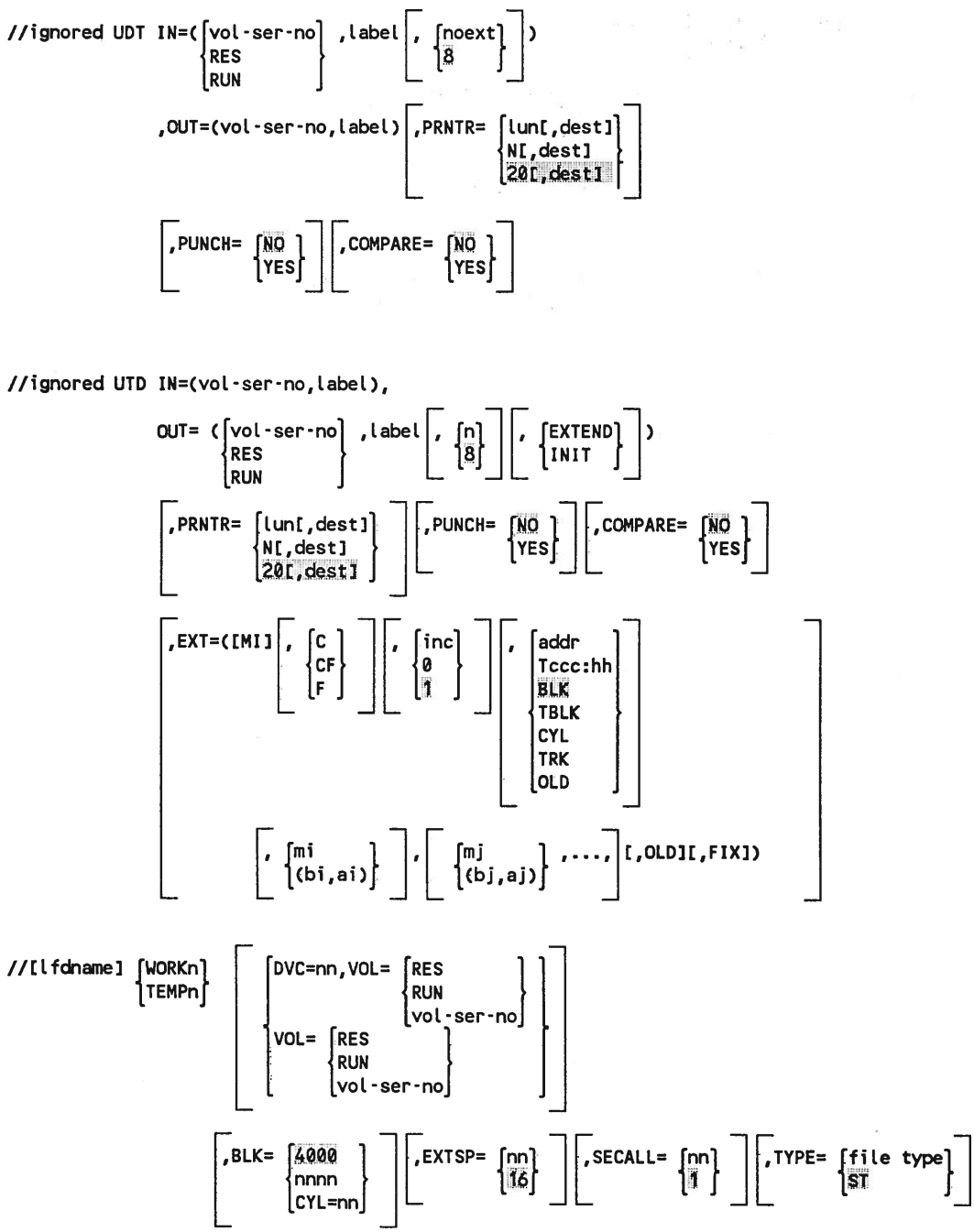

//[symbol] [FOR4 FOR4L FOR4LG] [PRNTR= [lun[,dest] N[,dest] 20[,dest]]] [,IN= [(vol-ser-no,label) (RES) (RES,label) (RUN,label) (*,label)]]
[,OUT= [(vol-ser-no,label) (RES,label) (RUN,label) (*,label) N (RUN,YRUN)]] [,SCR1= [vol-ser-no RES]]
[,ALTLOD= [(vol-ser-no,label) (RES,label) (RUN,label) (*,label) (RUN,YRUN)]] [,OPT=(S,N,X)][,LIN=filename]
[,LST=option][,MAP=(S,A,L)] [,SIZE= [L S]]
[,ERRFIL=(vol-ser-no,label,module-name)]


//[symbol] [FOR77 FOR77L FOR77LG] [PRNTR= [N (lun [,vol-ser-no]) N 20]] [,IN= [(vol-ser-no,label) (RES) (RES,label) (RUN,label)]]
[,OUT= [(vol-ser-no,label) (RES,label) (RUN,label) (*,label) (RUN,YRUN)]]
[,SCR1= [vol-ser-no RES]] [,SCR2= [vol-ser-no RES]]
[,SCR3= [vol-ser-no RES]] [,SCR4= [vol-ser-no RES]]
[,ALTLOD= [(vol-ser-no,label) (RES,label) (RUN,label) (RUN,YRUN)]] [,LIN=filename]
[,LIN= [(vol-ser-no,label) (RES,label) (RUN,label)]] [MIN= ,MAX=], [DUMP= [Y N]]

```
//[symbol] {LINK } [input-module-name-1,...,input-module-name-10]
           {LINKG}
                  [,PRNTR= [lun[,dest]  ]]
                           {N[,dest]    }
                           [20[,dest]   ]

                  [,IN= [(vol-ser-no,label)]] [,OUT= [(vol-ser-no,label)]]
                        {(RES)              }        {(RES,label)        }
                        {(RES,label)        }        {(RUN,label)        }
                        {(RUN,label)        }        {(*,label)          }
                        {(*,label)          }        {(N)                }
                        [(RUN,$Y$RUN)       ]        [(RUN,$Y$RUN)       ]

                  [,RLIB= [(vol-ser-no,label)]
                          {(RES,label)       }
                          {(RUN,label)       }
                          [(*,label)         ]

                  [,ALIB= [(vol-ser-no,label)]] [,SCR1= [vol-ser-no]] [,STD= [YES]]
                          {(RES,label)       }          [RES       ]         [NO ]
                          {(RUN,label)       }
                          [(*,label)         ]

                  [,ALTLOD= [(vol-ser-no,label)]] [,OPT='options']
                            {(RES,label)       }
                            {(RUN,label)       }
                            {(*,label)         }
                            [(RUN,$Y$RUN)      ]

                  [,CLIB= [(vol-ser-no,label,modname)]] [,CMT='comment']
                          {(RES,label,modname)       }
                          {(RUN,label,modname)       }
                          [(*,label,modname)         ]

                  [,ENTER=expression]
```

```
// [symbol] PASCAL [PRNTR=N]
                    ,IN= [(vsn,label)]  [,LIN= [(vsn,label)]]
                         {(RES)       }        {(RES,label)}
                         {(RES,label) }        {(RUN,label)}
                         [(RUN,label)]
                    [,OBJ=[ (vsn,label)   ]] [,ALTLOD=[ (vsn,label)   ]]
                          { (RES,label)   }           { (RES,label)   }
                          { (RUN,label)   }           { (RUN,label)   }
                          { (*,label)     }           { (RES,$Y$LOD)  }
                          [ (RUN,$Y$RUN)  ]           [ (RES,$Y$RUN)  ]
                    [,OBJLST=[ YES ]] [,MEM=valueK] [,DUMP=[ YES ]]
                             [ NO  ]                       [ NO  ]

//[symbol] [RPG  ] [PRNTR= [lun[,dest]]] [,IN= [(vol-ser-no,label)]]
           {RPGL }          {N[,dest]  }       {(RES)            }
           [RPGLG]          [20[,dest] ]       {(RES,label)      }
                                               {(RUN,label)      }
                                               [(*,label)        ]
                    [,OUT= [(vol-ser-no,label)]] [,LST= [K]]
                           {(RES,label)       }         {M}
                           {(RUN,label)       }         {N}
                           {(*,label)         }         [S]
                           {(N)               }
                           [(RUN,$Y$RUN)      ]
                    [,SCR1= [vol-ser-no]] [,SCR2= [vol-ser-no]]
                            [RES       ]          [RUN       ]
                    [,ALTLOD= [(vol-ser-no,label)]
                              {(RES,label)       }
                              {(RUN,label)       }
                              {(*,label)         }
                              [(RUN,$Y$RUN)      ]
                    [,CONSOLE=lfd-name]
                    [,EMB= [NO ]] [,MOD= [3   ]] [,COL=7]
                           [YES]         {4   }
                                         {5   }
                                         [IRAM]
                    [,ERRFIL=(vol-ser-no,label,module-name)]
                    [,MIRAM=ALL] [,UNPKDS=[YES]] [,FNAME8=[YES]]
                                          [NO ]          [NO ]
```


//[symbol] SPOOL
REDIRECT= DISK / TAPE / DISKETTE
[,BUF=nXm]
,COPIES= n / 1
,SKIPCODE= n / 7
,RECORDS= n / 5120 / 0
[,FORMNAME=forms]
,HDR= NO / YES
,TESTPAGE= NO / YES
[,PAGEBRK=n]
,UPDATE= NO / YES
,COMPRESS= NO / YES
,RETAIN= YES / NO
,HOLD= YES / NO
,SECURE= YES / NO
//ignored UDD IN=(vol-ser-no / RES / RUN ,label [, noext / 8])
,OUT= (vol-ser-no / RES / RUN ,label [, noext / 8] [, EXTEND / INIT])
,PRNTR= lun[,dest] / N[,dest] / 20[,dest]
,PUNCH= YES / NO
,COMPARE= YES / NO
,EXT=([MI] [, C / CF / F] [, inc / 0 / 1] [, addr / Tccc:hh / BLK / TBLK / CYL / TRK / OLD]
[, mi / (bi,ai)] [, mj / (bj,aj) ,...,] [,OLD][,FIX])


//ignored UDT IN=(vol-ser-no | RES | RUN) ,label [, noext | 8]
,OUT=(vol-ser-no,label) [,PRNTR= lun[,dest] | N[,dest] | 20[,dest]]
[,PUNCH= NO | YES] [,COMPARE= NO | YES]
//ignored UTD IN=(vol-ser-no,label),
OUT= (vol-ser-no | RES | RUN) ,label [, n | 8] [, EXTEND | INIT])
[,PRNTR= lun[,dest] | N[,dest] | 20[,dest]] [,PUNCH= NO | YES] [,COMPARE= NO | YES]
[,EXT=([MI] [, C | CF | F]) [, inc | 0 | 1] [, addr | Tccc:hh | BLK | TBLK | CYL | TRK | OLD]
[, mi | (bi,ai)] [, mj | (bj,aj) ,...,] [,OLD][,FIX])
//[lfdname] WORKn | TEMPn [DVC=nn,VOL= RES | RUN | vol-ser-no
VOL= RES | RUN | vol-ser-no]
[,BLK= 4000 | nnnn | CYL=nn] [,EXTSP= nn | 16] [,SECALL= nn | 1] [,TYPE= file type | ST]

```
//[symbol] [WRTBIG] 'block-1'[,'block-2',...,'block-8']
           [WRTSML]
                  [,IN=  [(vol-ser-no,label)]  ]
                         {(RES,label)          }
                         {(RUN,label)          }
                         {(*,label)            }
                         [(RES,$Y$LOD)         ]
                  [,LUN= { ([nnn] [, [lfd-name]] [,dest]) }
                         { ([20 ]      [PRNTR  ]         }
                         { ([N  ]                        }
```

Index

A

B

C

D

E

F

G

H

I

U

V

W

X

SYMBOLS