Reference Guide

JOL

Universal Command Language

Version 5.1

Open System Command And Retrieval (Oscar) Pty. Ltd.

Jol Universal Command Language "*Reference Guide"*.

Fifth Edition (June, 1999)

This is a major revision of, and obseletes, the
Fifth Edition of the Jol Command Language
"Reference Guide" September 1987

Open System Command and Retrieval (OSCAR) Pty. Ltd.,
45 Riversdale Road, Hawthorn, Victoria, Australia 3022.

Postal Address:
PO Box 475, Toorak, Vic, AUSTRALIA, 3142

Telephone	**(61)-3-9818-8351**
Telefax	**(61)-3-9819-2848**
Email	**oscarptyltd@ozemail.com.au**

Preface

The *Jol Reference Guide* provides a handy aid to the Jol programmer. It is a companion publication to the *Jol Reference Manual* and *General Information Manual* and it provides the syntax and a brief description of the functions of most of the instructions and commands provided with the Jol Command Language. For more detailed explanations of the instructions, refer to the *JOL Reference Manual*. This manual is intended for reference use at a terminal, and prior knowledge of Jol is assumed.

In conjunction with the *Jol General Information Manual*, it can be used by experienced Jol programmers to write programs in place of the full *Jol Reference Manual*.

Some instructions, such as DOSTOJOL, JCLTOJOL, Networking and Scheduling Instructions are not described in this Guide, but in other publications.

Supplemental publications include, but are not limited to:

- Jol Concepts and Facilities Manual
- Jol General Information Manual
- Answers to Questions about Jol
- Teach Yourself Jol Program Diskette for the IBM Personal Computer
- Jol Reference Manual
- Jol Messages and Codes
- Jol User Guide
- Jol Evaluation Plan and Financial Work Sheets
- Jol Conversion Utilities
- Jol System Programmer Guide
- Jol Release and SMP Guide
- Jol Program Logic Manual (Licensed Material)

Summary of Amendments

Scheduling and Networking Facilities

- Jobs may be defined as part of a network, and automatically submitted on certain days.

TSO Support.

- Options to allow the job to execute under TSO or Background, using the same Command Language.

ALLOCATE Command.

- **ALLOCATE** a data set in the Preprocessor or Macro Phase for Input or Output.

Addition of OPEN, READ and WRITE Instructions.

- **OPEN** a file for input or output.
- **READ** a record into a variable.
- **WRITE** a record from a variable.

Automatic Reset of Relative Generation Numbers

- Relative Generation Numbers are automatically reset for Reruns or Restarts.

Testing if a data set exists.

- **TEXIST** Command allows the testing of the existence of a data set at execution time.

Table of Contents

Introduction

Jol is a high-level, English-like, Universal Command Language. A Command Language is the highest level of communication between you and the computer. Command languages tell the computer what to do, when to do it, and what to do with theresult. Programming Languages, such as PL/I and COBOL, give the computer detailed instructions on how to do it.

Jol commands are coded in a free format English-like language similiar to TSO CLISTs, PL/I, Pascal and "C". These commands are then interpreted and run *immediately in foreground* (under TSO), or in a *background* region, either with or without JCL, or under full control of a monitor which uses Dynamic Allocation instead of JCL. The same command language can operate both in *foreground* and in *background* .

With Jol you can:

- Use Jol's extremely comprehensive Macro language to write your own Jol commands tailored specifically to your installation.

- Use simple **IF, THEN, ELSE, AND/OR,** and **DO** logic to test return codes, symbolic variables, calendar information, or **ABEND** situations.

- Define, initialize, test, branch and perform arithmetic on Symbolic Variables. Hence, you can create data set names, programs and instructions depending on symbolic variable or parameter information.

- **ALLOCATE** , **READ** and **WRITE** to data sets during the compilation. Therefore, you can access data, and dynamically alter your job depending on other data on your computer.

- Write full screen data entry panels.

In other words, Jol complements JCL, CLISTs and SPF in one easy to use and learn Command Language.

Procedural Programming

Jol programs are written in a procedural format that is already familiar to Programmers using programming languages such as COBOL, PASCAL or PL/I. The procedural format provides you the flexibility to solve the most complex type of requirements in a logical straightforward manner.

In **COBOL** or **PL/I** , you **DEFINE** or **DECLARE** variables and then use instructions to alter those variables. In Jol, you also **DEFINE** or **DECLARE** Data Set and Program details which *may* be required for your job and then you use simple instructions such as **RUN** and **SORT** to set the job in motion. In Jol:

- *Data Set* declarations specify *record-format, volume, unit, space* and other information.
- *Program* declarations specify whether the program's internal filenames (**DDNAMES**) *read* , *write* , *update* or *extend* the data set variables previously described.
- *Instructions* such as **RUN, IF ... THEN, COPY, SORT,** and **PRINT** manipulate these variables as you would variables in other high level languages. Other simple instructions are provided to **test** errors or conditions, **submit** jobs and so on.

Analysing Your Job

You tell Jol which data sets you require and describe the programs you wish to execute specifying whether they are toread, write, update or extend the data sets you have described. Later, you use simple instructions to run your programs, sort data sets,test conditions and so on. Jol examines in detail the instructions and declarations you have given, and automatically:

- Passes newly created or old data sets to following programs requiring them.
- Finds input data sets that are not created in your job, that is, data sets that are External to your job.
- Determines which data sets are used only to pass information from one program to another (temporary). Unless a **KEEP** or **CATALOG** instruction is used, *new* data sets are automatically deleted at the end of the job.
- Determines which data sets are used as work files, and organizes that the volumes containing or receiving these data sets will be mounted when your job executes

Error Detection

Before the Jol Compiler allows the job to execute, it runs detailed validation checks on the instructions and declarations that have been made available. Jol checks that no execution of an instruction will cause your job to fail at any stage, and that there are no data sets or libraries missing. If errors are detected the job is not allowed to execute (unless the **LET** option is specified). If the job was allowed to run, it would fail at that point having wasted whatever computer time was necessary to reach that stage. These types of error detection often save hours of otherwise wasted computer time.

Jol also checks that you are not executing an instruction that would cause your job to fail to execute correctly. For instance, if one of your programs creates a data set that is required as an input data set to a later program, a check is made to see that you do not delete it accidentally before your later program reads it. In this case, Jol would ignore the **SCRATCH** instruction.

Another type of error that Jol detects before allowing your job to begin executing is when you attempt to read a data set that you are not creating in your job and also cannot be found in the **System Catalog** .

Any errors or warnings are highlighted in your Jol program and they may also be printed on a separate page after your program has been listed.

Macros

Jol commands can be combined with themselves and with any other program to form new commands tailored specifically to your installation. With Jol you can also execute commands from within commands, adding greatly to the flexibility and simplicity of procedures. This open-endedness is one of the highlights of Jol.

The Jol Macro Language may be considered as a major extension of the **JCL PROC** edure statement, and provides a means of adding new instructions to the Language quickly and easily, thus allowing the tailoring of Jol to your installation. The Macro language provides you with the ability of using *any* of the instructions listed below, or instructions *you develop* , and combining them into new instructions specifically for your installation, or a specific application.

A Macro is a set of instructions that creates Jol instructions to the user's specification, and once coded is stored in a **MACRO** library. Whenever a non-standard Jol instruction is found, the MACRO library is searched and interpreted, creating the tailored Jol instructions. The compilation then continues recommencing at the first of the created instructions.

Macros can:

- Examine, verify and change User options or parameters.
- Set up default parameters.
- Execute other macros commands.
- Generate tailored Jol instructions.
- Set or alter the values of symbolic variables so that other instructions can test switches or values and take some action based on the results of the tests.
- **ALLOCATE, OPEN** and **WRITE** data in a manner similar to the Jol **SAVESYMS** macro command which saves the current values of symbolic variables in a data set for later use.
- Write tailored PL/I, COBOL or JCL language statements to a data set, or submit them to a background region to be executed.

Note : Macros may be considered as small **reentrant** and **reusable** programs.

Figure 1. Examples of Functions Macros can Perform.

Macros or Commands are easily prepared by coding a **MACRO** prototype statement followed by a series of Jol statements and terminated with an **END** .

The **SORT macro,** a standard Jol **macro,** is called with parameters and allocates work areas for the sort program and performs the sort function for you.

For example, to make a command to both Print and Delete the input data set may be coded as:

```
PRDEL : MACRO;
      PRINT %LIST(1);
      DELETE %LIST (1);
END;
```

Figure 2. Example of a Macro.

Jol Instructions

The following is a partial list of instructions included in **Jol** . Most of the instructions manipulate **DECLARED** variables, data set identifiers and load modules, while others provide information to the terminal User or the Operator. These instructions, and others, are explained in later pages.

Instructions	Use
ALLOCATE	A Data Set
ASSIGN	A Value to a Symbolic Location
BUILDGDG	Build Catalog Entries for GDGs
CATALOG	A Data Set Identifier or Data Set
COPY	A Data Set or DSID to another
DELETE	A Data Set Identifier or Data Set
DISPLAY	A Message on the System Log
DO	A Group of Instructions
EDIT	Symbolic Variables
END	A Group of Instructions
ENQ/DEQ	Use Special Resources
IF	Test the Value of a Return Code or Symbolic Parameter
INCLUDE	Source Text from a Library
INVOKE	A Load Module into the Jol Compiler
KEEP	A Data Set Identifier or Data Set
LIST	Data Sets
LISTCAT	List the Catalog
MERGE	Data Sets
OPENFILE	Open a File for Input or Output
PANEL	Create a Formatted screen for data entry
PRINT	Print Data Sets
PRINTSYM	Print *all* Symbolic Variables and Values
READ	A Record from either a File or the Terminal
RUN	A load Module or Program
SCRATCH	A Data Set Identifier or Data Set
SIGNAL	An Error or Warning
SORT	A Data Set or DSID to another
START AT	Start At a Specific Part of the Program
STOP	The Execution of the Job
STOP AT	Stop At a Specific Part of the Program
STOP WHEN	Stop When Specific Return Codes are Found
SUBMIT	Other Jobs
TEXIST	Test if a Data Set Exists for Scheduling or Restarts
TYPE	A Message on the Operator's Console
UNCATALOG	A Data Set Identifier or Data Set
WRITE	A Record to a File or the Terminal

Figure 3. Partial List of Jol Instructions

DECLARE or **DEFINE** statements are used to describe Data Sets and Programs, and to set up Symbolic Variables.

A **JOB** statement gives details such as the name of your job and its expected elapsed and **CPU** times.

A **MACRO** statement defines macro commands and sets up defaults for keywords used in the macro commands.

For example, to **PRINT** member **PAYROLL** of the **JOL.INCLUDE** data set you could code:

```
PRINT JOL.INCLUDE(PAYROLL);
```

Programmed Instruction Course

There is a Jol teaching aid which can be invoked by typing:-

```
CAIJOL;
```

directly after calling Jol with the * option.

The course is approximately three hours long; you may break and restart at any time. Note there is also a version of the course available for the IBM Personal Computer.

General Format of a Jol Program

In essence, a Jol job is made up of one or more statements, instructions, or commands. Statements may **DECLARE** or **DEFINE** variables, datasets, programs, etc. Instructions and Commands cause some action to be taken with the items **DECLARED** or **DEFINED** .

With Jol, like with all other high level languages, you divide your data set and program declaration part of your program from the logic part. This allows others, who do not know your Jol program, to see the logic very quickly. Jol will allow you to intermix Declarations and Commands. However, this is not a desirable practice as reading the program then becomes more difficult.

In general the following procedure is followed:

1. Code all symbolic variable definitions with their initial values.
2. Code all the dataset and program definitions. These do not have to be in any particular order. For example, even though you specify that a program is going to read or write a data set, the data set definition can appear after the program definition uses it.
3. Code the instructions and commands to manipulate the defined data.
4. These commands may appear in any order, but if you are reading a data set, ensure that any program that creates it **RUN** s first.
5. Symbolic Variable processing can appear anywhere within the program.

Remember that Jol is basically a SERIAL Compiler. Jol will execute commands in the order they are written.

The examples below how a Jol job can be layed out and documented. Both Jol programs perform the same work, but the second example shows how Jol programs can be documented with comments.

The Jol programs in below sort a data set called **INPUT.SORT** in character format, starting at column or character position 69 for 2 characters, and place the sorted file in a data set **OUTPUT.SORT** . If the **SORT** is successful, the sorted data set is **CATALOG**ed and **PRINT**ed.

```
SORTJOB :          JOB ACCT='(1,000,SYS,,,1)'
                           NAME=JOLUSER CLASS=A ;

DCL INPUT          DS DSN=INPUT.SORT ;

DCL OUTPUT         DS DSN=OUTPUT.SORT
                           FB 80,800 5 TRKS
                           SYSDA VOL=WORK01 ;

STEP010:
                SORT INPUT TO OUTPUT FIELDS = (69,2,CH,A) ;
                IF STEP010 = 0
                THEN DO ;
                       CATLG OUTPUT ;
                       PRINT OUTPUT ;
                END ;
```

Figure 4. A condensed Jol Job Example.

```
/*  THIS JOB PERFORMS A SORT ON THE DATA SET
     'INPUT.SORT'   AND PRODUCES A DATA SET
     CALLED 'OUTPUT.SORT'.

     NOTE:          THE ';' SIGN INDICATES THE END OF A STATEMENT
                    IN JOL.   THIS AND THE ABOVE STATEMENTS
                    ARE EXAMPLES OF CODING COMMENTS IN JOL */

/*  DEFINE THE JOBCARD */

     SORTJOB: JOB                     /* SORTJOB  is the   Jobname   */
          ACCT = '(1,000,SYS,,,1)'    /*  Accounting Information */
          NAME = JOLUSER              /*  Programmer Name */
          CLASS = A                   /*  Allocates the Job Class */
          USER = '??????'             /* Provides USERID for RACF */
          PASSWORD = '????????' ;     /*  RACF PASSWORD */

/*  DEFINE DATA SET INFORMATION */

     DECLARE INPUT DATASET            /* INPUT is the Data Set ID   */
          DSN = INPUT.SORT ;          /*  Specifies the Data Set Name  */

     DECLARE OUTPUT DATASET           /*  OUTPUT is the Data Set ID */
          DSN = OUTPUT.SORT           /*  Data Set Name for Output */
          FB 80,800                   /*   Record Format     */
          5 TRACKS                    /*   Space Allocation   */
          UNIT = SYSDA                /*  Unit Type    */
          VOLUME = WORK01 ;           /*  Volume Name */

/*  INSTRUCTION AND COMMAND SECTION */

     STEP10:                          /*   Stepname   */
          SORT INPUT TO OUTPUT        /*    SORT the DSID's           */
               FIELDS = (69,          /*    Start of Field */
               2,                     /*    Length   */
               CH,                    /*  Character */
               A) ;                   /*  Ascending */

          IF STEP10 = 0               /* SORT OK? */
          THEN DO ;
              CATLG OUTPUT ;          /*   Catalog the data set  'OUTPUT.SORT' */
               PRINT OUTPUT ;         /* Now Print it */
          END;
```

Figure 5. A Fully Documented Jol Job Example.

Notes on the Jol Examples

The above examples highlight a number of features in preparing a Jol job:

1. In the above example, **"INPUT"** and **"OUTPUT"** are known as Data Set Identifiers, or **DSID**s. Dataset identifiers or DSIDs can be allocated to all dataset definitions.

   DSID s can be used in instructions, commands, and program definitions, for example in Catalog Instructions, **SORT**s and other Commands. Thus details about a Data Set need only be defined once **-** any reference to the **DSID** automatically picks up the required information.

A **DSID** defines the Data Set Name, Volume, Unit and any other information that may be necessary, such as Record Format. You do not specify on whether the **DSID** is to be used for Input or Output; this is done in the **Program Definition** , or implied in some instructions such as **SORT** .

2. Comments can be coded anywhere in the jobstream where a blank space is allowed.

3. If you wanted to print the output on two part stationary, the **PRINT** instruction could be coded as:

   PRINT OUTPUT ON 2 PART;

   or **PRINT OUTPUT 2 PART** ;

4. The **CATLG** statement at the end of the above job catalogs the output data, otherwise it would be **DELETE**d at the end of the job.

   It is good practice to put all the **CATLG** , **KEEP** , **DELETE** , and **SCRATCH** instructions at the end of all the smaller jobs. If an error occurs, you can then rerun the job in its entirety without having to recreate data sets or adjust the catalog.

5. You will notice that there are no tests for an error condition above. Jol automatically aborts the job if a return code of 16 or greater than 2000 is returned by any program. This occurs, unless you override the Jol default condition codetesting with the **STOP WHEN** *condition(s)* instruction.

Further notes about Jol Programs

1. Definitions, instructions, and commands can be intermixed. However, clarity is greatly improved if the suggested method is followed.

2. Symbolic variable processing can be performed anywhere in the job.

3. All declarations can be placed in a library for use by all your Programmers and Operators.

   Like other high level languages, defined variables such as data sets, programs and symbolic variables do not have to be used in any instructions. This allows them to be placed in a member of a library and **INCLUDEd**, and used much like an **INCLUDE** in **PL/I** or a **COPY** statement in **COBOL** is used to copy variable definitions in programming languages.

   By having the Definitions for all the data sets and programs in a central library system testing for a set of jobs can be made much easier. You can use an **INCLUDE** to copy the program and data set definitions, and then follow the **INCLUDE** with appropriate instructions for the particular part of the system you are testing.

   This facility is explained in greater detail in the *Techniques* section of the *General Information Manual*

Use of Dsnames and DSIDs

Generally any Data Sets are declared as Data Set Identifiers (DSIDs). For example:

DCL INPUT DS DSN=TEST.DATA;

This is a **DSID** called **INPUT**. This **DSID** may be used in many ways, but a listing of the contents of the data set referred to by the DSID **INPUT** could be performed with:

```
LIST INPUT;
```

The **LIST** Command could have been written

```
LIST 'TEST.DATA';
```

which would have the same effect as the two previous instructions.

This facility is particularly useful for "one off" jobs; for example, **LIST** ing or **PRINT** ing a data set.

However, in a production job or a system test, it is <u>highly preferable</u> to use the DSID form of using Data Sets because:

1. The Dsname, Record Format and other details need only be specified once no matter how often it is referenced.

2. All the details about the data set may be seen at a glance.

3. Over-rides may be performed once only to effect changes whenever the DSID is used.

4. Volume, Space, Unit and DCB information may be declared in a Data Set Declare.

N. B. Some Commands or Macros assume that the **DSNAME** provided is of the type 'name1.name2', that is they contain at least one
Index Level.

Overview in Using Jol

In summary, to use Jol, you first specify the data sets and programs which may be required for the job. Simple instructions such as**RUN** and **SORT** can then be used to set the job in motion. The job may be: sorting of data sets, testing condition codes, running programs, etc. **-** in fact most tasks that can be done with Clists, ISPF and JCL.

The Jol compiler may be executed interactively or in a background region. Executing Jol interactively allows the use of full screen **PANEL**s to input data, and other interactive facilities. Data entered through a terminal can be used to create different job streams, as input to your programs or written to a data set for use by another program. This process is described as Preprocessing the Jol text, and is described in detail in a later section of the manual.

You can run Jol interactively by entering Jol commands directly from TSO, or you can have Jol read your commands from a Jol source file data set or member in a Partitioned Data Set.

After you have written your Jol program using a standard editor, you can:

- Invoke Jol under TSO/SPF to compile and run the job immediately under TSO.

- Invoke Jol under TSO/SPF to compile and submit your work to Background for execution.

or

- Submit a *batch* job to the Jol Compiler which will perform syntax and logic checks, and then submit the jobto run in a background region.

Operating Jol Interactively

Jol is most often used interactively. A Jol dataset is generally required for each user with a name **"sysuid.JOL"** .

To execute Jol simply enter:-

JOL

You will be prompted for a Jol member containing the program you want compiled.

Alternately you may type:-

JOL *

and enter Jol statements directly.

Once your job is compiled successfully you will be asked if the job should be submitted to the operating system for execution.

Note: The Jol Command Processor has many other options:- check with the specific Jol User Guide for your operating system.

Operating Jol with JCL

To execute the Jol Compiler in the background, you must code one or more JCL statements before you Jol program.

For example:-

//jobname JOB (accounting information)
// EXEC JOLCGO [,optional parameters]
Jol statements

If you are using an interactive system, **SUBMIT** the JCL and Jol for execution. If you are using a card based system, then submit the card deck in the usual way.

There are three supplied JCL procedures:-

JOLC	compiles but **does not submit** the job for execution. Useful for debugging.
JOLCGO	compiles and submits the job for execution.
JOL	allows the operator to START a Jol job through the operating system console.

Batched Jol Compilations

Batched compilations may be preformed by enclosing the Jol program with *** JOL;** statements. For example:

```
* JOL;
        first Jol program
* JOL;
        second Jol program
```

Note

1. The *** must** be coded in column 1.

2. Other options may be coded on an *** JOL** statement such as initial values of Symbolic Variables, and the Operating System on which the job is to execute on. See the *Compiler Options* section.

Language Structure

A Jol Program is made up of one or more statements or instructions. Statements are used to **DECLARE** or **DEFINE** variables, data sets, programs and so on. Instructions and Commands usually cause some action to be taken with the items declared or defined. For example:

```
DECLARE COPY PROGRAM
        SYSPRINT WRITES PRINTER
        SYSUT1 READS INPUT
        SYSUT2 WRITES OUTPUT;
```

is an example of a **Program Declare**.

```
RUN COPY 'TOTAL';
```

is an example of an Instruction.

As long as each statement is terminated by a semicolon, the format is completely free. Each statement may begin in any column and end in any column, subject to installation defined source-code limits. Usually, you can code in columns 1 to 72. Column 73 may be reserved for special characters that instruct Jol how to format your printed output.

However, all Jol programs must follow these general rules:-

- All statements must be terminated with a semi-colon.
- More than one statement may be coded per line, but in practice it is easier to correct errors and alter the program if only one statement is used per line.
- A statement is made up of a number of words or "tokens". A token is part of a statement, and a name, number or special character (for example a period) is considered to be a token. To Jol:

  5 CYLS is equivalent to **5CYLS**

  SYS1.LINKLIB is equivalent to **SYS1 . LINKLIB**

- Statements may be continued over as many cards as is necessary. Where one blank is permitted any number may be used. Comments may be used in addition to, or in place of, blanks to annotate the program.

  This means that parts of a statement may be coded on different card images. (That is, with any number of blanks separating them.) Thus:

  ```
  DCL INPUT DS JOL.INCLUDE;
  ```

  is equivalent to

  ```
  DCL INPUT DS
          JOL.INCLUDE;
  ```

- The percent sign (**%**) has special meaning in Jol:- It indicates the start of a **Symbolic Variable** name.

  To Display a message with a **%** sign two **%** signs must be coded:

  ```
  DISPLAY 'JOB 50%% COMPLETED';
  ```

- Symbolic Variable or Parameter processing may appear anywhere in the program, thus allowing alterations to the source code as it is being compiled.
- In general, commas, equal signs and brackets may be coded or not according to

preference. Often keywords are unnecessary as the meaning can be determined from the context of the word in the statement.

- Strictly speaking there are no reserved words in Jol, but often the Command or Macro Instructions use some keywords for special purposes.

  Therefore, it it advised not to **DECLARE** names of **DSID**, **DSN**, **SPACE**, **UNIT** or **VOL**.

- The first statement of a Jol program *may be* a Jol **separator card**. This is a card with an asterisk in column one and the word **JOL** followed by a semicolon coded after any options. The format is:

  *** JOL** *options***;**

  This statement indicates the start of a new Jol program and the end of the previous program. You can also specify special processing options with the *** JOL** statement such as compiler listing options or initial settings of symbolic variables. These options are discussed in the *Compiler Options* section.

- Between one *** JOL** and the next *** JOL** or the end of the source file input is the Jol program consisting of Jol statements, Jol macro commands, installation macros and any card image input for your job.

Statement Types

There are three basic statement types: Keyword, Assignment and Null Statements.

- A **keyword** statement consists of a keyword followed by a statement body. For example:

  ```
  TYPE 'THIS IS AN EXAMPLE OF THE TYPE
                      INSTRUCTION';
  ```

  This is a keyword statement and **TYPE** is the keyword.

- An **assignment** statement contains the assignment symbol (=) to indicate an assignment instruction. For example:

  ```
  %SYMBOL='A';
  ```

  is an example of assignment statement.

- A **null** statement is coded as a semi-colon and contains no other information.

The **IF** statement is a compound statement. That is, the IF statement is coded and the instruction to be executed follows without an intervening semi-colon. For example:

```
IF SORT = 16
THEN STOP
          'TERMINAL ERROR OCCURRED IN SORT';
```

Syntax Notation

The format of the statements described in this manual provide a brief, uniform and precise explanation of the general patterns of the Jol language instructions. The notation does not describe the meanings of the language elements, but it indicates the order in which the elements may (or must) appear, the punctuation, and the options that are allowed.

For example:

```
COPY   { dsid | dsname }
       TO     { dsid | dsname }
```

states that the **COPY** consists of:

- The **COPY** word itself.
- A **DSID** or **Data Set Name** that is to be used as input. The   indicate that you must write a **DSID** or a **DSNAME**.
- The reserved word **TO**, which is then followed by the name of the **DSID** or data set name to which the data indicated in the first variable will be copied to.

The following rules apply:

1 Uppercase letters and words are coded exactly as they appear in the format description, as are any special characters. Thus:

 PRINT *data-set-name*;

 states that the word **PRINT** must be coded exactly as shown. Special characters:

 = . / ( ) + - ' * , < > | ⌉ ^ : ; %

 and the composite special characters:

 ⌉> ^> ⌉< ^< >= ⌉= <= /* */ ||

 must be coded as shown.

 Note that the symbol ^ can be used instead of the symbol ⌉ on terminals that do not support the **EBCID** ⌉ (NOT) symbol.

2 *Italic* lowercase, letters, words**,** and symbols in the format descriptions represent variables for which specific information is substituted when the parameter is coded. For example:

 NAME = *dsname*

 indicates that the data set name must be substituted for the word "*dsname*".

3 **Braces**   are a special notation and are never coded in any statements. They indicate that a choice must be made from one of the items enclosed in the braces. For example:

```
{ DEFINE   }
{ DECLARE  }          data-set-identifier
{ DCL      }
```

 indicates that you must use either **DEFINE, DECLARE** or **DCL** to indicate that you wish to define an item in **JOL.**

Sometimes, instead of vertically stacking the items, the vertical stroke is used to indicate that a choice must be made. For example:

FB | VB

indicates that you must choose **"VB"** or **"FB".**

4 **Square brackets [ ]** are a special notation and are never coded. They indicate that an enclosed item or items are optional and you can code one or none of the items. For example:

PARALLEL [MOUNT]

indicates that the **MOUNT** may, or may not, be coded, and if it is coded, it must immediately follow the word **PARALLEL.**

5 **An ellipsis . . .** (three consecutive periods) is a special notation and is used to indicate that the preceding item can be coded more than once in succession._For example:

VOL [=] *volume* ***[, . . .]***

indicates that more than one volume may be coded. If another volume name follows the first, a comma must precede the second volume name.

6 **Underlining** is used to indicate the default that will be applied if none of the options are chosen. For example:

[<u>FB</u> | VB]

indicates that you may code **"VB"** or **"FB",** but if you do not code either option, **"FB"** is chosen as the default.

7 A blank is used to separate identifiers and constants that are not separated by special characters or comments. Where one blank is required or allowed any number may be used. With the exception of character string constants, no identifier, constant or composite symbol may contain blanks.

8 Comments are permitted wherever a blank is allowed and are treated as blanks if no blank is found. Comments do not effect the program in any way but are used for documentation purposes only.

A comment is commenced by the composite symbol, **/***, and terminated by the composite symbol, ***/**. Any character may be coded in the comment except the end of comment composite character which would terminate the comment. Examples of the comment are:

a) **/*THIS IS A COMMENT*/**

b) **/* THIS COMMENT EXTENDS
OVER TWO LINES */**

9 A special type of identifier or data type is the symbolic variable. A symbolic variable is a name which commences with the **percent character**, **%**.

It contains a character string of up to 253 characters which may be used to replace various parts of a JOL program during the preprocessor phase of the compile. When a statement is found with any symbolic variables as part of the text, they are replacedby the current value of the symbolic variables.

Symbolic Variables may be used to replace all or part of a Jol statement, or a card image file. For example:

```
%A='PRINT INPUT';
...
%A;
```

will modify the Jol program, and insert the statement **PRINT INPUT** at the **%A;**.

Symbolic Variables may be saved in a data set with the **SAVESYMS** command, and retrieved in another job, or the same job executed at a later time. This facility allows information entered by a user to be used as initial values at some later time.

See "*Compile Time Facilities"* on the following pages for further details about Symbolic Variables.

Compile Time Facilities

Jol offers a relatively unique facility **-** that of altering Jol code in a *programmed* manner.

Compile Time Facilities are performed by the Preprocessor. The Preprocessor allows program Source Text to be altered through the use of Symbolic Variables, and it invokes Macros and expands the Macro text. The expanded Source Text is then read by the main Jol Compiler and instructions are produced to execute the job.

For example, if the day of the week is Friday, you can include code to run only on Fridays. You can also read data sets such as machine schedules to see which of your programs should run.

Most of the facilities mentioned here make use of *Symbolic Variables.*

Symbolic Variables

A symbolic variable is a special variable, and you identify it by placing a **%** in front of its name. It contains a character string of up to 253 characters which may be used to replace various parts of a JOL program during the preprocessor phase of the compile. When a statement is found with any symbolic variables as part of the text, they are replaced by the current value of the symbolic variables.

Symbolic variables contain *text* and this text can be used in replacement statements, as well as other statements described later.

For example,

```
%X='TEST MESSAGE';
...
DISPLAY '%X';
```

is equivalent to writing

```
DISPLAY 'TEST MESSAGE';
```

Explanation: Jol found the symbolic variable **%X** and replaced the characters **%X** with the text contained in the *variable %X.*

Symbolic Variables can be:

- tested,
- changed,
- added, subtracted, multiplied, divided and concatenated to other variables or literals.

Compilation Phases

It is important to realize that a Jol compilation is really done in three separate steps:

1. The **Preprocessor** or **Macro** phase. The output from this stage is *text*, or Jol statements. These Jol statements then are read by:-
2. The **Compile** phase. The compile phase does syntax checking and creates tables which are read by:-
3. The **Generate** phase. The generate phase either creates special instructions so that Jol can execute your program under TSO, or it creates JCL so that the job can run in background.

It is during the *Preprocessor or Macro* phase that you can alter selected parts of your program.

Facilities

The Jol Preprocessor allows Program Source Text to be altered before it is compiled and executed. You can:-

- Define Symbolic Variables and initialize them.
- Test the current contents of Symbolic Variables.
- Change the values in the Symbolic Variables.
- Perform Arithmetic and String operations on Symbolic Variables.
- Replace text with Symbolic Variables.
- Replace text *in card image files* with Symbolic Variables.
- Save and Restore Symbolic Variables from disk data sets.
- Modify, alter and or insert program statements.
- Indicate which sections of the Source Program are to be compiled.
- Allocate, read and write data sets. The data read can then modify the program as above.
- Unconditionally or conditionally **INCLUDE** Jol Statements or card image files from libraries.
- Use Macros, with or without Parameters, to generate statements.
- Conditional Compilations based on the values of Symbolic Variables.
- User written Assembler, Cobol or PL/I code to be executed and generate Jol statements.

You can freely intermix preprocessor and macro statements with the rest of your instructions. Preprocessor and macro command statements are executed when they are encountered in the source text.

Preprocessor Operation

The preprocessor does as much processing as it can before it produces code to run at execution time **-** it is usually faster to do as much processing as possible in the *Preprocessor phase*. Jol does this for you as automatically as possible.

With the above in mind, Jol will attempt to reduce the number of *execution* statements to an absolute minimum.

Therefore, when the preprocessor reads a statement, it does the following:**-**

- If the statement follows a ***false*** **IF statement**, the statement (or group of statements) is ignored.
- *Execution statements* such as a **RUN, CATALOG** or similiar statement, are copied out to a file that is later read by the main compiler pass of Jol. Such a statement is then executed at *run time.*

  Note: If the statement about to passed to execution a Macro instruction, the **Macro** processor will be given control to interpret the **Macro** statements.

- *Preprocessor statements* such as:**-**

  - A symbolic variable declaration or assignment statement.
  - An **IF, DO, END, ELSE, SIGNAL ERROR** or a **STOP** instruction.
  - A **Macro** instruction.
  - A **READ, WRITE or PANEL** instruction.

  are *interpreted immediately*.

Preprocessor Replacement

In all cases, before either executing a statement as a preprocessor statement or passing it to the compiler phase, symbolic variable replacement takes place. Text replacement takes place whenever a **%** symbol is found. The**%** symbol and the associated identifier are replaced with the current value of the named symbolic variable.

In Jol, symbolic variable replacement stops when the next special character or blank is found in the text, and if two symbolic variable names are found without an intervening special character or blank, the replacement results in one composite symbolic being formed. If **%NO** contained a **1** and **%VAR** contained **'A'** then **%VAR%NO** would be changed to **A1.**

Examples

Example 1. If a program contained the following statements:

```
1   %UNIT='TAPE';
2   DCL DATASET1 DATA SET UNIT %UNIT;
3   IF (%UNIT='DISK' | %UNIT='DISC')
      & %SPACE=' '
4   THEN %SPACE ='SPACE=10,2 CYLS';
5   DCL DATASET2 DS UNIT %UNIT %SPACE;
```

then the output from the preprocessor would be:-

```
2   DCL DATASET1 DATA SET UNIT TAPE;
5   DCL DATASET2 DS UNIT TAPE;
```

However, if **statement 1** had been **%UNIT='DISK'** then the output example would have been:-

```
2   DCL DATASET1 DATA SET UNIT DISK;
5   DCL DATASET2 DS UNIT DISK SPACE=10,2 CYLS;
```

Example 2. If a program contained the following statements:

```
1   %START = 'STEP2';
2   IF %START='STEP2' THEN
3   DO;
4               DCL X DATA SET NAME TEST.INPUT (0);
5               DISPLAY RESTARTING AT STEP2';
6   END;
7   DCL X DATA SET TEST.INPUT(+1) VB 750,800 TAPE;
```

then the output from the preprocessor would be:-

```
4   DCL X DATA SET NAME TEST.INPUT(0);
5   DISPLAY 'RESTARTING AT STEP2';
7   DCL X DATA SET TEST.INPUT (+1) VB 750,800 TAPE;
```

Statement 4 would set the **NAME** of the data set to be **TEST.INPUT(0)** and **statement**

7 would supply the **UNIT** and Record Format information for the data set. This is a Jol override, and the name **TEST.INPUT(+1)** would be ignored, and when the data set identifier **X** was referenced, generation **(0)**, that is the latest generation, would be accessed.

Symbolic Parameters and Variables

Symbolic Variables are often used to change program execute parameters. For example, suppose that a program required a parameter with the current day. By using the Jol symbolic variable **%DAY**, the program will automatically have the correct day passed to it through the program parameter field. For example:

```
RUN PROGRAM1 'DAY=%DAY';
```

will execute **PROGRAM1** with a parameter of **'DAY=MONDAY'** or **'DAY=TUESDAY'** and so on.

Other user defined symbolic variables may be set up, for example **%TEST='YES'**. Whenever the symbol **%TEST** is found, it will be replaced by **'YES'**, or whatever the symbolic variable **%TEST** currently contains. This means that **%TEST** need only be changed in one place in the Jol program, and Jol will effectively change it whenever it is used.

Testing Symbolic Variables and Arithmetic

Jol allows testing of symbolic variables, and arithmetic to be performed on them. For example:

```
IF %RANGE<10 | %RANGE>100
THEN STOP 'RANGE=%RANGE:- INVALID';
```

- The **IF** instruction tests that **%RANGE** contains a value between **10** and **100**.
- If the variable **%RANGE** is not within **10** and **100**, the **STOP** is executed and will terminate the Compile. That means that the job will not execute.
- Otherwise, the next instruction is executed.

Defining a Symbolic Variable

The **Symbolic Variable Definition** follows this form:-

```
        { DECLARE | DCL | DEFINE }
                %name[,%name ...]   options  ;

    where options are:

        INIT [ ( ]    { constant | ‘character string' }   [ ) ]

        EXTERNAL | EXT
```

Figure 6. Defining Symbolic Variables

Symbolic Variable Names are recognized in the text by the % preceding the name in the source text.

Notes

1. If a value has been previously **ASSIGNED** to a Symbolic Variable, the declaration is ignored (an assignment being an implicit declaration), thus allowing the declare to set default values in the symbolic variable which may be overridden.
2. The maximum length of the value is **253** characters.
3. When **EXTERNAL** or **EXT** is used inside a Macro Definition, the symbolic name is

available when the Macro terminates. In main line code, **EXTERNAL** has no effect.

4. If a symbolic variable has been previously defined, the Declaration statement is ignored. This allows default values to be set.

Examples

1. **DCL %A INIT '';**

   Defines **%A** and initializes it to null.

2. **DCL %B, %C, %D INIT 'EXT';**

   Defines **%B,%C** and **%D** and initializes them to **'EXT'**.

3. **DCL %GENNO INIT '(0)';**

   Defines **%GENNO** and initializes it to **'(0)'**.

Automatically Initialized Symbolic Variables

Jol initializes certain symbolic variables at startup time. These variables are described below:

Variable	Explanation
%SYSDATE	The current date in Julian format e.g. **86290**
%DAY	**MONDAY**, **TUESDAY**, etc.
%MONTH	**JANUARY**, **FEBRUARY**, etc.
%MONTHNO	**01, 02** Through **12**
%DAYNO	**01** through **31**
%YEAR	**1987, 1988**, etc.
%HOURS	**0** through **23**
%MINS	**0** through **59**
%SECS	**0** through **59**
%SYSUID	System user identification
%SYSPREF	Dataset Prefix or Current Directory
%SYSPFK	Program function key number from **PANEL**s
%SYSTEM	**MVS, DOS, VM**, **PC, UNIX** etc.
%SPOOL	**HASP, ASP**, **JES1**, **JES2**, **JES3**, or Blank
%TSOCLASS	Contains the **SYSOUT** class used by TSO to retrieve output from a background job

Figure 7. Startup Time Symbolic Variables.

These variables are initialized once and can then be used with the Jol logic statements to control events and functions.

Using Jol's Calendar

The symbolic variables above can be used in **IF** statements, or in simple replacement statements. The following two examples use the preinitialized examples to examine the date.

1 **IF %DAY='FRIDAY'**

```
IF %DAY='FRIDAY'
THEN DO ;
    RUN WEEKLY ; /*Program Weekly*/
    PRINT FILEA ;
END ;
```

This code could be part of a daily procedure. Only on a Friday would the program '**WEEKLY**' be executed and '**FILEA**' be printed.

2

```
IF %DAYNO > 25
& %MONTH='JUNE'
& %DAY='FRIDAY'
THEN DO ;
    SUBMIT YEARLY ;    /*SUBMIT JOB YEARLY*/
    TYPE 'JOB YEARLY SUBMITTED TO RUN' ;
END ;
```

In this example, if the day is Friday and the day of the month is greater than 25 and the month is June, then the job 'Yearly' is submitted and a message is placed on the operator console.

Special Symbolic Variables for Macros

In addition, the following Symbolic Variables are automatically initialized and changed during the compile. They are of particular interest to someone writing Jol Macros.

These, and their contents, are:-

Variable_	Explanation
%SYSLABEL	The label on the statement invoking the macro.
%SYSMACNM	The name of the macro called ... useful when you have one macro doing several related functions and each is stored as an **ALIAS**. You can have this to determine which "entry point" the macro was called at.
%SYSNLIST	The number of **%LIST** (non-keyword) items found on the invoking statement.
%SYSLEVEL	The current level of the macro. If a macro invokes another macro it will be set at 2; in the mainline code it is set at 0; and in the first macro invocation it will be set at 1.
%SYSSTMT	This variable is set at 0001 - 9999, and is altered every time a new statement is read.

Figure 8. Special Macro Symbolic Variables

When an instruction is found that is not one of Jol's own, control is given to a Macro. It will examine any parameters and produce the necessary Jol code to perform the function required.

For example:

```
COPY        'SYS1.MACLIB (CALL)'
    TO      'TEST.MACLIB (CALL)' ;
```

will transfer control to the **COPY** Macro, which will generate a program to perform the **COPY**.

Requesting Space for Data Sets

Every data set on **Direct Access Volumes** must be allocated space on that volume. Conceptually, you may consider a disk volume as a finite amount of Space; the **SPACE** Parameter allows you to specify how much space is required for your data set. Too much space is wasteful, while too little will result in abnormal terminations when you are writing a data set and have filled the requested space.

VSAM Data Spaces must be allocated with the **DEFINE** command of the Access Methods Service Program.

You must specify how much space you require for your data set. For example, if you require space for 1000 Records, code:

1000 RECORDS

If you require 10 tracks, or 10 cylinders of space, code:

10 TRACKS *or* **10 CYLS**

If the space you request is not available the job is terminated.

There are two ways to request space:

- Tell the system how much space you want and let the system find the required free space on the volume and assign it to you.
- Tell the system the specific tracks you want allocated to your data set **-** the **ABSTR** parameter is used for this.

Letting the system find the free space you want is the most commonly used method of allocating space. Using the **ABSTR** method of allocating space is discussed in the **SPACE** parameter in the *Jol Reference Manual*.

Methods of Specifying Free Space

There are four methods you can use to specify free (that is, non-specifically placed) space:

1. Code the number of records expected to be output.
2. Code the number of blocks expected to be output.
3. Code the number of tracks required.
4. Code the number of cylinders required.

You specify a primary or initial amount of space to be allocated in the data set. You can also specify a secondary quantity to be used if the primary allocation is filled.

You can also specify the following options:

- Space for a directory or index **- DIR**.
- Release of free space in the data set **- RLSE**.
- Contiguous or unfragmented space **- CONTIG**.
- Whole cylinders **-** the data set is to begin on a cylinder boundary **- ROUND**.

Specifying Data Set Space Requirements

Specifying space requirements for a Data Set on a Direct Access Volume may be done in any of the following methods:

1. *n1* [-*n2*] **{ RECORDS | RECS } [OF** *modal-record-length]*

or

```
2.  [SPACE] [=]   primary allocation        [ CYLS      CYL         ]
              [   , secondary space    ]    [ TRACKS    TRKS  TRK   ]
              [  , {  directory-blocks } ]  [ BLOCKS    BLOCK       ]
              [   {  index-blocks       }   ]           [ BLKS      BLK ]
```

or

3. **ABSTR** [=] primary-allocation,track-number
[, ***directory-blocks* | *index-blocks*_]**

Other options which may be coded anywhere in the Declare Statement are:

<u>RLSE</u> | NORELEASE | NORLSE
MXIG
CONTIG
ROUND
ALX
DIR *directory-blocks*

where:

address	specifies the first track number to be allocated.
modal-record length	specifies the most common record size in the data set (for Variable or Undefined Record Formats only).
secondary quantity	specifies how many more tracks or cylinders are to be allocated if additional space is required, or how many more blocks of data may be included if additional space is required. This allocation can be done up to 16 times for each volume, less the numberof extents for primary quantity and user label space.
ABSTR	specifies that the data set is to be placed at a specific location on the volume.
ALX (all extents)	specifies that up to five different contiguous areas of space are to be allocated to the data set and each area must be equal to or greater than the primary space requested._Do not use **ALX** for indexed sequential data sets.
CONTIG (contiguous)	specifies that space allocated to the data set must be contiguous. If the request cannot be satisfied the job is terminated._If secondary space is allocated to the data set it may not be contiguous. **CONTIG must** be used for indexed sequential data sets.
CYL or **CYLS**	specifies that space is to be allocated by cylinder(s).

DIR	is the number of directory blocks to be allocated.
K (record number space request)	specifies thousands of records.
MXIG (maximum contiguous)	specifies that the space allocated to the data set must be the largest area of contiguous space on the volume and the space must be equal to or greater than the space requested. This subparameter applies only to primary space allocation. Do not use **MXIG** for indexed sequential data sets.
NORELEASE | NORLSE	cancels **RLSE:** see below
RLSE	specifies that space allocated to the data set that is not used when the data set is closed, is to be released. **RLSE** is negated if an Abend occurs or **CLOSE** with the TYPE=T option is used.
ROUND	specifies that space is requested by specifying the average block length of the data and the space allocated to the data set must be equal to an integral number of cylinders, and ensures that the space begins on the first space of a cylinder and ends on the last track of a cylinder.
TRACKS, TRKS or TRK	specifies that space is being allocated by track(s).

Notes

1. Space requests are made in Data Set or Printer Definitions only.
2. When using method (1.) above to specify space:
   a) The *modal recordsize* need only be coded for Variable Blocked files. If not coded, Jol will use the average recordsize.
   b) **K**, if coded, specifies thousands of records.
   c) If no blocksize is specified, **JOL** will use the default blocksize to calculate the storage requirements.
3. When using method (2.) above, the **SPACE** keyword need not be coded if **TRACKS**, **CYLS** or **BLKS** is used.
4. Directory Space can be allocated separately:

   DIR *number-of-directory-blocks-required.*
5. If no Space Request is made for an output Data Set, the installation defined default will be used.
6. Unused Space will be automatically released.

Examples of Space Requests

1. **10-20 RECORDS**
2. **10K RECORDS OF 250**
3. **SPACE= 5,2 CYLS DIR 10**
4. **5,2 CYLS**
5. **1 TRK**

Disposition Instructions

Purpose

Each of the following instructions is described seperately later in this publication. This is an overview of the actions taken on data sets, and depends on whether the data sets existed before the job commenced, or were created by the executing job.

```
    { CATALOGUE |  CATALOG | CATLG  }
or
    DELETE
or
    KEEP
or
    SCRATCH
or
    { UNCATALOGUE |  UNCATALOG | UNCATLG  }

        [ dsid           ]
        [ dsid-list      ]        [ (generation-number) ]    . . .
        [ 'dsname'       ]
        [ 'dsname-list'  ]

     [ON  [VOL] volume [UNIT] unit
                [ALWAYS]   ;
```

Function: To alter the status of Data Sets, and to add, alter or remove the names of Data Sets from the System Catalog.

Overview

- **NEW data sets**: When a new data set is created by a program that **WRITES** a Data Set, it is automatically deleted by the Operating System unless it is **KEPT** or **CATALOGED**. The **KEEP** or **CATALOG** instruction ensures that the Data Set will be available for other jobs when the currently executing job terminates.

- **OLD data sets**: When a job reads a data set that was created by a prior job, they are normally kept when the job finishes. To Delete such a Data Set, the **DELETE** or **SCRATCH** instruction is used.

Specifics

- The **CATALOG** instruction **KEEPs** a Data Set, and also enters the Data Set Name, its Volume, Unit and Position (if the Data Set is on a Tape) into the System Catalog; this means that when the data set is required to be read in another job, the User need not specify Unit or Volume information as this will be obtained from the System Catalog.
- The **KEEP** instruction ensures that new Data Sets will not be deleted at the end of the job.
- The **SCRATCH** instruction scratches the space used by a Data Set, and allows the space thus freed to be used for other data sets.
- The **DELETE** instruction **SCRATCHES** the data set and also removes the data set from the System Catalog.

- The **UNCATALOG** instruction removes the data set name from the System Catalog,but the data set is not Scratched.

Notes

1. All **OLD** data sets are automatically kept unless a **DELETE** or **SCRATCH** instruction is executed, but all **NEW** data sets are scratched by the Operating System unless they are kept or cataloged.
2. Jol will obtain Volume, Unit and Position information from the Data Set Declaration unless a **SCRATCH** volume is associated with a new data set, in which case the information will be found at execution time from the Operating System.
3. The 'dsname' form of the instruction is not recommended for long production jobs as the **DSID** format allows the Data Set Name to be overridden; thus a data set name may be altered in one place **-** the **DECLARE -** and all references to the **DSID** will have the new Data Set Name.
4. If Volume and Unit information is provided in the Data Set Declare or the **ON** option is used for Cataloging a Data Set, Jol does not check that the Data Set is on the specified volume; the data set name and Specified Volume and Unit will be entered in the System Catalog.
5. The **ON** or **FROM** options, if specified, must be coded after the **DSID** or **DSNAME** list.
6. **ALWAYS**, when used with a **CATALOG** instruction, will **RECATALOG** the data set on the specified or implied volume even if it is already cataloged.
7. **ALWAYS**, when used with a **SCRATCH** or **DELETE** instruction, will scratch or delete a Data Set even if the Expiration Date has not expired.

Examples

1. **DELETE DSID1;**
2. **DELETE 'TEST.DATA.SET1'**, **'TEST.DATA.SET2'**

```
DELETE 'TEST.DATA.SET1', 'TEST.DATA.SET2'
            FROM VOL1 2314;
  CATLG 'TEST.DATA.SET1' ON 660011 TAPE;
```

Jol Compiler Options

Jol Compiler Options available may be used to:

- Initialize Symbolic Variables.
- Alter default Compiler Options.

Compiler Options

When Jol was installed on your system, certain default values for the options will have been specified by your system programmer.

The Jol compiler offers a number of optional facilities that may be selected by including the appropriate keywords when using the command processor, in the **PARM** parameter of the **EXEC** statement that invokes it, on the *** JOL** card which separates Jol procedures when using the **Batched Compilation Facilities,** or on the **JOLOPT** instruction.

Initializing Symbolic Variables

Facilities also exist that allow you to initialize Symbolic Variables through the **PARM** field, or the *** JOL** card, thus allowing Jol to be **START**ed by the Computer Operator and have certain Symbolic Variables initialized as though they had been input as Assignment statements**,** or Declare statements.

Compiler options and initial values of symbolic variables may be specified through:

- The Jol Command Processor.
- The Compiler Parameter field.
- On the *** JOL** card.
- By using the **JOLOPT** instruction.

Syntax

> **JOL** *data-set-name options* [**SYMS**(*symbolic-name=value ...*)]
>
> **CALL JOL** *options* [**SYMS**(*symbolic-name=value ...*)]
>
> *** JOL** *options* [/ *symbolic-name=value* [,...]];
>
> **JOLOPT** [*options*] [/ *symbolic-name=value* [,...]];
>
> **// EXEC JOL,PARM**='*options/symbolic-name=value* [,..]'

Example of the TSO Command Processor

To call Jol under TSO, you code the name **JOL** followed by the name of the input data or an *, followed by any other options you require. For example:

```
JOL (PAYROLL) NOCAT SYMS('DOLLVAR=1.25,RESTART=STEP2')
```

Jol is invoked in the foreground, and instructed to read member **PAYROLL** from the users **"*sysuid*.JOL"** data set. In addition **NOCAT** is specified as an option (this tells Jol not to search the catalog for data sets) and the symbolic variables q**DOLLVAR** and **RESTART** are initialized.

Example of Executing Jol in Background

Suppose you have a job called **SALES,** and it required a symbolic called **%INPUT** to be set to **'SALES.INPUT.DAY16.MONTH 11'** before execution. You could execute the job by:

```
      START JOL,JOB=SALES,
            SYMS='INPUT=''SALES.INPUT.DAY16.MONTH11'''

or    // EXEC JOL,JOB=SALES,
      // SYMS='INPUT=''SALES.INPUT.DAY16.MONTH11'''

or    //EXEC PGM=JOL,
      // PARM='INPUT=''SALES.INPUT.DAY16.MONTH11'''

or    // EXEC JOL
      %INPUT='SALES.INPUT.DAY16.MONTH11';
            INCLUDE SALES;
```

Any of the above will set **%INPUT** to **'SALES.INPUT.DAY16.MONTH11'** and compile the **SALES** job.

The * JOL Separator Card

The *** JOL** card is the Jol **separator card**. The ***** must commence in **column 1**.

The separator card is used to separate Jol programs from each other. Rather than execute the Jol compiler separately for each Jol program, you can group together as many Jol programs as you wish, then have Jol compile the entire string of programs or jobs at the one time. So that Jol can determine the start and end point of each job, each must be preceeded by an *** JOL** card. The *** JOL** card may also specify compiler options and/or symbolic variables that are to be initialized before the first statement of the Jol program is read. For example:

```
* JOL MVS JES2/A=10;
```

specifies that Jol is to produce **JCL** for an **MVS** system with a **JES2** spooling package. Additionally, **%A** is to be set to **10**. **%A** may then be used in the Jol program.

The JOLOPT Instruction

The **JOLOPT** instruction allows you to alter Compiler Options, Print Options etc from within your Jol program. For example:

```
JOLOPT PEXPAND;
SORT A TO B FIELDS=(10,10,CH,A);
JOLOPT NPEXPAND;
```

coded in the program specifies that the expanded Jol instructions from the **SORT** command are to be listed. Once they are listed, the **JOLOPT NPEXPAND** instruction tells Jol not to print any more expanded macro instructions.

The Jol Compiler Parameter Field

When executing Jol with **JCL** or **TSO**, you can also specify compiler options and the initial settings of symbolic variables. For example:

```
      // EXEC JOL,PARM='MVS NOPRINT'

or       CALL JOL 'MVS NOPRINT'
```

specifies that the job(s) is to execute on an **MVS** system. The **NOPRINT** option states that *no* Jol statements are to be printed unless an error occurs.

The table following shows the options that may be used, and the default options.

COMPILER OPTIONS	**ABBREVIATED OPTIONS**	**STANDARD OPTIONS**
MVS,F4,X8,PC, DOS,UNIX	**-**	**MVS**
DYNAM, TSO, JCL	**-**	**JCL**
HASP,ASP,JES1, JES2,JES3	**-**	**JES2**
CARD1 **CARD2 = JCL** *Card* **CARD3**	**-**	**-**
DUMPCARD		
FLAG0-FLAG5 **NOFLAG0-NOFLAG5**	**-** **-**	**FLAG0**
LET	**-**	**NOLET**
LINECNT=*n*	**LC=***n*	**LC=55**
NOCAT **NOCATLOG**	**NCAT** **NCATALOG**	**CAT**
PINCL,NPINCL	**PI,NPI,NOPI, NOPINCL**	**PI**
PMACRO,NPMACRO	**PMAC,PM,NPM, NOPM, NOPMACRO**	**NPM**
PEXPAND,NPEXPAND	**PEXP,PE,NPE, NOPE, NOPEXP**	**NPE**
PJCL,NPJCL	**PJ,NPJ,NOPJCL**	**NPJ**
POPTION,NPOPTION	**POPI,PO,NPO, NOPOPT**	**PO**
PRINT,NPRINT	**P,NP,NOP,NO**	**PRINT**
PCOMM,NPCOMM	**PC,NPC,NOPC**	**PC**
SM= (*start-column, stop column, format-column*)	**-**	**SM=(1, 72,73)**
SRDR,NOSRDR	**NSRDR**	
TEST,NOTEST	**NTEST**	**-**

Figure 10. List of Jol Compiler Options.

These options are summarized on the following page.

Summary of Compiler Options

- **DYNAM, TSO, BATCH**

  BATCH specifies that **JCL** is to be generated for execution by the Operating System in a batch region.

  DYNAM specifies that dynamic allocation is to be used instead of JCL. The job is then executed by the monitor in a batch region.

  TSO specifies that dynamic allocation is to be used instead of JCL in a background region. The job is then executed immediately in the foreground by the monitor under **TSO** instead of in the background.

- **MVS, F4, DOS, VS1, X8, HASP, ASP, JES1, JES2, JES3**

  In order to create the correct and most efficient instructions for each version of the Operating Systems, Jol must know which version it is creating instructions and which Jes system is in use.

- **NOCAT, NOCATGDG**

  Code **NOCAT** to stop Jol searching the catalog, and copying Volume and Unit information to the executable code.

  Code **NOCATGDG** to stop Jol searching the catalog for generation data sets.

- **PINCL, NPINCL**

  Code **NPINCL** or **NPI** to stop Jol printing **INCLUDE**d text.

- **PEXPAND, NPEXPAND**

  Code **PEXPAND** to see the code generated by any Macros.

- **PMACRO, NPMACRO**

  Code **PMACRO** or **PM** to have Jol print the Macro statements as they are interpreted. After the macro is debugged it is recommended that the default **- NPMACRO -** be used.

- **PJCL, NPJCL**

  Code **PJCL** to have Jol produce generated code with compiler listings. The generated JCL is also produced on another file ready for submission.

- **TEST**

  Code **TEST** to stop Jol producing messages to the Operator if Jol errors were detected. For Production jobs it is recommended that this option not be used.

- **LINECNT = nn**

  The **LINECNT** option specifies the number of lines per page of the Compiler printed output.

Merging Definitions - Overrides

Job, Program and Data Set Definitions may be overridden by simply Declaring the overriding Definition **before** the Definition that is being overridden.

For example, suppose the following DSID is declared, but it is required to change the SPACE and the DSNAME.

Original:	**DCL INPUT1 DS DSN AR.MASTER(+2)** **SPACE 10,2 CYLS SYSDA VB 7000,257;**
Required:	**DSNAME = AR.MASTER(+3)** **SPACE = 50,2 CYLS**

The following definition is placed before the original.

DCL INPUT1 DS DSN AR.MASTER(+3) SPACE 50,2 CYLS;

When Jol processes the second **DCL INPUT1,** the **VB 700,257** and **UNIT** of **SYSDA** will be merged with the details from the first declare, and the **DSN** and **SPACE** defined on the second Declare will be ignored.

A similar process occurs for Job and Program Definitions.

Multiple overrides are permitted, thereby making it possible to build a Definition piece by piece.

Notes

1. Overrides may appear in any order and do not have to be specified in the order in which they are coded in the Jol program.
2. The overriding statement must come first, therefore when overriding statements in an **INCLUDE** library, the overriding statement must be coded before the **INCLUDE**.
3. When merging information containing multiple values such as **TIME**, specify both values in the first statement.
4. To override a **DDNAME** entry of the type:

   A READS B

   in a Program Definition, the "**DDNAME** action **DSID**" must be referenced in full.

   If a DDNAME is to be nullified, it must be coded:**-**

   ddname action **'NULLFILE'**

ALLOCATE - *Immediate Instruction*

Purpose

The **ALLOCATE** instruction Dynamically Allocates Data Sets, including **SYSOUT** or Printer files. Allocated data sets can be accessed with the **OPENFILE, GETFILE**, **READ, PUTFILE** or **WRITE** instructions, or used with **INVOKE**d or **CALL**ed programs. The **ALLOCATE** instruction can be used instead of the **TSO ALLOCATE** or **JCL DD** card to allocate a data set.

```
{ ALLOCATE | ALLOC }    options

where the options are all those for the Data Set Declare plus:

    { F | FILE } { ddname }

    CATLG | DELETE | KEEP | SCRATCH | UNCATLG

    MOD | NEW | OLD | READ | SHR | UPDATE
```

Notes

1 **CATLG** - Specifies the data set name, volume and unit will be recorded in the system catalog. It is the default for *new* data sets.
DELETE - **DELETE**s the data set when it is **FREE**d.
KEEP - **KEEP**s the data set when it is **FREE**d. It is the default for *old* data sets.
MOD - Specifies data will be added to an existing data set.
NEW - Specifies the data set is *new*, that is, it does not currently exist. The final disposition of the data set will be as specified with the **CATLG, DELETE, KEEP** or **UNCATLG** disposition parameters.
READ | SHR - Specifies the data already exists, and is to be allocated as a Shared data set; other Users may also Read the data set at the same time.
SCRATCH - **SCRATCHE**s the data set when it is **FREE**d.
UNCATLG - **UNCATALOG**s the data set when it is **FREE**d.
UPDATE | OLD - Specifies the data set already exists. Allocation will enque it so that no other User can Read or Update the data set until it has been **FREE**d.

2 If the file cannot be **ALLOCATED**, **%LASTCC** is set non-zero; this condition can be tested with the **IF** statement. Successful allocation can be tested by coding:

```
IF %LASTCC=0 THEN ...
```

3 All allocation failure messages are the standard System Messages. Refer to the Operating System Message Manual in the case of difficulties.

1

2

Examples

1

```
ALLOC F(INPUT) SYS1.CONTROL.LIB(%DAY) SHR ;
                                        /* Opens member depending on the day */

OPENFILE INPUT INPUT;           /* Open the file */
IF ^ EOF(INPUT) THEN
     READ FILE(INPUT) INTO(A);
further processing ....
```

2

```
ALLOC FILE(OUTPUT) TEMP.OUTPUT NEW
          5 TRACKS
          FB 80, 800;
```

File **OUTPUT** is allocated to a new data set **TEMP.OUTPUT**. The data set is allocated **5 TRACKS**, with a Fixed Record Format of **80** byte records, and a Blocksize of **800** characters.

ASM - *Execute Command*

Purpose

The **ASM** Command is used to call the System Assembler to perform an assembly.

ASM { *data-set-name* | *dsid* **} [** *options* **]** ;

where options are:

{ PRINT { ***data-set-name* | *dsid* **} | NOPRINT }
{ SYSOUT * | *class* **}**

LIB { *dsid-list* | *dsname-list* **}**

OBJ { *dsid-list* | *dsname-list* **} | NOOBJ**

MACLIB | NOMACLIB
ALIGN | NOALIGN
BATCH | NOBATCH
ALOGIC | NOALOGIC
BUFSIZE (**STD | MIN**)
ESD | NOESD
FLAG (*number*)
LIBMAC | NOLIBMAC
LINECNT (*linecount*)
LIST | NOLIST
MCALL | NOMCALL
MLOGIC | NOMLOGIC
NAME | NONAME
NUM | NONUM
RENT | NORENT
RLD | NORLD
STMT | NOSTMT
SYSPARM ('*system parms*')
TERM | NOTERM
TEST | NOTEST
XREF (**FULL | SHORT**) **| NOXREF**

Notes

1 **SYSOUT** defaults to **SYSOUT=%TSOCLASS** so that the Time Sharing System can retrieve the output.

2 **OBJ** defaults to a temporary data set **&OBJ**. It may be referenced by the DSID **OBJ** for the **LINK** Command.

3 If **LIB** and **MACLIB** are specified, **SYS1.MACLIB** is concatenated to the libraries specified.

4 See also the **COMPILE** command.

Example

1
```
ASM SOURCE (PAYROLL);
LINK OBJ LOAD(TEST.LOAD(PAYROLL) );
RUN PAYROLL;
```

ASSIGNMENT or SET - *Immediate Instruction*

Purpose

The **SET** or **ASSIGNMENT** instruction is used to change or set the value of a Symbolic Variable.

```
     [%]name = {expression | function-expression } }
or
     SET name  = {expression | function-expression } }

where expression is:

                          [  +                                ]
     { symbolic-variable  [  -        { symbolic-variable   }  ]
     { constant           [  *        { constant            }  ]
     { 'character-constant'} [ /      {'character-constant'  }  ]
                          [  ||                               ]

and function-expression is one of:
     1.   SUBSTR (symbolic-name,start-position [,length])
     2.   INDEX (symbolic-name,'search string')
     3.   TYPE (symbolic-name)
          returns 'LIT', 'CHAR', or 'NUM'
     4.   LENGTH (symbolic-name)
     5.   EOF (file-name)
```

Notes

1. If the variable is not already Declared, it is implicitly declared at the current level.

2. When used in a Macro, and if the name is not defined nor declared **EXTERNAL** to the Macro, it is implicitly declared at the same level as the Macro, and the value will not be available after the Macro terminates.

3. The **SET** may be used to indirectly alter the value of a Symbolic Variable by using an existing symbolic variable to specify the name of the variable to be altered. For example:

   ```
   X = 'ABC';
   SET %X = 1;
   ```

   After execution of the **SET** instruction, the symbolic variable **ABC** will contain the **1**, and symbolic variable **X** will be unaltered.

4. The maximum length of any assignment value is 253 characters. Assigning more than 253 characters will result in truncation.

5. All arithmetic is performed in integers with 13 digit precision. Thus:

   ```
   A = 5 / 3;
   ```

   results in 1 being assigned to **%A**.

6. Expressions containing more than one operator are not yet supported **-** complex expressions must be broken into two or more statements.

7. The names may be preceded by a **%** symbol. They may be from one to eight alphameric or national (**$, # or @**) characters.

   For example:

   > **A** or **%A**
   > **%123**
   > **$NAME** or **%$NAME**
   > **GEN** or **%GEN**

8. All **symbolic variable** contents are held in *character format*, even if they are numeric. When arithmetic operations are performed on symbolic variables, they are first converted to decimal packed numbers. If errors are detected during this conversion, an error is signalled and the result is forced to zero. The maximum number of digits in a number is allowed to be thirteen (13) : longer numbers are treated as characters

Examples

1 **A='ABC';**

The symbolic variable **A** is set to the characters **ABC**. Whenever **%A** is now written (except in an assignment), the characters **ABC** will replace it.

2 **NUM = B + 5;**

The symbolic **NUM** is set to the current contents of **B,** plus 5. If **B** contained a 2, then **NUM** would now contain 7.

3 **CHAR ='%A%B%C';**

The symbolic **CHAR** is set equal to the result of concatenating the current contents of the symbolic variables **A, B** and **C**.

4 **NAME='MASTER.FILE' || %NUM;**

The symbolic variable **NAME** is set to the characters **MASTER.FILE** concatenated to whatever the current contents of **NUM** are. If **NUM** contained **02**, then **NAME** would equal **MASTER.FILE0**.

5 **NAME = 'SET A = B+C';**
...
%NAME;

Sets **A** to the result of **B** plus **C**.

The **%NAME** statement is replaced by the current value of **%NAME** which, in this case, is an assignment or **SET** instruction. It is then executed. Other types of instructions may also be set up in this manner.

6 This example is a macro that splits variables into two if the length of the information contained in the symbolic variable **OP** is greater than 71 characters.

```
1     SPLITOP: MACRO;
2     L = LENGTH (OP);
```

```
IF %L>72
THEN DO;
LINE2 = SUBSTR (OP,72);
OP = SUBSTR (OP, 1,71);
END;

ELSE LINE2 ='';

END;
```

The **SPLITOP** command above checks the number of characters in **OP** and if there are more than 72 characters it will place the 73rd and subsequent characters in **LINE2**, and truncate **OP** to 72 characters. If there are fewer than 72 characters, **LINE2** will be cleared and **OP** will remain unaltered.

BUILDGDG - *Execute Command*

Purpose

The **BUILDGDG** command creates an index in the Catalog for Generation data sets. Options specify how many data sets are to be retained, and the action to be taken to data sets when the specified maximum have been cataloged.

{BUILDG | BUILDGDG | BUILDG }

index [,*index*....] *options*;

options		defaults
CVOL	= *cvol*	**-**
UNIT	= *unit*	**-**
DELETE		**DELETE**
NODELETE		
EMPTY		
ENTRIES	*maximum*	**3**
MAXDS	*number of datasets in the group*	
OWNER		**-**

Notes

1. Jol and/or the Operating System automatically builds and deletes indexes for **Non-generation** data sets, but before any **Generation Data Sets** can be cataloged, an index must be built.

2. Up to *five* (5) index levels may be built in one **BUILDGDG** Command.

3. To build Indexes with a differing number of **ENTRIES**, two or more **BUILDGDG** Commands must be used.

4. **ENTRIES** specifies the maximum number of entries to be contained in the generation index. It may not exceed 255.

5. **EMPTY** specifies that all data sets are to be uncataloged when the number of data sets specified in the **ENTRIES** parameter has been exceeded. Otherwise only the oldest data set(s) will be uncataloged.

6. **DELETE** specifies that the data sets whose names are removed from the catalog when the maximum number of entries has been exceeded are to be also Deleted.

Examples

1 **BUILDGDG SALES.MASTER**
ENTRIES 15;

```
1   BUILDGDG      SALES.MASTER
                  ENTRIES 15;
```

The **BUILDGDG command** will create an index that will allow no more than 15 **SALES.MASTER** generation data sets. When 15 data sets have been created, the oldest will be **DELETEd** when more data sets are **CATALOGed** because **DELETE** is a default option.

2
```
BUILDGDG      TEST.INPUT
              TEST.OUTPUT
                 NODELETE;
```

The **BUILDGDG** will create two indexes for the maximum of three data sets in the groups **TEST.INPUT** and **TEST.OUTPUT**. Old data sets will be uncataloged but not deleted when the limit of three has been exceeded.

BUILDJOB - *Interactive Command*

Purpose

The **BUILDJOB** Command asks you step by step, what programs to run, what condition codes are needed for continuing the job, and whether or not data sets are to be cataloged, deleted, or printed at the end of the job.

BUILDJOB leads the inexperienced User through all the steps required to build a Jol jobstream to a family of elaborate programs within a jobstream. These programs can be used by other Users.

```
BUILDJOB ;
```

CAIJOL - *Interactive Command*

Purpose

The **CAIJOL** command is an interactive Computer Aided instruction Course providing a teach yourself facility for all Jol Users.

```
CAIJOL;
```

The Course is approximately 3 hours long, and includes question and answer reviews.

You may break and restart at any time.

CALL - *Immediate Instruction*

Purpose

The **CALL** instruction loads and executes a program. Files needed by the program may be allocated with the **ALLOC** instruction (TSO or Jol) or with JCL.

```
CALL    'data-set-name(program-name)'
        ['parameters'] ;
```

Notes

1. If the **CALL** instruction cannot be executed (for example the data set is not available or an ABEND occurs), **%LASTCC** is set non-zero; this condition can be tested with the **IF** statement. Successful execution can be tested by coding:

   IF %LASTCC=0 THEN ...

2. The program is executed with the ATTACH Operating System instruction.

3. **CALL** is typically used with the **ALLOCATE** instruction. A **CALL**ed program runs in exactly the same manner as a program run in Background with one difference **-** special programs may be written that alter Jol internal variables. The *Jol System Programmer Guide* contains details on *Extending the Language*.

Examples

1 **ALLOC F(INPUT) SYS1.CONTROL.LIB(%DAY) SHR ;**
/* *Opens member depending on the day* */

CALL 'SYS2.LINKLIB(CHECKSHD)'; /* *Check Schedule* */

File **INPUT** is allocated to data set **SYS1.CONTROL.LIB(%DAY)**. If the day is **MONDAY** then member **MONDAY** will be allocated as the member.

Program **CHECKSHD** is then called from library **SYS2.LINKLIB**.

CATALOG - *Execute Instruction*

Purpose

The **CATALOG** instruction **KEEPs** a Data Set, and also enters the Data Set Name, its Volume, Unit and Position (if the Data Set is on a Tape) into the System Catalog; this means that when the data set is required to be read at a later time, the User need not specify Unit or Volume information as this will be obtained from the System Catalog.

```
      CATALOGUE
or    CATALOG     { dsid   } [ (generation- number) ] . . .
or    CATLG  { dsname}

          [ON [VOL] volume [UNIT] unit]
          [ALWAYS] ;
```

When a new data set is created by a program that **WRITES** a Data Set, it is automatically deleted by the Operating System unless it is **KEPT** or **CATALOGED**. The **KEEP** or **CATALOG** instruction ensures that the Data Set will be available for other jobs when the currently executing job terminates.

Notes

1. The '*dsname*' form of the instruction is not recommended for long production jobs as the **DSID** format allows the Data Set Name to be overridden; thus a data set name may be altered in one place **-** the **DECLARE -** and all references to the **DSID** will have the new Data Set Name.

2. If Volume and Unit information are provided in the Data Set Declare or the **ON** option is used for Cataloging a Data Set, Jol does not check that the Data Set is on the specified volume; the data set name and Specified Volume and Unit will be entered in the System Catalog.

3. The **ON** or **FROM** options, if specified, must be coded after the **DSID** or **DSNAME** list.

Examples

1
```
CATALOG DSID1, 'TEST.DATA.SET';
```

2
```
DECLARE OP1 DATA SET
           COPY.SYS1.LINKLIB
           3380  20,10,100 CYLS;

COPY SYS1.LINKLIB TO OP1 PDS;
CATALOG OP1 ;
```

The Declare **OP1** defines a data set called **COPY.SYS1.LINKLIB**. The **COPY Command** copies the **SYS1.LINKLIB** data set to the **OP1** definition, then the **CATALOG OP1** instruction enters the data set name **COPY.SYS1.LINKLIB** and its volume and unit into the System Catalog.

Note that the definition of **OP1** does not specify a volume; the system will allocate a volume when the **COPY** is performed, and then the catalog function will find which volume the data set was placed on, and enter the volume name into the catalog.

CLOSFILE - *Immediate Instruction*

Purpose

The **CLOSFILE** instruction is used to close a file after either input or output actions have been performed.

```
CLOSFILE  filename ;
```

Notes

1. The file must have been previously **OPEN**ed with the Jol **OPENFILE** instruction.
2. **%LASTCC** is set to zero if file was closed successfully, otherwise it is set non-zero.
3. See also the **ALLOC, OPENFILE, PUTFILE, READ, GETFILE, WRITE,** and **FREE** instructions.
4. Files opened with the Jol **OPENFILE** instruction are automatically closed at the end of the Preprocessor phase of Jol if they are not specifically closed with the **CLOSFILE** instruction.

   Files not closed by **CALL**ed or **INVOK**ed programs are closed by the operating system when Jol ends.

Examples

1

```
CLOSFILE OUT;              /* Close File OUT */
```

File **OUT** is closed. The buffers are freed and returned to the Operating System.

COBOL - *Execute Command*

Purpose

The **COBOL** or **COB** command calls the **COBOL** compiler to convert the **COBOL** source to object code. The object code then can be input to the **LINK** or **LOADGO** command and executed.

{ COBOL | COB } {*dsid* | *data-set-name*} [options]

where *options* are:

{ PRINT **{***dsid* | *data-set-name***} | NOPRINT** **}**
{ <u>SYSOUT</u> **<u>*</u>** | *class* **}**

LIB { *dsid [*|| **...** *]* | *data-set-name [*|| **...***]* } | **NOLIB**

OBJ `{*dsid* | *data-set-name* **} | NOOBJ**
TEST `{*dsid* | *data-set-name* **} | <u>NOTEST</u>**
<u>APOST</u> | QUOTE
BATCH | <u>NOBATCH</u>
BUF (*number* | *number* **K**)
CDECK | <u>NOCDECK</u>
CHECK | <u>NOCHECK</u>
CLIST | <u>NOCLIST</u>
COBOL
COPY | <u>NOCOPY</u>
COUNT | <u>NOCOUNT</u>
CSYNTAX | <u>NOCYNTAX</u>
DECNT (**<u>NO</u>** | *number*)
DMAP | <u>NODMAP</u>
DYNAM | <u>NODYNAM</u>
FILE (**<u>STD</u>** | 052)
<u>FLAGW</u> | FLAGE
FLOW (*number*) | **<u>NOFLOW</u>**
IDENT | <u>NOIDENT</u>
INTERRUPT (**<u>STD</u> | OS2**)
<u>LCOL1</u> | LCOL2
LIL | <u>NOLIL</u>
LINECNT (*number*)
LINK | LINK1
LSTONLY | LSTCOMP | <u>NOLST</u>
NAME | <u>NONAME</u>
<u>NUM</u> | NONUM
OPTIMIZE | OPT | <u>NOOPT</u>
<u>PANEL</u> | NOPANEL
PMAP | <u>NOPMAP</u>
RENT | <u>NORENT</u>
RPT (**<u>NEW</u> | OLD**)
<u>SDS</u> | NOSDS
SEQ | <u>NOSEQ</u>
SIZE (*number* | *number* **K**)
<u>SOURCE</u> | NOSOURCE
<u>SPACE1</u> | SPACE2 | SPACE3
STATE | <u>NOSTATE</u>

<u>SUPMAP</u> | NOSUPMAP
<u>SXREF</u> | NOSXREF
SSIN (*ddname*)
SSOUT (*ddname*)
TERM | <u>NOTERM</u>
SYMDUMP | <u>NOSYSDUMP</u>
<u>TRAP</u> | TRAPC
TRUNC | <u>NOTRUNC</u>
VBREF | <u>NOVBREF</u>
VERB | <u>NOVERB</u>
<u>XREF</u> | NOXREF
<u>ZWB</u> | NOZWB

Notes

1. With a Time Sharing System **SYSOUT** defaults to **%TSOCLASS,** so that the output can be viewed at the terminal. Otherwise, **SYSOUT** defaults to the installation defined **SYSOUT** class.

2. **OBJ** defaults to a temporary data set **&OBJ**. It may be referenced by the DSID **OBJ** for the **LINK** command.

3. See also the **COMPILE** command.

Example

1
```
COB SOURCE (UPDATE);
LINK OBJ LOAD TEST.LOAD (UPDATE) COBOL;
RUN UPDATE;
```

COMPARE - *Execute Command*

Purpose

The **COMPARE** command will compare:

- Two sequential data files.
- Two partitioned datasets.
- A member of a partitioned dataset to another member of a partitioned dataset, or a sequential dataset.

```
COMPARE    { dsid  |  dsname }

              TO      { dsid | dsname }

                      [PDS | PO] ;
```

where:

PO | PDS	specifies that an entire partitioned data set is to be compared with another

Examples

1 **DCL NEW DS MASTER.FILE(+1) ;**
DCL OLD DS MASTER.FILE(0) ;

```
   COMPARE OLD TO NEW ;
```

The data referred to as dsid **OLD** is compared to the data referred to by dsid **NEW**. In this example, the data sets **MASTER.FILE(0)** and **MASTER.FILE(+1)** are compared.

2 **COMPARE 'SYS1.PROCLIB' TO 'NEW.PROCLIB' PDS ;**

The data sets **SYS1.PROCLIB** and **NEW.PROCLIB** are compared. Any missing members names will be noted, and any differences in each member will also be noted (to a maximum of 10 per member).

3 **COMPARE 'SYS1.PROCLIB(PAYROLL)'**
TO 'SYS1.PROCLIB(OLDPAY)' ;

The comparison occurs between two members of the same data set members **PAYROLL** and **OLDPAY**.

COMPILE - *Execute Command*

Purpose

The **COMPILE** command calls the appropriate language compiler to compile a program. If the **LINK** option is specified, the program is link edited into the Load Module library specified on the **REGISTER** command (unless the **LOAD** option is specified, thereby temporarily specifying a new library).

COMPILE [{ & | AND} LINK]
prog1 [,prog2 ...] options ;

where options are:

COMPOPT	**-**	over-riding compiler options
COPYLIB	**-**	over-riding library for **COPY** or **INCLUDE** statements in **COBOL, PLI** or Assembler programs
LINKOPT	**-**	over-riding linkage editor options
LIB	**-**	over-riding **AUTO-CALL** libraries
LOAD	**-**	over-riding **LOAD** Module library
ALWAYS	**-**	allow link editing even if compile errors occur.

Notes

1. If the **COMPOPT, COPYLIB, LINKOPT, LIB** or **LOAD** parameters are not specified, those specified originally in the **REGISTER** command are used.
2. If multiple program names are specified, they will all be compiled and will form **one** Load Module or executable program. Separate **COMPILE** commands must be used to create separate load modules.
3. If the **LINK** option is not specified the object text is written toa temporary data set that may be referred to by the DSID **OBJ**. Thus, **LINK OBJ LOAD ('TEST.LOAD (VALIDATE)')** will link the object text created by any previous **COMPILE** commands and store the executable Load Module in the **TEST.LOAD** data set under member **VALIDATE**.
4. The **COMPILE** command must be used in conjunction with the **REGISTER** and **EXEC** commands.
5. See also the **ASM, COBOL, PLI** and **LINK** commands.

Examples

1 **COMPILE AND LINK VALIDATE;**

The program **VALIDATE** is compiled using the compiler specified when the program was Registered (that is, **Assembler, COBOL, FORTRAN** or **PL/1)**. If no serious compiler errors occurred during compilation, the load module is stored in the library specified when the program was registered.

2 **COMPILE UPDATE;**

LINK OBJ LOAD('PRODN.LOAD(UPDATE');

The program **UPDATE** is compiled and the object text written to a temporary data set that may be referred to by **OBJ**. Subsequently the **LINK** command is used to link edit the program and store it in the **PRODN.LOAD** library.

COMPRESS - *Execute Command*

Purpose

The **COMPRESS** command will reorganize or remove unused space from a library or partitioned data set.

> **COMPRESS** {dsid$_1$, dsid$_2$ | dsname$_1$, dsname$_2$}
> **[SHR] ;**

Compressing Libraries

Whenever a member is replaced in a library, the spacethat it previously occupied is left unused. Eventually, there is no room left in the data set to add new members, or replace existing ones, and a reorganization is required. The **COMPRESS** moves all the active members to the start of the data set **-** thus forming contiguous data. The unused portions of the data set are placed at the end of the data set, and so more members can be added to it.

Notes

1. The data set is enqueued, so no other User can use it until the job ends. The **FREE** command can be used to free the data set before the job ends.

2. Up to fifteen libraries can be compressed in one **COMPRESS** command.

3. The **SHR** option allows other jobs to continue accessing the data during the **COMPRESS**. However, if another program updates the data at the same time, the data set may be destroyed.

Example

```
1   COMPRESS
            PDS
            SYS1.MACLIB;
```

The **COMPRESS** command is requested to compress two libraries. The first is a **DSID,** and it will refer to a library. The second library is named in the command, and is **SYS1.MACLIB**.

Both libraries will be compressed.

COPY - *Execute Command*

Purpose

The **COPY** command copies data from one data set to another. The data may be Sequential, **VSAM,** Indexed Sequential, or a Library (Partitioned Data Set).

COPY { *dsid* | *dsname* **}**
TO { *dsid* | *dsname* **}**
[options] ;

The default if no options are specified is a **sequential** dataset copy.

options

[**OLD**] **|** [**SHR**] **|** [**MODS**] **|** [**EXTEND**]

For Vsam Data Sets, code:

VSAM

For Indexed Sequential Data Sets, code:

ISAM | IS
[**PRIME** = *prime-dsid*]
[**INDEX** = *index-dsid*]
[**OFLOW** = *oflow-dsid*]

For Partitioned Data Sets, code:

PDS | PO
{ SELECTING | SELECT | S } *member*
MEMBERS | MEMBER | M (*member-name-list*)
REPLACE | R

The default *is* Sequential, but you *may* code:

SEQ

where:

input-dsid or *dsname*	is the data to be copied.
output-dsid or *dsname*	is the data set in which the copied data is to be placed. Unless **OLD** is coded, new space will be allocated, that is the data set is assumed to be **NEW**.
prime-dsid, *index-dsid*, and *oflow*, *oflow-dsid*	are areas of an Indexed Sequential data set.
member or *member-name-list*	is the name or names of members in the library to be copied.
OLD	specifies that the **"to"** data set already exists.
SHR	specifies that the **"to"** data set already exists. It also specifies that the operating system will not request exclusive use of the data set, that is, other users may read the data set while it is being updated. Use this parameter with caution.
MODS | EXTEND	specifies that the data being copied to the data set will be placed after any existing data set. The data set is extended with the new data. If the data set does not already exist it will be created, but unless a **KEEP** or a **CATALOG** instruction is executed

	referring to that data set it will be deleted at the end of the job.
ISAM or **IS**	specifies that the data set is an Indexed Sequential data set.
PDS or **PO**	specifies that the **"from"** data set is a library. If the **"to"** dsid is a tape, an unload function takes place; if the **"to"** dsid is a direct access data set and the "from" dsid is a tape created from a library with a **COPY command,** then the library will be reloaded.
SEQ	specifies a sequential copy.
SELECT	specifies a name or names of the members of the library that are to be copied. If the names already exist in the **"to"** data set, they will not be **REPLACED** unless the **REPLACE** option is specified.
REPLACE	specifies that if members are already in the **"to"** data set, they are to be replaced by the members in the **"from"** data set.

Notes

1 The data in the data set referenced by the first data-set-identifier is copied to the data set referenced by the second data-set-identifier.

2 If you wish to overwrite the information in the second data set, or if you wish to add to an existing **Partitioned Data Set** or **Library,** code the **OLD** keyword.

3 If you are not copying a Sequential Data Set, or **SELECTing** members for a **Partitioned Data Set** copy, you must specify the type of data set being copied by using the **ISAM, IS, PDS** or **PO** keywords.

The Default is a SEQUENTIAL copy.
Code **IS** or **ISAM** for an **Indexed Sequential Data Set**.
Code **PO** or **PDS** for a **Partitioned Data Set**.

4 To copy **SELECT**ed members from a **Partitioned Data Set** code the **SELECT** or **MEMBER** keywords, followed by the member name(s) you wish to copy.

If you wish to overwrite existing members of the name(s) you are copying, you must code the **REPLACE** or **R** keyword.

If there is insufficient space to hold the members, the **COMPRESS** command may be used to reorganize the library.

5 The **PRIME, INDEX** and **OFLOW** parameters are used when copying an **ISAM** data set which requires **PRIME, INDEX** and **OFLOW** data sets.

Examples

1. **COPY INPUT TO OUTPUT;**

The data from the **DSID called INPUT is copied to the data set specified by the DSID OUTPUT. The default is a sequential copy.**

2.
```
COPY TEST.PROCLIB
     TO SYS1.PROCLIB OLD
     SELECTING (PAYROLL,ARPROC)
```

```
        REPLACE;
```

TEST.PROCLIB is the input data set, and members **PAYROLL** and **ARPROC** are to be copied into the system data set **SYS1.PROCLIB.** If members **PAYROLL** and **ARPROC** are already in **SYS1.PROCLIB,** they are to be replaced with the members from **TEST.PROCLIB.**

3.
```
DCL X DS TAPE
    COPY.PRODUCTN.PROGS(+1);

COPYONE:
            COPY PROD.LIB TO X PDS;

            IF COPYONE = 0
            THEN CATALOG X;

            ELSE STOP 'COPY OF PRODUCTION PROGRAMS FAILED';
```

The declare of **X** specifies a Tape data set called **COPY.PRODUCTN.PROGS.** The **COPY** statement with the label **COPYONE** copies the library **PROD.LIB** to **X** (the **COPY.PRODUCTN.PROG(+1)** data set). The **COPY** has **PDS** specified, and so the **COPY** will create an unloaded version of the library on the tape.

The **IF statement** then tests the stop **COPYONE.** If it returned a zero, the copy data set is cataloged, otherwise a message is passed to the operator.

DECLARE and DEFINE Statements

Purpose

Jol allows and sometimes requires that items be *declared* or *defined*. These definitions are then used, as in other high-level languages, in instructions and commands.

The following types of definitions are available:

Card Image:	See next page.
Data Sets:	Page 66.
Printer Files, **Card Punches**	See **PRINT Definition**, page 72
Programs:	See **Program Definition**, page 74.
Symbolic Variables:	See page 79.

Additionally, how to specify **Record Formats** is detailed on page 26, and how to specify **SPACE** for data sets is described on page 29.

Declaring Card Image Data Sets

Purpose

The **Declare** * instruction defines the beginning of card image data. The actual data cards images follow the definition, and to inform the system where the last card image is, an **EOF;** is coded in *columns one to four*.

```
{ DECLARE    }
{ DEFINE     }   dsid-name   *  [options] ;
{ DCL        }

followed by EOF;

options are:

     [  REPLACE  ]

     [  PRINTNONE | PRINTALL  ]

     [  SPLIT | NOSPLIT | JOL  ]

     { FB  }      { recordsize,blocksize     }
     { VB  }      { blocksize,recordsize     }
                  { recordsize               }
```

Card Image Input

Card image files are one of a number of methods of inputting information to a computer. Card image data sets are used to input data to processing programs, or instructions to utility programs. Small or large files may be included within your Jol program, but on some occasions large card image files are better read directly from a disk data set while the job is executing.

> *Note:* Card image data sets can be placed in Jol **INCLUDE** or command source code. This effectively allows you to store data in JCL cataloged procedures.

Card Images Containing Varying Information - REPLACE Option

Some programs, usually utilities, require instructions and parameters on card image files. The **SORT** program is such a utility program; it requires, for example, the sort control field information in **FB** 80, *nn* format.

To provide maximum flexibility, you may write symbolic variable names on card image files, and the current value of the symbolic variable will replace the name. After replacement of symbolic data, the card image may exceed 71 characters in which case a continuation card will be created with data commencing in column 16, after placing an asterisk in column 71 of the first card. Other options described below can alter the action taken if a card image exceeds 71 or 80 characters.

To create a card file with the day and date in the first card, code similar to:

```
DCL CARDFILE * REPLACE;
%DAY%DAYNO
EOF;
```

If the day was **MONDAY** the 14th, then the card would be:

MONDAYbb**14**

The program can read the card image file and use the day and day number for any purpose it desires, for example to print headings. This facility is most often used in Jol commands to create utility control cards from **variable parameters** used in the command. However, this facility of text replacement is not restricted to to creating parameter files for utility programs **-** it can be used for any program.

Notes

1. The first twenty cards are listed unless the card file definition is made in a **Macro** Command, whereupon no cards are printed. **-PRINTALL** will list **all** the cards,**PRINTNONE** will **not** list **any** cards.

2. If the **DECLARE** *name* *; is bypassed after an immediate **IF** statement, the cards are totally ignored.

3. The declared cards are copied to a temporary data set and therefore may be read more than once.

4. The **EOF;** statement <u>*must be coded in columns one to four*</u> (1-4) of the card following the last data card. Otherwise all subsequent cards, including any Jol statements, are appended to the data cards and lost.

5. The **VB** or **FB** parameters indicate the record format in which the processing program expects the file. If **FB** is chosen, the record length may not be greater than 80. Similarly if **VB** is chosen as the record format, the maximum record size is 84.

6. The **REPLACE** option specifies that all symbolic variable names found in the input stream are to be replaced with the current contents of the variables. If, after symbolic variable replacement, the length of the information on a card is greaterthan 71 characters, the following occurs:

   - If **SPLIT** is coded, the card image is truncated at column 80, and any excess information is copied to the second and subsequent card images.
   - If **NOSPLIT** is coded, the card image is truncated at column 80.
   - If **JOL** is coded, the card image is split at column 72 and the excess information copied to the second and subsequent card images.
   - Otherwise, two or more card images are created, the first card terminating with a convenient comma and an asterisk placed in column 72, and the remaining information commencing in column 16 on any subsequent cards.

Examples

1

```
DECLARE CARDIN *;
CARD1
EOF;
```

This declaration defines a data set identifier **(DSID)** which may be read or printed. It contains one card.

2

```
DCL CARDS * VB 84,30;
THIS CARD IMAGE FILE WILL BE MADE
INTO VARIABLE LENGTH RECORDS
EOF;
```

This declaration defines a data set identifier **(DSID)** that will have the characteristics of a **Variable Blocked** data set when a program reads it.

3

```
DECLARE %SYMBOL INIT '%DAY';

EDIT SYMBOL SYMBOL A(3);
DCL CONTROL * REPLACE;
```

```
%SYMBOL%DAYNO%MONTHNO%YEAR
EOF;
```

These statements create a **DSID** called **CONTROL** which will have:-

Columns	**Data**
1-3	*day*, e.g. THU
4-5	*day number*, e.g. 9
6-7	*month number*, e.g. 7
8-11	*year number*, e.g. 1987

THU09071987

This control file can be used by a program to print headings on a report.

Declaring Data Sets

Purpose

The **DECLARE** data set definition describes details about the datasets that may be used in the job. Datasets to be used for input require a **DSNAME**, and, if not cataloged, **VOLUME** and **UNIT** information. Datasets to be used for output may also require **SPACE** and **DCB** information unless the installation Defaults suffice.

	{ DECLARE }
	{ DEFINE } *dsid-name* **{ DATASET | DS }** *options* **;**
	{ DCL }
	
	where *options are:*

Special Jol Options	**LIKE** [=] *earlier-dsid* **EXTERNAL | EXT** **NOTCATLG | NOCAT**
Dsname Parameters	**[NAME]** **{** *dsname* **[(***member-name***)]** **[DSN]** **{** *&&temp-name* **[(***generation number***)]** **[DSNAME]** **{** *'dsname'* [*/password*] *or* **{ NODSNAME | NODSN | NONAME }** *or* **DUMMY | NOIO**

Record Formats	**{** *record-size,block-size* **}**
	record format **{** *block-size,record-size* **}**
	{ *block-size* **}**
	
	or **NODCB | NODEFAULT | NODEF**

Volume Parameters	**{ VOLUMES | VOLUME }** [=] *volume [volume ...]* **{ VOLS | VOL }** *or* **VOLREF** [=] **{** *dsid* | '*dsname*' **}** *or* **MSVGP** [=] *mass-storage-volume-group [,ddname]* [**PRIVATE**] **{ MAXVOLS | MAXVOL }** [=] *maximum-number-* **{ VOLCOUNT }** *of-volumes* **{ VOLSEQ | VSEQ }** [=] *volume-sequence-number*
Space Parameters	*n1* [-*n2*] **{ RECORDS | RECS } [OF** *modal-record-length]* or [**SPACE**] [=] *primary allocation* **[CYLS CYL]** **[** *, secondary space* **]** **[TRACKS TRKS TRK]** **[[, {** *directory-blocks* **]]** **[BLOCKS BLOCK]** **[[{** *index-blocks* **]]** **[BLKS BLK]** *or* **ABSTR** [=] primary-allocation,track-number

	[, { *directory-blocks* | *index-blocks* **}]**
Unit Parameters	**{ UNIT | UNITS }** [=] *unit-type [,no-of-units* **|** ***P*** *]* **{** { **UNITNO | UCOUNT** } **}** [=] *no-of-units* **| *P*** **{ PARALLEL** [**MOUNT**] **}**

Tape Options	**{ FILENO | FILESEQ }** [=] *file-start-*
	{ POSITION } *position-on-tape*
	
	{ SL | SUL | NL | BLP | AL | AUL | LTM | NSL }
	
	{ { **800 | 1600 | 6250** } **BPI }**
	{ DEN = { **0 | 1 | 2 | 3 | 4** } **}**

Retention Period Parameters	[**RETAIN**] **{ FOR** *n* [**DAYS**] **}**
	{ { **UNTIL | TILL | TIL** *date* } **}**

Freeing Resources	**FREE** [**AT**] [**END | CLOSE**]
Data Organizations	[**DSORG**] { **DA | IS | PS | PO | PSU | ISU | DAU | POU** }
VSAM options	**VSAM** Forces AMP for Recfm, Bufno etc.
ISAM options	**CYLOFL** [=] *number-of-tracks* **KEY** [=] *key-length,key-position* *or* **{ KEYPOS | KEYPOSN** [=] *key-position* **}** **{ KEYLEN** [=] *key-length* **}**
Buffer Options	**{ BUFFNO | BUFNO }** [=] *number-of-* **{ BUFFERS | BUFFER }** *buffers* **{ BUFLENGTH | BUFLEN | BUFL }** [=] *buffer-length*

Error Options	**{ DIAGNS** [=] **TRACE }**
	{ TRACE }
	
	{ ACCEPT | ACC }
	EROPT [=] **{ SKIP | SKP }**
	{ ABEND | ABE }
	
	{ SKIP | ACCEPT [**ERRORS**] **}**

Protection Options	**{** { **READ | READONLY | READ ONLY** } **}**
	{ PROTECT }

Sundry Options	**QNAME** [=] *qname* **CODE** [=] **A | B | C | F | I | N | T** **OPTCD** [=] **A,B,C,E,F,H,I,L,M,O,Q,R,T,U,W,Y** *and/or* **Z**

	DCBEXTRA = *'any dcb information not currently supported'*

Notes

1 The **DSNAME** keyword need not be coded when using data sets of the form ***name1.name2***. For example:

SHELL.PGMLIB

2 To read or write a particular member of a Partitioned Data Set or Library, code the member name required in parentheses after the Data Set Name. For example:

SORT.CONTROL(PAYSORT)

3 When generation data set names are requested, the relative or absolute generation number must be coded in brackets after the data set name, otherwise the entire group of data sets will be accessed.

A **NEW** generation of a data set is requested by coding (+ ***relative*** *generation number*). For this type of output data set a **CATALOG** instruction must be issued in the Jol program if the data set is to be made available automatically in another job.

An **OLD** generation of a data set is requested by coding (**-** ***relative*** *generation number*) or (***absolute*** *generation number*).

The **LATEST** generation of such a data set is obtained by coding **(0)** after the data set name. For example:

Dsname	Action
FILA(0)	locates the ***latest*** copy of **FILA.**
FILA(+1)	is used when a ***new*** generation of **FILA**!a is to be created.
FILA(-1)	locates the ***previous*** generation of **FILA.**
FILA(15)	locates ***absolute*** generation **15** of data set **FILA.**

See also "**Program Definition**", note 7.

4 Unlike **JCL**, Symbolic Variable replacement is not terminated by ('.'). Instead replacement is terminated by a blank, special character or another Symbolic Variable. This considerably reduces the number of periods that must be coded when using Jol. But when defining certain types of **DSNAMES**, the concatenate symbol must be used.

To code the following **DSNAME**

%MONTH<u>01</u>.GEN(+1)

where **01** is a constant, and not part of the Symbolic Variable name **%MONTH,** the following must be done:

%MONTH||01.GEN(+1)

If **%MONTH** is **JAN**, the **DSNAME** will be

JAN01.GEN(+1)

5 For **ISAM** files, the keywords **PRIME**, **INDEX** or **OFLOW** must be coded in brackets following the Data Set Name to reference or create **ISAM** files with independent areas.

6 Sometimes, no **DSNAME** is allowed (for instance when using the **IEHMOVE** Utility). In such cases code **NONAME** or **NODSN**, otherwise Jol will assume that you are attempting to read a Temporary data set and issue an error message.

7 The **VOLUME** keyword (and the **UNIT** keyword and unitype) need not be coded if the volume is one specified to Jol at installation time.

8 The **VOLREF** is used to refer to a previously defined DSID or cataloged data set name.

The **DSID** referred to may be a Cataloged Data Set, contain a Volume Reference itself, or bea data set created before the step using this declaration is executed, in which case the Operating System will find the correct volume.

VOLREF is used mainly for writing more than one data set on the same **SCRATCH** Volume but has other uses such as copying Generation Data Sets onto alternating volumes.

9 The **UNIT** keyword need not be coded when using any **IBM** or **FACOM** unit name such as **F493, 3330, 2400-3** and so on, or when specifying one of the Units specified

when Jol was installed.

The following units are automatically known by Jol**:**

> **DISK, DISC, TAPE, DUAL, PUBLIC, SYSDA, SYSSO**

and

> **2301, 2302, 2303, 2305-1, 2305-2, 2311, 2314, 2319, 2321, 3330, 3330-1, 3340, 3350, 3375, 3380, 2400, 2400-1, 2400-2, 2400-3, 2400-4, 3400-1, 3400-2, 3400-3, 3400-4, 3400-5.**

plus any installation defined units.

10 Use **NOCATLG** or **NOCAT** when Jol is not to search the System Catalog for the data set.

11 The **LIKE** parameter may be used to copy any undefined items from an earlier **DSID**. It has two main uses:

a) To save coding effort
b) A data set may be redefined with different **DCB** parameters but have the bulk of the information taken from the original definition.

12 The **DCBEXTRA** is used to code any rarely used **DCB** options. For MVS, these are:

> **BFALN =, BFTEK =, FRID =, FUNC =, GNCP =, LIMCT =, MODE =, NCP =, NTM =, PCI =, PRTSP =, STACK =, TRTCH =**

TCAM only

> **BUFIN =, BUFMAX =, BUFOFF =, BUFOUT =, BUFSIZE =, CPRI =, INTVL =, RESERVE =, THRESH =,**

13 It is preferable not to call a **DSID** by any of the Data Set Definition keywords even though strictly speaking there are no reserved words in Jol. Therefore do not use DSIDs with the names of:

> **DSID, DSN, SPACE, UNIT, VOL**

as many of the Commands or Macros use these words.

Examples

1 **DCL MACROS DS JOL.CMDLIB;**

The **DSID MACROS** describes the **JOL.CMDLIB** command instruction library. When a program **reads** the **MACROS dsid,** the system catalog will locate the volume the data set is on.

2 **DCL OUTPUT DS DISK VB 7294,200 1 TRK;**

The **DSID OUTPUT** describes a temporary disk data set with variable sized blocks and one track of space on disk.

3 **DCL FIL2 DS VOLREF = FIL1 V 100;**

The variable length record file (blocks are 100 bytes) is found, or is to be put on the same volume as FIL1. Standard, installation-defined space allocation is to be made. This is a temporary data set.

4 **DCL FIL1 DS**

DSN PAYROLL.MASTER	*Dsname*
FB 80, 800	*Record Format*
VOLUME = 111111	*Volume*
UNIT=DISK,	*Unit Type*
SPACE=4,,3 CYLS,	*Space Requirements*
IS, KEY=5,3	*Indexed Sequential*
BUFFNO=10;	*Number of Buffers*

This **ISAM** data set is found on a disk with volume name **111111**.

Space allocated for the data set was 4 cylinders. No secondary space is permitted for **ISAM** data sets so a comma is coded to indicate the space's absence. **ISAM** keys are 5 bytes long, located 3 bytes from the beginning of each record.

Records are fixed length (80 bytes) and blocks are 800 bytes long. For processing 10 buffers are to be allocated.

Declaring Printer and Punch Files

Purpose

The **PRINTER** Definition defines a spooled Printer. When data is written, moved or copied to a **Printer** definition it results in the data being **PRINT**ed or **PUNCH**ed**.**

	{ DECLARE }
	{ DEFINE } *dsid-name* **{ PRINTER | SYSOUT }**
	{ DCL }
	
	options ***;***
	
	where *options are:*

Special Jol Options	**LIKE** [=] *earlier-dsid* **EXTERNAL | EXT**
Printer Classes	*classname* **A** to **Z**, **0** to **9** or *****

Copies and Stationary	{ ***n*** **COPIES | COPIES [=]** ***n*** }
	
	{ *n* **PART** [**STATIONERY**] **}**
	{ FORM = *form-id* **}**

Record Formats	**{** *record-size,block-size* **}**
	record format **{** *block-size,record-size* **}**
	{ *block-size* **}**
	
	or **NODCB | NODEFAULT | NODEF**

3380 Parameters	**BURST | NOBURST** **CHARS** [=] *table-name [, table-name ...]* **MODIFY** [=] *module-name [, trc]*
Print Control	**FREE** [**AT**] [**END | CLOSE**] | **HOLD**
Miscellaneous	**DEST | ROUTE** = *destination* **{ FCB }** [=] *forms-control-name* [**ALIGN**] [**VERIFY**] **{ UCS }** [=] *character-set-name* [**FOLD**] [**VERIFY**] **{ PROG | PGM }** [=] *output writer program name*
Space Parameters	*n1* [-*n2*] **{ RECORDS | RECS } [OF** *modal-record-length]* or [**SPACE**] [=] *primary allocation* **[CYLS CYL]** **[** *, secondary space* **] [TRACKS TRKS TRK]** **[BLOCKS BLOCK]**

	[BLKS BLK]
Unit Parameters	**{ UNIT | UNITS }** [=] *unit-type [,no-of-units* **|** ***P*** *]* **{** { **UNITNO | UCOUNT** } **}** [=] *no-of-units* **|** ***P***

Retention Period Parameters	[**RETAIN**] **{ FOR** *n* [**DAYS**] **}**
	{ { **UNTIL | TILL | TIL** *date* } **}**

Freeing Resources	**FREE** [**AT**] [**END | CLOSE**]
Data Organizations	[**DSORG**] { **DA | IS | PS | PO | PSU | ISU | DAU | POU** }
Buffer Options	**{ BUFFNO | BUFNO }** [=] *number-of-* **{ BUFFERS | BUFFER }** *buffers* **{ BUFLENGTH | BUFLEN | BUFL }** [=] *buffer-length*

Error Options	**{ DIAGNS** [=] **TRACE }**
	{ TRACE }
	
	{ ACCEPT | ACC }
	EROPT [=] **{ SKIP | SKP }**
	{ ABEND | ABE }
	
	{ SKIP | ACCEPT [**ERRORS**] **}**

Protection Options	**{** { **READ | READONLY | READ ONLY** } **}**
	{ PROTECT }

Sundry Options	**DCBEXTRA** = *'any dcb information not currently supported'*

Notes

1. The classname can be any alphameric character (A-Z, 0-9) and can be used with VS1 or MVS. The **default class** is *****.

2. Apart from the options listed in the general form of the **PRINTER** definition, no other parameters (e.g., VOL) may be coded.

3. For special requirements not allowed by Jol, code:-

   DCL *name* **SYSOUT=**'*any special information*'

   Whatever is coded within the quotes will be copied directly to the **SYSOUT** parameter of the JCL produced.

4. If **UNIT** and/or **SPACE** is coded when using a **VS** system, they will be ignored.

5. If the Blocksize you specify is not a multiple of the Record Length for Fixed files the Blocksize is rounded down to be a multiple of the Record Length. This allows an efficient block size to be specified even though the record length is unknown.

6. **FOLD, VERIFY** or **ALIGN** are ignored if coded without **FCB** or **UCS**.

Examples

1. **DECLARE TWOPRT PRINTER 2 PART;**

   COPY SYS1.MACLIB(CALL) TO TWOPRT;

   When data is written to **TWOPRT** with the **COPY** Command, the records in member **CALL** of the **SYS1.MACLIB** data set will be printed on 2 part stationery.

2. **DECLARE MVTPRINT PRINTER Z UCOUNT 3**
   1K - 20K RECORDS OF 121;

   The **MVTPRINT PRINTER** definition specifies that the Class is Z. Sufficient space is to be allocated for 1,000 to 20,000 records of 121 characters each. If necessary, the system will search up to three units in an attempt to allocate the space required.

Declaring Programs

Purpose

Before executing a program, you must tell the system what input and output data sets it uses, how much storage it needs, the expected execution time, which library contains the program, and so on. The specified program must be a member of a temporary library, the system library, or a private library.The program declaration allows you to specify these details and when you use the **RUN** instruction later, the program is loaded and all the options specified will be activated, and any specified data sets will be made available to the program.

```
[label :]

        { DECLARE   }
        { DEFINE    }   program-name PROG  options;
        { DCL       }
```

Specifying the Filenames and Their Usage for a Program

Under **OS**, you **READ** or **WRITE** to *filenames* or **DDNAMES** in your program. These **DDNAMES** are then connected or bound to Data Sets, Printers, Card Readers and so forth at the Job Control or Command (Jol or TSO) level. This allows a great amount of flexibility because your program can be written with Device Independence in mind; your program can read or write a Disk, Tape, or Card Data Set with no changes.

Filenames and their actions are described in the following manner:

```
{  filename  }     action     { dsid    }   [ (  member    ) ]
{  ddname    }                { dsname  }   [ ( generation ) ]

                              [ ||         {  dsid   }  ]
                              [            { dsname  }  ]

                              [ { UNITAFF } ]      { filename }
                              [ { UNITREF } ]      { ddname   }

   where action may be

                        { READS                 }
                        { WRITES                }
                        { UPDATES               }
                        { MODS | EXTENDS        }
                        { USES | REWRITES       }
               or
                              { READ              }
                              { WRITE             }
                  MAY         { UPDATE            }
                              { MOD | EXTEND      }
                              { USE | UPDATE      }
```

Note

- **READS** is the same as **DISP=SHR**
- **WRITES** is the same as **DISP=NEW**
- **UPDATES** is the same as **DISP=OLD**

- **MODS** or **ADDS** is the same as **DISP=MOD**
- **USES** creates a data set if such is non-existent or uses an old data set if currently present on the system.
- **MAY** indicates that the volume is to be mounted only when and if the data set is opened by the program.

Other Optional Parameters on the Program Declare

Optional Parameters	Default
ACCT [=] *acct*	*step-acct*
EXTERNAL | EXT	-

LIB [=] **{** library-data-set-name **}**
 { dsid **}**
	-

LIKE [=] *pgmid*	-
NODYNAMS [=] *max-dynamic-allocations*	0

{ SIZE } [=] *n* **{ K }**
 { REGION } **{ BYTES }**
	step-region

{ *elapsed,cpu* **}**
 TIME [=] **{** *cpu,elapsed* **} { MINUTES | MINS | M}**
 { *elapsed* **} { SECONDS | SECS | S }**
 or
 NOTIMING
	remainder of Job Time

VIRTUAL | REAL VIRTUAL	

Notes

1. The *label* need only be coded if there are two programs to be executed of the same name using different **DSIDs** or Data Sets. The '*label*' is **RUN**, not the pgmid specified in the Declare.

2. The LIB keyword is not required if a Dsname of the form *x.y* is used. For example:

   LIB = SYS1.LINKLIB is equivalent to coding **SYS1.LINKLIB**

3. The **SIZE** or **REGION** keyword is not required if **K** or **BYTES** is used. For example:

   SIZE = 60K is equivalent to **60K**

4. The **TIME** keyword is not required if a qualifier such as **M** is used. For example:

   TIME=5,2MINS is equivalent to **5,2MINS**

   NOTIMING is equivalent to **TIME=1440** in **JCL**.

5. **LIKE** merges a previously defined Program Declare with the current declare. All non-defined items are copied from the old Declare.

6. **EXTERNAL** or **EXT** allows a Declaration in a Macro to be used outside the Macro.

7. To assist in Restart situations, Jol automatically resets generation numbers. For example, when Jol finds the first reference to a data set is (+1), it will subtract one from *all* generations of that data set until the lowest reference is (+0). Jol will perform similar processing for output data sets, and reduce the generation number until the lowest is (+1).

8. **Generation** data sets may be considered as a 'stack' of data sets. The stack may be referenced in its **entirety**, or you may refer to elements within the stack **relatively** or **absolutely**.

- The top element or data set is referred to by coding **(0)** after the data set name.

- A lower element or data set may be referred to by coding **(-*nn*)** after the data set name, for example **PAYROLL.MASTER(-2)**.

- New elements or data sets are added to the stack by coding **(+*nn*)** after the data set name. To make permanent additions to the stack, the data set **must** be **CATALOG**ed (see **CATALOG** instruction).

- **Specific** or **absolute** elements can be referred to by coding **(*nn*)** after the data set set name.

- **All** elements may be read consequectively by coding the generation data set name without specifying either a relative or absolute element number.

- You may code relative and/or specific elements on either **or both** the program **and** the data set declare statements. The element numbers you specify are **arithmetically** added or subtracted to give either a relative or absolute element number.

The table on the following page shows the results when element numbers are coded on program and data set declares.

1 **DCL GDG DATA SET PAYROL.MASTER;**

In Program Declare	Result
SYSUT1 READS GDG	all elements
SYSUT1 READS GDG(0)	latest element
SYSUT1 READS GDG(-1)	second generation
SYSUT1 READS GDG(4)	absolute generation 4.

2 **DCL GDG DATA SET PAYROL.MASTER(0);**

In Program Declare	Result
SYSUT1 READS GDG	top element, or latest
SYSUT1 READS GDG(0)	as above
SYSUT1 READS GDG(-1)	second generation
SYSUT1 READS GDG(4)	absolute generation 4.

3 **DCL GDG DATA SET PAYROL.MASTER(-1);**

In Program Declare	Result
SYSUT1 READS GDG	second generation
SYSUT1 READS GDG(0)	as above
SYSUT1 READS GDG(-1)	**third (-2)** generation
SYSUT1 READS GDG(4)	**error,** but absolute generation 4 used.

4 **DCL GDG DATA SET PAYROL.MASTER(6);**

In Program Declare	Result
SYSUT1 READS GDG	generation **6**
SYSUT1 READS GDG(0)	as above
SYSUT1 READS GDG(-1)	generation **5**
SYSUT1 READS GDG(4)	**error,** but absolute generation 4 used.

In other words, Jol performs arithmetic on the two numbers, thus easing coding for restarts.

Examples of Program Declarations

1 **DECLARE IEFBR14 PROG 8K;**

IEFBR14 is declared as a program. Only one option is specified **-** it is to have a maximum storage of 8,000 bytes. A **RUN IEFBR14** instruction is necessary before the program is actually executed.

2
```
DECLARE PRINT PROG 60K
          INPUT      READS   TEST.DATA (0)
                                || TEST.DATA (-1)
          PRINTER    WRITES     PRINTER;
```

The program **PRINT** uses the internal file **INPUT** to read data. In this case, **INPUT** will read two concatenated data sets, **TEST.DATA(0)** and **TEST.DATA(-1)**. Jol will search the catalog and resolve the generations of zero and minus one before allowing the job to start executing.

3

```
DCL     VALIDATE PROG
        LIB       SALES.LIB
        SIZE      250K
        TRANSIN        READS          CARDS
        TRANSOUT       WRITES         EDITEDT
        LIST           WRITES         PRINTER;

        other instructions

        RUN VALIDATE;
```

Program **VALIDATE** is declared. The LIB parameter specifies that it is in the **SALES.LIB** library, and the **SIZE** parameter indicates that a maximum of 250K bytes are to be allocated to it.

File **TRANSIN** is to **READ** the **CARDS DSID**, file **TRANSOUT** will write the **EDITEDT DSID** and the **LIST** file will write to the **PRINTER DSID**.

Sometime later, the **RUN VALIDATE** instruction will cause the program to start executing. At this time all the required data sets are allocated for it.

4

```
DCL     IEBGENER       PROG        60 K  SYS1.LINKLIB
        SYSUT1         READS       'INPUT.MASTER(0)'
                                   || 'TRANS.ACTION.FILE'
                                   UNITAFF SYSUT1
        SYSUT2         WRITES      OUTPUT
        SYSPRINT       WRITES      PRINTER
        SYSIN          READS       'SYS1.CONTROL(DUMMY)';
```

Program **IEBGENER** is defined to Jol. When it is executed, it will be loaded from library **SYS1.LINKLIB,** and allocated **60K** bytes of storage to execute in.

File **SYSUT1** will read **'INPUT.MASTER(0)'** first, and then **'TRANS.ACTION.FILE'**. These two files will be copied to the Data set Identifier (**DSID**) **OUTPUT**, which will be defined before the **RUN IEBGENER** instruction.Notice that **'TRANS.ACTION.FILE'** has a **UNITAFF** specified to **SYSUT1**. This means that the same unit (usually a Tape device) will be used for both files.

SYSPRINT and **SYSIN** are files also required by the utility; **SYSPRINT** will write to a printer, and **SYSIN** will read data set **'SYS1.CONTROL'** member **DUMMY**.

5

```
DCL     XYZ PROG       PROG
        IN01      READS          B(+1) || C(10)
        IN02      READS          D UNITAFF A;
```

When program **XYZ** is run, file **IN01** will read the **(+1)** generation of the data set defined by the **DSID B**, then it will read the tenth **(10)** generation of the data set defined by **DSID C**. When the end of the file used by **IN01** is reached, file **IN02** is opened and the data referenced by **D** is used. Because **UNITAFF** was specified, the data set referred to by **D** will be mounted on the same unit as was used by file **IN01**.

DELETE - *Execute Instruction*

Purpose

The **DELETE** instruction to Scratches and Uncatalogs data sets. The data contained in the data set is lost, and any reference to the data set is removed from the system catalog.

```
DELETE { dsid | dsname } [ (generation-number) ] . . .
            [ FROM [VOL] volume [UNIT] unit ]
                    [ALWAYS] ;
```

Notes

1. All **OLD** data sets are automatically kept unless a **DELETE** or **SCRATCH** instruction is executed, but all **NEW** data sets are scratched by the Operating System unless they are kept or cataloged.

2. Jol will obtain Volume, Unit and Position information from the Data Set Declaration unless a **SCRATCH** volume is associated with a new data set, in which case the information will be found at execution time from the Operating System.

3. The '*dsname*' form of the instruction is not recommended for long production jobs as the **DSID** format allows the Data Set Name to be overridden; thus a data set name may be altered in one place **-** the **DECLARE -** and all references to the **DSID** will have the new Data Set Name.

4. The **ON** or **FROM** options, if specified, must be coded after the **DSID** or **DSNAME** list.

5. **ALWAYS**, when used with a **SCRATCH** or **DELETE** instruction, will scratch or delete a Data Set even if the Expiration Date has not expired.

Examples

1. **DELETE DSID1;**

2.
```
DELETE      'TEST.DATA.SET1'
            'TEST.DATA.SET2'
                    FROM VOL1 2314;
```

DISPLAY - *Execute Instruction*

Purpose

The **DISPLAY** Instruction prints a message on the job's system log.

The message is not typed on the operator's console. If a message is to be typed on the operators console, the **TYPE**, **SIGNAL ERROR** or **STOP** instruction should be used.

DISPLAY { *number* | ‘message’**}** ;

The **DISPLAY** Instruction is most often used to inform those examining the output from the job about certain situations or conditions. It can, for example, be used to indicate when a restart point has been passed, and if a restart is required for any reason **-** how it can be done.

Note: The message must be less than 100 characters.

DO *Instruction (Immediate and Execute Time)*

Purpose

DO and **END** Statements are used to construct '**DO blocks**. A **DO** block begins with a **DO** Statement and ends with a matching **END** Statement.

```
DO ;
    jol statement...
    jol statement...
END ;
```

A **DO** block is usually used after an **IF** statement and signifies the beginning of a group of instructions which may then be executed or bypassed subject to the result of an **IF** Instruction.

Notes

1. The maximum number of levels of **DOs** allowed is *eight* (8).
2. A **DO** group is terminated by an **END** statement.
3. A **DO** group may be repeated with the **GOTO** or **REDO** instructions.

Example

1

```
IF PROG1 = 0 THEN
DO;
        COPY INPUT TO OUTPUT;
        CATALOG OUTPUT;
END;
```

DUMPVOL - *Execute Command*

Purpose

The **DUMPVOL** Command copies the entire contents of a disk volume to tape.

The volume may then be restored using the **LOADVOL** Command.

DUMPVOL *volume1 [,volume2. . .] options* ;

options			**default**
BACKDSN	=	*high level of tape data set name name*	**JOL.BACK.UP.OF** *volume(+1)*
FROMUNIT	=	*unit-type of disk*	**DISK**
TOUNIT	=	*unit-type of tape*	**TAPE**

Notes

1. *Volume1, volume2* etc. specifies up to 6 volumes to be copied to tape.
2. **BACKDSN** specifies the first part of the data set on the tape containing the dumped version of the data on the disk volume. It defaults to **JOL.BACKUP.OF** volume (+1), and so if **DUMPVOL 111111** were coded, the data set name would become **JOL.BACKUP.OF111111(+1)**.
3. When more than one volume is specified in the command, the disk volumes specified are copied to tape concurrently and as many tape drives as are necessary are allocated to perform the function concurrently.
4. If fewer tape drives than volumes to be dumped are to be used, code separate **DUMPVOL** commands.
5. The Volume dumped can be restored with the **LOADVOL** command.

Examples

1 **DUMPVOL DISK01;**

The **DUMPVOL** command will copy the entire contents of disk volume **DISK01** to a tape called **JOL.BACK.OFDISK01 (+1)** and catalog the tape. The volume **DISK01** must be accessible using a generic unit name of **DISK,** or else the extra parameter **FROMUNIT** must be coded.

2 **DUMPVOL 111111, 222222 FROMUNIT 3350;**

The **DUMPVOL** command will copy the contents of disk volumes **111111** and **222222** to tape concurrently. They are **3350** type devices, and will be copied to tape data sets of:

JOL.BACKUP.OF111111 and
JOL.BACKUP.OF222222 respectively.

EDIT - *Immediate Instruction*

Purpose

The **EDIT** Instruction splits information contained in one symbolic variable according to specifications in a format list. The results are placed in other specified symbolic variables. Multiple **SUBSTR** functions may be used to perform the same function.

```
EDIT        input-symbolic-name
            [ ( ] output-symbolic-name-list [ ) ]

            [ ( ]     A(n)   [,...] [ ) ]   ;
                      X(n)
```

The **EDIT** instruction is interpreted by the Jol pre-processor. This means that only symbolic variables can be altered. You cannot **EDIT** symbolics at Execution Time.

Notes

1. Names in the **EDIT** should **not** have the **%** symbol preceeding them. Jol will replace any names coded with **%** with the current contents of the variables. For example:

   %VARIABLE = 'A,B,C A(2)';

   EDIT SYMNAME %VARIABLE;

   After the EDIT, symbolic names A,B,C will contain the edited information.
   Format item **A** specifies that ***n*** characters from the input string are to be copied into the current output location.

   Format item **X** specifies that ***n*** characters are to be skipped on the input string.

1. If there are more names in the output list than format items, the format items are reused until either the input string or the output name list is exhausted.
2. If the output variable names are not declared, they will be implicitly declared.

Examples

In all cases assume that **%INPUT** contains **'123456789'**.

1 **EDIT INPUT (A,B,C) (A(3));**

After the **EDIT A** will contain '**123**', **B** will contain '**456**' and **C** will contain '**789**'.

2 **EDIT INPUT A,B,C X(2),A(2);**

After the **EDIT A** will contain '**34**', **B** will contain '**78**' and **C** will contain ''(null).

END - *Instruction*

Purpose

The **END** Instruction terminates a **DO** group or Macro Command. See the **DO** and **MACRO** instructions for further details.

```
END;
```

Ending a DO Block or Macro Command

Any block of code commenced with a DO or **MACRO** instruction must be terminated with an **END** instruction so that Jol will know where the Macro or **DO block** terminates. **DO blocks** and **MACRO** instructions are fully discussed in their appropriate sections.

Examples

1

```
IF RESTART = 1
THEN DO;
        DCL X DS GEN.DATA(0);
        DISPLAY 'STARTING AT STEP10';
        STARTAT STEP10;
END;
```

2

```
IF STEP1 = 0
THEN DO;
        RUN STEP3;
        RUN STEP4;
END;
```

ENQ/DEQ - *Immediate Instructions*

Purpose

The **ENQ** and **DEQ** instructions enque and dequeue system resources. These instruction should be used with caution, otherwise system interlocks could occur.

```
        ENQ qname/'minor-name' [ WAIT | NOWAIT ] ;
and
        DEQ qname/'minor-name' ;
```

Enquing on Resources

You can use the **ENQ** instruction to stop another user updating a data set that you have allocated as **SHR** by enquing on it when you need to update it. When you have updated the data set, you can then free it so that other users can update it.

As long as everyone using the data set uses the same enque names, you can gain control of the data set for the short time you require it, and then allow others to have access to it. The Jol **SAVESYMS** command uses this technique to control access to data sets.

Notes

1. When Jol is running in a background region, **WAIT** is the default. Your job will wait until the resource is available, and the Operator will be notified that your job is waiting.
2. When Jol is running under TSO, **NOWAIT** is the default. You will be notified if the resource is not available, and can request that Jol trys again, or ignores the request.
3. **%LASTCC** is set to 0 if the request was successful, otherwise a non-zero result is set._You can use the **IF** statement to determine if you gained access to the resource.
4. You *must* **DEQ** the resource after you have finished with it. Jol will not automatically **DEQ** resources you have enqued.

Example

1. **ENQ SPFDSN/'%SYSUID.SAVESYMS';**

   The data set **%SYSUID.SAVESYMS** will be made available to youif it is not already enqued under that name. This could occur if another Jol user enqued on that data set, or if SPF was saving data into that data set when you requested control.

EXEC - *Execute Command*

Purpose

The **EXEC** Command is used to **Execute** a previously registered problem program, optionally supplying a list of data sets, and a parameter for the program.

```
     EXEC        program
                         [list of dsnames or dsids]
                         [options] ;

 where options are:

        PARM   -         Parameter for the Executing Program
        LOAD   -         Over-riding LOAD Module library
```

Notes

1. The program must have been previously **REGISTER**ed.
2. The list of data set names or data set identifiers must be in the same sequence used when the program was registered.

Example

```
1  DCL OUTPUT       DS OUTPUT.MASTER
                         VB 70,7000 SYSDA 10 CYLS;
   DCL TRANSIN      DS TRANS.INPUT;
   DCL PRT          PRINTER B FORM 100 FBA 121,1210;

   EXEC MASTUPDT
             'INPUT.MASTER',
             INPUT,
             TRANSIN,
             OUTPUT,
             PRT;
```

The **DSID**s and ***data sets*** will be bound to the ***filenames***_or **DDNAME**s in the order that they were specified when the program was registered.

EXIT - *Immediate Instruction*

Purpose

The **EXIT** Instruction either:

- terminates a **MACRO**

or

- terminates the Jol compile.

```
EXIT    [optional return code] | 'message';
or
EXIT QUIT ;
or
EXIT JOB ;
```

When coded in a macro, processing is returned following the macro invocation statement. **%LASTCC** is set to the return code, or **0** if not specified.

When coded in the main-line program, or the **QUIT** option is specified, the compiler is terminated and a return code of 24 returned to the operating system.

EXIT JOB terminates the *current* Jol compile immediately, but the compiler continues to read data from the terminal or data set, searching for other jobs to compile.

Terminating the Jol Compiler

When the EXIT QUIT instruction is executed Jol exits and produces *no* generated Jol. The **EXIT** instruction allows the Jol programmer to check for programmed conditions and cancel the compilation.

For example, checks may be made for program loops in the preprocessor phase by the Jol programmer setting up a count and **EXIT**ing when his criteria for exiting has been met.

EXTEND - *Execute Command*

Purpose

The **EXTEND** Command is used to add data to an existing data set._The data set is lengthened.

```
EXTEND  { dsid | dsname }
        WITH   { dsid | dsname }
        [ || ...] ;
```

Extending Data Sets

It is sometimes desirable to add data to the end of an already existing data set. One reason would be to maintain one data set with many days transactions; if necessary, the data set could be interrogated or split into separated transactions files for backup recovery purposes.

Notes

1. The **EXTEND** command uses the system copy utility to copy the data on to the end of the specified data set.
2. If the data set being **EXTENDed** does not exist, it will be created. If it is created it must be **KEPT** or **CATALOG**ed, or it will be **DELETE**d when it is used last in the job, or at the end of a job.
3. If a specific volume and unit are specified (that is, a non-cataloged data set is to be **EXTEND**ed), the data set must exist on the specified volume. If it does not exist, an abend will occur. If it is known that the data set does not exist, use the **COPY** command to allocate it. The **TEXIST** command can be used to see of a data set exists.

   For example:

```
DCL TRANS DS        SALES.TRANS.BACKUP
                    VOL DA01 3330
                    10 CYLS;

DCL CARDS*;
      cards
EOF;

TEST: TEXIST TRANS;
IF TEST = 0  /* IF YES, TRANS DOES EXIST */
      THEN EXTEND TRANS WITH CARDS;

ELSE DO;            /* TRANS DOES NOT EXIST */
      COPY CARDS TO TRANS;
      KEEP CARDS;
END;
```

The **TEXIST** checks that the data set is on the volume. If it is, it will return a zero, and the **EXTEND** command will append card data to the data set.

If the **TEXIST** returns a non-zero value, the data set is not on the volume, and the **COPY** and **KEEP** instructions will be executed.

4. The **EXTEND** command extends sequential data sets only.
5. If the data set is cataloged, you should Catalog the data set again with the **ALWAYS** option because the data set may have been lengthened and extended on to another volume

Example

1

```
DCL OLDTRANS DS SALES.BACKUP.TRANS;

DCL NEWTRANS *;
        cards
EOF;

EXTEND OLDTRANS WITH NEWTRANS;
```

OLDTRANS defines a data set called **SALES.BACKUP.TRANS** and **NEWTRANS** defines a card file with new transactions. The **EXTEND** command will append the card data to the data already in the **SALES.BACKUP.TRANS** data set.

FORT - *Execute Command*

Purpose

The **FORT** Command calls the **FORTRAN** Compiler to convert **FORTRAN** Source code to object code. The object code may be input to the **LINK** or **LOADGO** Command and executed.

```
FORT { data-set-name | dsid } [ options ] ;

where options are:

    { PRINT      { data-set-name | dsid } | NOPRINT    }
    { SYSOUT     * | class                             }

    OBJ    { dsid-list | dsname-list } | NOOBJ
    ALC | NOALC
    DEBUG | NODEBUG
    DOUBLE | NODOUBLE
    DOVAL (0 | 1 | 2 | 3)
    EBCDIC (BCD)
    FLAG(I | E | W)
    GOSTMT | NOGOSTMT
    INLOG2 | NOINLOG2
    LET | NOLET
    LINECNT (line-count)
    LIL | NOLIL
    LIST | NOLIST
    LMSG | SMSG
    MAP | NOMAP
    NAME (name) | NONAME
    OPTIMIZE ( 0 | 1 | 2 ) | NOOPT
    PANEL | NOPANEL
    RENT | NORENT
    SEQ | NOSEQ
    STATIS | NOSTATIS
    SOURCE | NOSOURCE
    TERM | NOTERM
    XREF | NOXREF
```

Notes

1. With a Time Sharing System **SYSOUT** defaults to **%TSOCLASS,** so that the output can be viewed at the terminal. Otherwise, **SYSOUT** defaults to the installation defined **SYSOUT** class.

2. **OBJ** defaults to a temporary data set **&OBJ**. It may be referenced by the DSID **OBJ** for the **LINK** Command.

3. See also the **COMPILE** Command.

Example

1
```
FORT SOURCE (CALCULAT);
LINK OBJ LOAD TEST.LOAD (CALCULAT) FORT;
RUN CALCULAT;
```

FREE - *Execute and Immediate Command*

Purpose

Until a job ends, new data sets cannot normally be used by another job. **FREE** frees data sets so that other jobs may access them immediately. **FREE** can also free files or datasets that were allocated under TSO with the Jol or TSO **ALLOCATE** instruction.

(See also the FREE parameter in the Data Set Declare statement)

```
        FREE   {  dsid | dsname } [ (generation-number) ] . . .

    or

        FREE          F | FILE(file-name)

    or

        FREE          DA(data-set-name)
```

Notes

1. You may **FREE** up to ten (10) data sets in one **FREE** command; to free more than ten, use a second **FREE** command.

2. If the data set name you wish to **FREE** is a single length data set name, you must enclose it in apostrophes, otherwise Jol attempts to find a data set identfier (**DSID**) to free. For example, to free data set **X**, code:

```
FREE 'X';
```

Example

1

```
DCL DSIDONE DS SYS1.MACLIB;

FREE NEW.MASTER(+1),
          DSIDONE;
```

The generation data set **NEW.MASTER(+1)** must have been created by an earlier step in the job. The **FREE** command allows other Users to use the data set immediately - without waiting for the job to end. The data set referred to by **DSIDONE** is also freed, that is, **SYSI.MACLIB**.

FS ON | OFF - *Immediate Command*

Purpose

The **FS ON | OFF** Command turns the full screen panels on or off.

```
FS   ON | OFF  ;
```

Notes

1. Many Jol instructions use the **PANEL** instruction to prompt the User for information and to receive data. **FS OFF** allows experienced Users to perform their work faster without prompts.

2. **FS OFF** turns **PANEL**s off and the defaults in the panels will be used by Jol. **FS ON** turns panels and other terminal related instructions on.

3. See the **PANEL** instruction for details of displaying full screen data entry panels.

Example

1. **FS OFF;**

   FS OFF temporarily disables the **PANEL** instruction.

GET - *Immediate Instruction*

Purpose

The **GET** Instruction reads data into symbolic variables at compile time (*not at execution time*). Optionally, you can search the data set for records with specific keys in preset positions.

```
            GET        filename [(member-name)] options ;

    options

            KEYS = key-length, key-position, key-name-list
```

Notes

1. The file must have been previously allocated with the **Jol ALLOCATE** instruction, the**TSO ALLOCATE** instruction or with the **JCL** used to invoke Jol.
2. Any records longer than 253 characters are truncated.
3. If no options are specified, the entire file is read into Symbolic Variables called **CARD(1)** to **CARD(*nnnn*)**.
4. Only Fixed Blocked or Variable Blocked files are supported.
5. **ISAM** and **VSAM** files are not supported.
6. When the **KEY** option is specified, the file is searched for the specified keys; the keys are the names specified in the name-list, and the symbolic variables are set equal to the contents of the record if the key is found in the file.
7. Any names specified in a name list will be implicitly defined if they are not already declared, and set to null if the corresponding data is not found on the input file.
8. The **READ** or **GETFILE** instructions are more flexible, and can be used in place of the **GET** instruction.

Examples

For the following examples, assume that the file referenced by **DDNAME INPUT** contains the following records:

a)	**JANbbb01**
b)	**FEBbbb02**

1. **GET INPUT;**

Result **%CARD(1)** will contain '**JANbbb01**'
%CARD(2) will contain '**FEBbbb02**'

2. **GET INPUT KEYS 5,1 JAN,FEB,MAR;**

Result **%JAN** will contain '**JANbbb01**'
%FEB will contain '**FEBbbb02**'
%MAR will contain '**(null)**

3. **GET INPUT LIST JAN,FEB,MAR;**

Result	**%JAN**	will contain **01**
	%FEB	will contain **02**
	%MAR	will contain **(null)**

GETFILE - *Immediate Instruction*

Purpose

GETFILE reads a record from a file into a symbolic variable with the same name as the filename. *See also the **READ** instruction.*

```
GETFILE  filename ;
```

Notes

1. The file must have been **OPEN**ed with the Jol **OPENFILE** instruction.

2. Before reading a record the **IF** instruction may be used to determine if the file is at the end of the file, by coding:

   IF EOF(*filename*);

3. **%LASTCC** is set to zero if a record was read successfully, otherwise it is set non-zero.

4. You can use the **GOTO** instruction to read the entire contents of the file. Thus files may be copied with a combination of **READ** or **GETFILE**, **WRITE** or **PUTFILE** and **REDO** instructions, with processing on the contents of the file being performed.

5. The file must only contain character information, and the maximum record length that may be read is 253 characters.

6. If the file contains Variable length records, the symbolic variable length is set equal to the length of the record read. For Fixed length record files, the size of the symbolic variable is set equal to the length specified in the file.

7. See also the **ALLOC, OPENFILE, READ, PUTFILE, WRITE, CLOSE** and **FREE** instructions.

Examples

1.
```
OPENFILE INPUT2 INPUT;       /* Open the file */
IF ^ EOF(INPUT2) THEN
DO;
    GETFILE INPUT2;
    ... perform processing required.
END;
```

GETTIME - *Immediate Instruction*

Purpose

GETTIME resets the day, date and time Symbolic Variables. This can be useful if Jol has been in a long wait, and you wish to find the current date and time.

```
GETTIME  ;
```

Automatically Initialized Symbolic Variables

As described in *Declaring Symbolic Variables* above, Jol initializes certain symbolic variables at startup time. They are initialized once and can then be used with the Jol logic statements to control events and functions. These variables are shown below:

Variable	Explanation
%SYSDATE	The current date in Julian format e.g. **87020**
%DAY	**MONDAY**, **TUESDAY**, etc.
%MONTH	**JANUARY**, **FEBRUARY**, etc.
%MONTHNO	**01, 02** Through **12**
%DAYNO	**01** through **31**
%YEAR	**1987, 1988**, etc.
%HOURS	**0** through **23**
%MINS	**0** through **59**
%SECS	**0** through **59**
%SYSUID	System user identification
%SYSPREF	Dataset prefix or Current Directory
%SYSPFK	Program function key number from **PANEL**s
%SYSTEM	**MVS**, **DOS**, **VM**, **PC, UNIX** etc.
%SPOOL	**HASP, ASP**, **JES1**, **JES2**, **JES3**, or Blank
%TSOCLASS	Contains the **SYSOUT** class used by TSO to retrieve output from a background job.

The **GETTIME** instruction resets **%SYSDATE, %MONTH, %MONTHNO, %DAY, %DAYNO,** and **%TIME** to the current date and time.

Notes

1. The Jol compiler usually starts converting a job to an executable program, and stops after it has performed this function. **WAIT**s can be introduced by calling a program that waits for a specific event to occur, or by using such instructions as **PANEL** or **READ**.

Example

1. **GETTIME ;**

   GETTIME resets the Symbolic Variables as described above.

GOTO - *Immediate Instruction*

Purpose

The **GOTO** instruction provides a looping facility.

```
label:
          any JOL code;

          GOTO label  ;

where
          label       is a 1 to 8 character label name.
```

The **GOTO** statement allows you to **GOTO** a set of instructions **within a macro** or **an included member**; it allows you to repeat Jol code in the preprocessor phase.

You can **GOTO** from any nested level of a **DO** group to a lower level, for example:

```
A:        IF expression
          THEN DO;
               Jol statements
               . . .;
               IF expression
               THEN DO;
                    Jol statements
                    . . .;
                    IF . . .
                    THEN DO;
                         GOTO A;
                    END;
               END;
          END;
```

With this example the **GOTO** instruction is coded at the third level and refers back to a first level Jol statement.

Notes

1 The **GOTO** instruction cannot be used for execution type statements, for example:

```
STEP01: RUN PROG1;
IF PROG1 > 0 (e.g. did not not run)
THEN GOTO STEP01 ;
```

is **incorrect** and will not be allowed in Jol. Howver, because a **CALL** instruction is executed immediately, you can call a program any number of times.

2 If the label has not yet been encountered in your program, all statements will be bypassed until the statement is found.

Example

Suppose you require a tape number to be entered by the userof your Jol program.

You decide that the tape number must be from 1 to 25, and wish to validate that the User has entered a number within that range. The following code will be repeated until the correct value is entered.

```
PANEL ('TAPE NUMERICS MUST RANGE FROM 01-25')
                                                    ///  /* Space Screen Down */

        ('ENTER TAPE NUMBER =====>',ANS1,2,'10') ;

VALIDATE:
        TYPEANS1 = TYPE(ANS1);

        IF TYPEANS1 ^= 'NUM'
        |  ANS1 > 25
        |  ANS1 < 10
        THEN DO;
            PANEL REREAD FROM ANS1  /* Place Cursor at ANS1 */
                'TAPE NUMBERS MUST BE 01 to 25';
                    /* Error Message is at Top of Screen */
            GOTO VALIDATE;
        END;
```

If the terminal Operator does not type in a number then the **PANEL** is redisplayed again and the error message appears on the screen. If the Operator types in a number that is too small or too large then the **PANEL** is also redisplayed.

The **REREAD FROM** clause places the cursor at the beginning of the field in error as a convenience for the Operator. The message that follows is displayed on the top of the terminal.

HELP - *Immediate Command*

Purpose

Jol provides a **HELP** facility to give assistance when using Jol at a terminal.

```
HELP;
```

If you need assistance type and press the enter key on your terminal.

Jol will then ask you a series of questions and provide you assistance.

Notes

1. This facility is primarily for the use of programmers and assistance with such things as the Panel Command, IF Statement, declaration of data sets etc. It is not designed as a facility for users who have incorrectly entered data, or similar errors.

2. Many Jol commands will provide assistance when Function Key 1 is pressed, and it is good practice to write your own Commands and Panels so that they too provide assistance when Function Key 1 is pressed.

IF - *Immediate and Execute Instruction*

Purpose

The **IF Statement** can:

- Compare the values of return codes from executed programs with either constants or other program return codes.
- Compare the current value of symbolic variables with constants or other symbolic variables.
- Test if a program has executed or if a symbolic is declared.
- Test if **E**nd-**O**f-**F**ile (**EOF**) has been reached on an input file opened by **OPENFILE**
- Test if an **ERROR** has occurred.
- Test the **LAST** and/or **MAXIMUM** completion code.

```
           { ANY                 }
           { constant            }
           { ERROR               }   [            { symbolic-variable } ]
IF         { label-of-a-run      }   [ operator   { program-name        } ]
           { LASTCC              }   [            { label-of-a-run      } ]
           { MAXCC               }   [            { constant            } ]
           { program-name        }
           { symbolic-variable   }
           { EOF(file-name)      }

                THEN unit - 1 ;
                [ELSE unit - 2 ;]
```

The fields between the **IF** and the **THEN** are known as the '*expression*'. The comparison operator may be one of the following:

Operator	**Meaning**
=	equal to
⌉=	not equal to
^=	not equal to
<	less than
<=	less than or equal to
⌉<	not less than
^<	not less than
>	greater than
>=	greater than or equal to
⌉>	not greater than
^>	not greater than

Depending on the result of the comparison or test, one or more of the statements can be either executed or bypassed.

The instructions in '*unit-1*' and '*unit-2*' may be:

- Any Jol instruction
- Any **MACRO** instruction

- Any group of Jol instructions enclosed within a **DO/END** pair.

Meaning of the operands:

ANY	means if any program so far executed (via **RUNs** or as part of a Jol command) meets the criteria, then the result is true. For example: **IF ANY=8 THEN . . .**
ERROR	means if an **Abend** (either a System or User Abend) has occurred, then the result is true. For example: **IF ERROR THEN . . .**

constant	is any numeric or character constant. If a program return code is being compared with a constant, the constant must be a number in the range 0-4095. For example:
	
	IF %A ='CHARACTER STRING' THEN
	
	IF %X='STRING' THEN . . .
	
	IF SORT > 16 THEN . . .

label-of-a-run	is the *label* coded on **RUN** statements or Commands. For example:
	
	A: RUN VALIDATE;
	
	SORTRUN:
	SORT INPUT TO OUTPUT USING PARM;
	
	IF A=20 & SORTRUN<8 THEN . . .

LASTCC	is the return code from the last executed program, whether the program was part of a Jol command or a problem program executed with the **RUN** instruction. The number will be in the range 0-4095. For example: **IF LASTCC > 24 THEN . . .**
MAXCC	is the highest return code issued up to this point in the Jol procedure, whether from programs executed by the **RUN** instruction or as part of a Jol command. For example: **IF MAXCC=0 THEN . . .**

symbolic variable	is any *symbolic variable* name preceded by a **%** symbol. For example:
	
	IF%SYMBOLIC < 10 | %SYMBOLIC > 25 THEN . . .
	
	IF %RESTART THEN . . .
	
	IF %RESTART='STEP10' THEN . . .

Notes

1 Symbolic variable testing and return code testing may be combined into one **IF** statement.

- If the entire **IF** statement is found to be false, the next statement or group of statements is skipped.
- If the **IF** statement is found to be true, the next statement(s) are passed to the main compiler phase, unless it, or they in the case of a group of instructions, are found to be statements for the pre-processor; for instance, a symbolic variable declaration or assignment.
- If the **IF** statement was found to contain references to program return codes, the **IF** statement is simplified, re-constructed and passed to the main compiler phase, along with the next instruction or group of instructions.

2 Two or more expressions may be **AND**ed using the operator **&**, and two or more expressions may be **OR**ed using the operator **|**.

3 Expressions may be enclosed in parentheses to give precedence in the evaluation.

4 Testing return codes issued by steps within a macro command is achieved by placing a '**label**' on the macro statement and testing this label in the **IF** statement. The return code issued by the macro command is the highest return code issued by any step within the macro.

5 When **symbolic variables** are tested, the following procedure takes place:-

a) Quotes are placed around any symbolic variable names. This is so values containing special characters or blanks will not upset the comparison.

b) The symbolic variable names are replaced by the contents of the variables.

c) Any numbers are converted to numeric form and compared arithmetically. Numbers longer than 13 digits or any character strings are compared character by character. All character strings (including null strings) are padded with blanks to a length of 253 characters before any comparisons take place.

To differentiate between **null** and **blank** you must use the **LENGTH** function of the assignment statement and then test that the length of the string is zero. Null strings have a length of zero, blank strings do not.

6 When testing symbolic variables it is not necessary to code quotes around them; Jol will automatically do this. However, this means that the following statements will be evaluated at preprocessor time rather than execute time.

```
        %PROG ='IEFBR14';
        RUN %PROG;
        IF %PROG=0 THEN . . .
```

To have the **IF** evaluated at execute time, the following must be done:

```
        %PROG='IEFBR14';
STEPNAME:
        RUN %PROG;
        IF STEPNAME . . .
```

7 The maximum number of tests that may be performed in one **IF** statement based on the normal Jol statement limits of 512 tokens, and a maximum statement length of 2048 characters. Thus approximately 512/4 (i.e. 128) comparisons may be made as each name, comparison operator, comparand and connector count one token, for example:

IF A=1 | B=2 is **8** tokens.

If the test you are making is too complex to code in one **IF** statement, code multiple **IF**s, for example,

```
IF . . .
THEN IF . . .
THEN . . . .
```

8 The **STOP WHEN** instruction may be used to apply continuous (and overriding) tests to all program return codes.

9 If steps are bypassed with a **STARTAT** instruction, any test referring to them will be marked False.

10 After an Abend the **ERROR** may be used to execute another instruction, or group of instructions. Should that instruction also Abend another **IF ERROR** may be used to recover from the second level error. This process may be repeated as many as eight times. Some examples:

a)
```
A:    RUN IEBGENER;
      IF ERROR THEN . . .
```

b)
```
A:    RUN IEBGENER;
B:    RUN IEGBENER;

    IF A & ERROR THEN . . . /* DID STEP A ABEND */
    IF B & ERROR THEN . . . /* OR STEP B? */
```

c)
```
IF A=10 & (B=49 | B=32) & ERROR THEN . . .
```

Note that the **ERROR** condition will give unpredictable results if it is coded within a parenthesized expression as it is given precedence over any other condition test within an **IF** statement.

11 Because an **INCLUDE** instruction is effectively replaced by whatever instructions are in the included member, it must be placed in a **DO** group if an **ELSE** clause is used. For example:

```
IF %X
THEN DO;
     INCLUDE MEMB1;
END;
ELSE . . .
```

Examples

1 **IF %A THEN...**	'***true***' if the symbolic variable **A** is declared.
2 **IF %A=10 THEN...**	'***true***' if the symbolic variable **A** is declared and equal to **10.**!a

3 **IF PROGRAM1 THEN...**	**'*true*'** if **PROGRAM1** executed.
4 **IF PROGRAM1=50 THEN...**	**'*true*'** if **PROGRAM1** executed and returned a value of **50**.
5 **IF %A=10 & PROGRAM1=50 THEN...**	**'*true*'** if the symbolic variable **A** was declared and currently held a value of **10** and **PROGRAM1** executed and returned a value of **50**. This is the equivalent to coding **IF %A=10 THEN IF PROGRAM1=50**.

6

```
IF (%VAR1='RUNNO'
    & (%VAR2='RERUN'   | %VAR2=''))
    & STEP3=0 THEN

STEP4:  RUN STEP4;
```

'*true*' if the symbolic variable **%VAR1** has the value **RUNNO** *and* the symbolic variable **%VAR2** has either a **'null'** value (no value) *or* the value **RERUN** *and at execution time* the return code issued by **STEP3** is zero.

If true, run **STEP4**.

If the value of **%VAR1** was say, **DATE**, then the **ANDed** test would have failed at compile time and **STEP4** would not even be considered for running at execution time.

If all the **'AND'** operators had been **'OR'** operators, then **STEP3** would be checked at execution time to determine whether or not **STEP4** is to run.

7

```
IF (PROGRAM1 = 10 | PROGRAM1 = 20)
& (PROGRAM5 < PROGRAM6)
| PROGRAM3
THEN STOP 'INVALID RETURN CODES FOUND, JOB ABORTED';
```

It means true:

	if **PROGRAM1** executed and returned a return code of **10** or **20**
and	the return code of **PROGRAMS** is less than the return code of **PROGRAM6** and they both executed
or	**PROGRAM3** executed at all.

If the **IF** is true, the **STOP** instruction will execute thus terminating the job. The **STOP** will notify the operator that an error was detected in one or more of the return codes.

8

```
IF %DAY = 'FRIDAY'
& (%DAYNO>8 & %DAYNO<5)
& %MONTH = 'JUNE'
THEN DO;
        any instructions, eg RUN, SUBMIT, INCLUDE;
END;
```

If the current day is Friday, and the date is between the eighth and the fifteenth of June (that is, the second Friday in June), then **DO** the next group of instructions. Using the **IF** in this manner gives one the ability to schedule jobs on a calendar basis.

INCLUDE - *Immediate Instruction*

Purpose

The **INCLUDE** instruction copies Jol *source text* from the member specified in the Jol Include Library. The source text is compiled as though it was part of the original input to the Compiler.

	INCLUDE	*member-name* ;
or		
	INCLUDE	'*dsname*'

Including Instructions or Source Text

Typically, Jol code is stored in a library called the **INCLUDE** library. The **INCLUDE** instruction can instruct Jol to read the statements from the library.

Notes

1. The copied or **INCLUDEd** text is inserted immediately after the **INCLUDE** instruction. Included Source Text may contain any Jol Macro Instructions or *card image files*.

2. The **INCLUDEd** text may **INCLUDE** other members, up to a nesting level of twelve. That is, an included member may include another member, and that member may include another member, and so on up to a level of twelve (12) concurrent **INCLUDES**.

3. Card Image Data may be stored in text members for inclusion; these cards (perhaps utility control statements) may have variable data changed by using the symbolic parameter facility.

4. The member name to be included may be defined by a symbolic parameter. For example:

   INCLUDE %DAY;

   will **INCLUDE** member **MONDAY** or **TUESDAY**, etc.

5. Member names must be eight characters or less, and must commence with an alphabetic or national (**$**, **@** or **#**) character.

6. Your installation may have several libraries concatenated to each other. When you **INCLUDE** a member, each library is searched in the order in which they are concatenated. This means that if two or more libraries contain identically named members, the first member found by the specified name will be the one to be included.

7. After the instructions in the included member have been analyzed, the instructions following the **INCLUDE** are analyzed.

8. Users with **TSO** can use the **TSO EDIT** Command or **SPF** to store and update members or procedures in the **INCLUDE** library.

Examples

1 **INCLUDE SALES;**

The **INCLUDE** instruction directs that the libraries are to be searched for the

SALES member. The text (Jol source statements) is then interpreted by Jol as though the contents of the **INCLUDE**d member **SALES** had been placed where the **INCLUDE** instruction had been.

After the instructions in **SALES** have been interpreted, the instruction following the **INCLUDE** will be interpreted.

2 **IF %DAY = 'MONDAY'**
THEN INCLUDE STARTWK;

Jol automatically initializes **%DAY** to the current day. **%DAY** is tested; if it is **MONDAY** then the **INCLUDE** instruction is activated and the instructions in member **STARTWK** will be analyzed.

If the day is not **MONDAY**, the **INCLUDE** instruction will be ignored.

3 **INCLUDE PAYSYS;**
RUN VALIDATE;
LIST OUTPUT1;

The **PAYSYS** member is **INCLUDE**d. In this example, **PAYSYS** contains all the program and data set definitions that a group of programmers are working on. This programmer wishes to test a program called **VALIDATE**, and then list its output. Instead of each programmer having separate procedures for their part of the system, all the common elements are coded and stored in member **PAYSYS**. Whenever any programmer in the group wishes to test his or her program, he includes the common elements and then places his instructions after the **INCLUDE**.

In this example, the programmer includes the common elements, then **RUN**'s the program **VALIDATE** and **LIST**s its output.

INVOKE - *Immediate Instruction*

Purpose

The **INVOKE** instruction executes a User written load-module during the preprocessor phase of the Jol Compile.

The **INVOKE** Instruction, in its implicit or explicit form, provides one aspect of Jol's openendedness **-** it allows new Jol instructions to be written by your installation. Additionally, it allows any ordinary problem program to be executed in the Compile Phase rather than the Execute Phase.

```
        INVOKE          module-name [parameters]   ;
or Implicitly
        module-name   [parameters] ;
```

Invoking Programs into the Compiler

Any program may be **Invoked** into the Jol Compiler. Programs written specially for invocation into Jol can:-

- Read or write any **EXTERNAL** Data Set
- Alter and Examine Symbolic Variables directly
- Send back to Jol instructions which Jol will execute as if they had been submitted in the usual manner
- **All**, or any, of the three above.

For example, the Jol **GET** instruction is an invoked routine.

Notes

1. The name of the **INVOKED** module must conform to your Operating System naming standards **-** typically, it cannot be greater than eight characters.
2. The **CALL** instruction can be used in place of **INVOKE -** it allows programs to be executed from other libraries.
3. Full details of writing **Invoked Routines** may be found in the Jol **Systems Programmer Guide**.
4. An alternative method by which a module may be **INVOKEd** is when a non-Jol instruction is found and it is not a macro instruction. In this case the module is executed and may act as if it was a Jol instruction.

Example

1 **INVOKE CHECKJOB JOB19;**

Module **CHECKJOB** is invoked with a parameter **JOB19**. **CHECKJOB** could determine if **JOB19** has terminated, and return an indicator that the job can test.

The JOB Definition

Purpose

The **JOB Definition** supplies details which will be used for accounting purposes, specifying the maximum storage required for the job, and so on.

```
    jobname: JOB  options  ;

  where options may be coded in any order from those listed below.
```

```
    { PROGRAMMER  }
    { SUBMITTER   }      [=]  programmer-name
    { NAME        }

    GROUP           [=]  RACF-group-name

    PASSWORD        [=]  RACF-password

    USER            [=]  RACF-user-id

                    { elapsed,cpu   }
    TIME     [=]    { cpu,elapsed   }    { MINUTES | MINS | M }
                    { elapsed       }    {SECONDS | SECS | S }
 or NOTIMING

    CLASS           [=]   job-class

    { SIZE     }           { K      }
    { REGION   }   [=] n   { BYTES  }

    { VIRTUAL | VIRT        }
    { REALCORE | REAL       }

    { ACCT     }           { accounting-information  }
    { DEPT     }     [=]   { (subaccount1,subaccount2...)      }
    { CHARGE   }           { 'accounting-information'          }

    [  ROOM | BOX     | SUBACCT2                             ]
    [  elapsed-time   | SUBACCT3                             ]
    [  MAXLINES       | SUBACCT4                             ]
    [  MAXCARDS       | SUBACCT5    [=]   sub-accounting     ]
    [  COPIES         | SUBACCT6          information        ]
    [  SUBACCT7                                              ]
    [  SUBACCT8                                              ]


    { PRIORITY   }
    { PRTY       }     [=]    priority-number

    NOTIFY             [=]    user-id

    PERFORM            [=]    performance-group
```

```
HOLD | SCAN

MSGCLASS         [=]    message-class

[  PALLOC | PALL         ]
[  NPALLOC | NPALL       ]

[  PJCL | PJ       ]
[  NPJCL | NPJ     ]

JOBEXTRA         [=]    'extra-job-information'
```

Note Where a value must be supplied, the coding is done in one of four ways. For example:

```
      CLASS=B
or    CLASS B
or    CLASS=(B)
or    CLASS(B)
```

Notes

1. The parameters above are *all* options and need not be coded.

2. The **PROGRAMMER** keyword need not be coded if the name of the submitter contains periods. For example:

   NAME = R. JONES is equivalent to **R. JONES**.

3. The **TIME** keyword is not necessary when **MINUTES** or **SECONDS** is coded. Therefore **TIME = (10,2)MINS** is equivalent to **10,2MINS**.

4. The **TIME** parameter may also be coded on Program Declares.

5. The **SIZE** keyword is not necessary when **K** or **BYTES** is coded. Therefore **SIZE = 60K** is equivalent to **60K**

6. When **ACCT** is coded within quotes, for example:

   '(*account1*, *account2*,...)'

   then the accounting information is copied to the generated JOB card exactly as coded, including any blanks. Also, the **SUBACCT** fields may not be used.

   When **ACCT** is coded **without** quotes, for example:

   (*account1*, *account2*,...)

   any blanks are removed from the generated JOB card. **SUBACCT** fields may be used.

7. For JES2 systems, the **ROOM, BOX, MAXLINES, MAXCARDS** and **COPIES** keywords may be used. The elapsed-time coded in the **TIME** parameter is also copied to **SUBACCT3**. See also the JOBPARM Command.

8. **PALLOC** prints JCL allocation messages. The default is installation defined.

9. **PJCL** prints JCL at execution time. The default is installation defined.

10. **JOBEXTRA** allows the inclusion of JOB parameters that may be included in future Operating System Releases.

Examples

1. **PAYROLL : JOB 150 K CLASS C;**

   Job **PAYROLL** is defined as a class **C** job which will require 150 K of storage.

2. **LIST: JOB 60K USERID DOE PASSWORD DATABASE;**

   LIST is defined as a job requiring 60,000 bytes to execute in. The **RACF** userid of **DOE** and a password of **DATABASE** are also supplied.

JOBCAT - *Execute Command*

Purpose

The **JOBCAT** statement defines a private catalog to be made available by the system to an entire job.

```
JOBCAT        catalog1 [. . . catalog2] ;
```

Normally, the system searches the main system catalog to find data sets. Private catalogs may be used with the **JOBCAT** Command. When you use the **JOBCAT** command, each time you request a data set, the system looks for it in the specified catalog. Another way is to use the **STEPCAT** option in the Program declaration.

Notes

1. The **JOB**CAT command may be coded anywhere in a Jol program.

2. The Catalogs must be cataloged. The maximum number of catalogs that may be specified is **five**.

3. If two or more **JOBCAT** commands are issued, the catalogs from the 2nd and subsequent commands are concatenated to catalogs specified in the earlier commands.

4. If you include a **JOBCAT** command in your Jol program, each time the job requests a data set, the system first searches the private catalog. If it does not find the data set there, it next searches the system catalog.

Example

1.
```
JOBCAT PRIVATE.CATALOG,
       SECOND.CATALOG;
```

The libraries known as **PRIVATE.CATALOG** and **SECOND.CATALOG** are defined as Job Catalogs.

JOBLIB - *Execute Command*

Purpose

The **JOBLIB** statement defines a private program library to be made available by the system to an entire job.

```
JOBLIB  dataset1 [. . . dataset2] ;
```

Unless you inform the system that any program you run resides in a private or temporary library, the system expects to find it in the system library (**SYS1.LINKLIB**) or one of its optionally concatenated libraries.

One way to tell the system that a program resides in a private library is to use the **JOBLIB** Command to inform the system that the named library is to be searched for all programs in the job before the system library is searched. When you use the **JOBLIB** command, each time you request a program, the system looks for it in the private library. Another way is to use the **LIB** option in the Program declaration.

Notes

1. The **JOBLIB** command may be coded anywhere in a Jol program.

2. The Libraries must be cataloged. The maximum number of libraries that may be specified is **five**.

3. If two or more **JOBLIB** Commands are issued, the libraries from the 2nd and subsequent commands are concatenated to libraries specified in the earlier commands.

4. If you include a **JOBLIB** command in your Jol program, each time the job requests a program, the system first searches the private library. If it does not find the program there, it next searches the system library.

5. Use the **LIB** parameter, described under the program Declaration, to define a private library to be made available to one job step in a job.

   If you include a Library statement for a program and a **JOBLIB** command for the entire job, the system first searches the step library and then the system library for the requested program. The job library is ignored for that step.

Examples

1. **JOBLIB PRIVATE.LIBRARY;**

   The library known as **PRIVATE.LIBRARY** is defined as the Job Library.

2.
   ```
   JOBLIB PAYROLL1.LINKLIB
          PAYROLL2.LINKLIB;
   ```

   Two libraries, **PAYROLL1.LINKLIB** and **PAYROLL2.LINKLIB**, are to be used as Job Libraries.

   Whenever a program is to be executed, the two private libraries are searched first for the program. If it is not found in either library, the system libraries are searched in the usual way.

JOBPARM - *Execute Command (JES2 only)*

Purpose

The **JOBPARM** Command specifies extra job information. You may specify the expected elapsed time of your job, and other details like the maximum number of printed lines the job is allowed to produce.

JOBPARM	*options* ;

where options are:

CARDS	=	*maxcards*
COPIES	=	*copies*
{ ELAP }		
{ TIME }	=	*expected*-elapsed-time
FORMS	=	*form-number*
LINECNT	=	*lines-per-page*
LINES	=	*max-lines*
ROOM	=	*room-number*
BOX	=	*box-number*
SYSTEM	=	* | **ANY |** *system-number*
NOLOG		

Notes

1. Most of the parameters above may also be specified in the JOB Definition.

2. **SYSTEM=*** specifies the job is to run on the system the job was read in on; **ANY** specifies any JES2 system; system-number is a four alphameric system-id on which the job is to execute.

3. **JOBPARM** will execute the **MAIN** Command if executed on a machine with JES3 or ASP.

JOLOPT - *Immediate Instruction*

Purpose

The **JOLOPT** Instruction allows you to specify Jol compiler options with a programmed instruction, thus resetting the options to those you desire.

```
JOLOPT        options ;
```

Notes

See the section on Compiler Options for a full explanation of the parameters allowed.

Examples

1. **JOLOPT PM PE;**

   JOLOPT can be used to respecify compiler options. In this example, **PMACRO** and **PEXPAND** are specified. These options inform Jol that it is to list any Macro Commands are they are interpreted, and also to list the expanded generated macro code in the compiler listing file. Specifying **PM** and **PE** is frequently done when debugging new macros as the Jol listing file can then be examined or browsed to find errors, or to show exactly what was generated by your macro.

2. **JOLOPT DOS;**

   JOLOPT DOS informs Jol that is to generate executable code for the IBM DOS/VSE Operating System.

3. **JOLOPT X8;**

   JOLOPT X8 informs Jol that is to generate executable code for the Fujitsu MSP/X8 Operating System.

KEEP - *Execute Instruction*

Purpose

The **KEEP** Instruction ensures that the space allocated for a data set will not be used by another job and that the data contained in the data set may be accessed at a later date.

```
KEEP    { dsid    }    [ (generation-number) ] . . .
        { dsname  }

       [  ON [VOL] volume  [UNIT] unit ]
```

Notes

1. All **OLD** data sets are automatically kept unless a **DELETE** or **SCRATCH** instruction is executed, but all **NEW** data sets are scratched by the Operating System, unless they are **KEPT** or **CATALOGED,** or the data set has a Retention period (see **RETAIN)** or was **PROTECTED** (see **PROTECT** and **READONLY)**.

2. **KEEPing** an **OLD** data set has no effect.

3. Jol will obtain Volume, Unit and Position information from the Data Set Declaration unless a **SCRATCH** volume is associated with a new data set, in which case the information will be found at execution time from the Operating System.

4. The '*dsname*' form of the instruction is not recommended for long production jobs as the **DSID** format allows the Data Set Name to be overridden; thus a data set name may be altered in one place **-** the **DECLARE -** and all references to the **DSID** will have the new Data Set Name.

5. The **ON** option, if specified, must be coded after the **DSID** or **DSNAME** list.

6. If Volume and Unit information is provided in the Data Set Declare or the **ON** option is used for Cataloging a Data Set, Jol does not check that the Data Set is on the specified Volume.

Examples

1. **KEEP DSID1, 'TEST.DATA.SET';**

   Data Set Identifer **DSID1** and Data Set '**TEST.DATA.SET'** will be kept, but not cataloged.

2.
   ```
   DECLARE OP1 DATA SET
           COPY.SYS1.LINKLIB

           CYLS;

   COPY SYS1.LINKLIB TO OP1
           PDS;

   KEEP OP1;
   ```

   The Declare **OP1** defines a data set called **COPY.SYS1.LINKLIB**.

   The **COPY** Command copies the **SYS1.LINKLIB** data set to the **OP1** definition, then the **KEEP OP1** instruction keeps the dataset **COPY.SYS1.LINKLIB**.

Note that the definition of **OP1** does not specify a volume to which the data set should be copied. The system will allocate a volume when **COPY** is performed, and then the **KEEP** instruction will find which volume the datset is placed on, and print the volume name on the system log so that the whereabouts of the data set will be known to the user.

LINK - *Execute Command*

Purpose

The **LINK** Command invokes the system Linkage Editor to convert the object text from the **COB**, **PL1**, **FORTRAN** or **ASM** Commands to load module format. The Load module can then be executed with the **RUN**, **EXEC** or similar commands.

```
LINK      { data-set-list    }   options ;
          { dsid-list        }
```

where *options* are:

```
LOAD     dsid | data-set-name

NAME (program-name)

         { dsid [|| dsid ... ]                    }
LIB      { data-set-name [|| data-set-name] }

{  PRINT        data-set-name | dsid  | NOPRINT   }
{  SYSOUT       * | class                          }

[  PLILIB      ]
[  PLICMIX     ]
[  PLIBASE     ]
[  FORTLIB     ]
[  COBLIB      ]

INCLUDE  {  data-set-name   [(member-list)]   }
         { dsid                                }

AC=0 | 1
SSI       =        system status information
SIZE      =        virtual storage size
DCBS      =        blocksize
LIST | NOLIST
MAP | NOMAP
CROSSREF | XREF
OVERLAY | OVLY | NOVLY
TEST | NOTEST
LET | NOLET
RENT | NORENT
REFR | NOREFR
NCAL | NONCAL
XCAL | NOXCAL
OL | NOOL
PANEL | NOPANEL
NE | NONE
DC | NODC
SCTR | NOSCTR
```

Notes

1. Default values are satisfactory for most purposes.
2. **LOADGO** may be a faster alternative to **LINK**.
3. **TEMPNAME** is the default for the **LOAD** parameter.
4. **SSI** defaults to the current date and time. Thus day 156 in 1987, time 14:15 will become **SSI 71561415**. The **SSI** will be printed when the program **RUN**s, thus providing an audit trail of which version of a production program ran.

Example

1 **LINK OBJ LOAD('PROD.LOAD(PAYUPDTE)') COBLIB;**

The object code specified will be linked and stored in the data set **PROD.LOAD** as member **PAYUPDTE**.

LIST - *Execute Command*

Purpose

The **LIST** Command will print all, or part of, a data set in character, and/or hexadecimal format. Record numbers will be printed with each record, and each set of eight characters will be separated from the next by two blanks.

```
{ LIST }     { dsid   }
{ PM   }     { dsname }          options ;
```

where options are:

options	default
number-of-records **{ RECS }** *to-print* **{ RECORDS }**	**ALL**
[**CHAR | HEX | DUMP**]	**CHAR**
EVERY *nth-record*	**1**
STARTAT *record-number*	**1**

where:

number-of-records-to-print specifies how many records are to be printed

CHAR	specifies that the records are to be listed in character format.
EVERY *n*th-record	specifies that only every *n*th record is to be printed.
HEX	specifies that the records are to be converted to hexadecimal format before listing.
STARTAT	***n***th record specifies at which record in the data set printing is to start.

Notes

1. Indexed Sequential data sets cannot be listed with the **LIST** Command.

2. To list members of a Partitioned Data Set, list each member individually or use the **PRINTPDS** Command.

3. If **STARTAT**, *number-of-records* or **EVERY** is not specified, the entire data set is printed.

Examples

1. **LIST INPUT;**

   The **DSID** called **INPUT** will be listed in its entirety in its character format.

2. **LIST '%SYSUID.TEST.INPUT'
   RECORDS;**

   The first 99 records of the Users data set **TEST.INPUT** are listed.

LISTCAT - *Execute Command*

Purpose

The **LISTCAT** Command is used to list entries from a catalog. Options permit the listing of the entire catalog or a selection based on the higher level indexes.

LISTCAT *options* ;	
options	default

ALL
{ L }
{ LEVEL } [=] *level* **{**
NODE }
	-
	
	SYSUID

VOL, **VOLS** **CVOL, CVOLS** [=] *volume*	-
2314 | 3330 | 3340 | 3375 **3380 | F493** etc.	**3380**

Listing Catalog Entries

The system provides a method of easily finding the whereabouts of data sets **-** the volumes that data sets reside on are entered into the system catalog with the **CATALOG** instruction. Whenever a cataloged data set is used, only its name need be coded and its volume and unit will be found in the catalog. To find out the names of the data sets in the catalog, the **LISTCAT** command may be used.

Node Points

Data set names are usually made up from a series of simple names. For example, **SALES.MASTER** and **SALES.TRANS.FILE** are two data sets made up of simple names, Both data sets have **SALES** as their first name. Similarly, payroll data sets may commence with **PAYROLL**. To list all cataloged data sets commencing with specific names, the **LEVEL** or **NODE** parameter may be used. For example, to list all names commencing with **SALES**, code **LISTCAT LEVEL SALES;**

Notes

1. If no options are specified, all the cataloged data sets from all the catalogs known to the **LISTCAT** instruction are listed using the Users Node Point.

2. If a Node Point or Level Listing is requested without specifying a particular catalog, the **LISTCAT** will request that particular node be listed from all the catalogs known to it.

Examples

1. **LISTCAT;**

   Lists all the Users cataloged data sets.

2. **LISTCAT L(PAYROLL);**

   The **PAYROLL** entry names will be listed from the catalogs.

LISTSYMS - *Immediate Instruction*

Purpose

The **LISTSYMS** instruction lists the Symbolic Variable table in alphabetical order at the terminal. *Only those variables known to the procedure and their current contents are listed.*

```
LISTSYMS  ;
```

Listing the Symbolic Variable Table Data

As Jol is compiling your program, it stores the names of all symbolic variables in a table so that symbolic names can be replaced when necessary. The names and current contents of the variables are stored in the table.

Sometimes, it is desirable to know what variables are currently available to Jol and the contents of those variables. For example, when testing a macro program, you may wish to see what variables have been used, and the effect of any of your Jol instructions on those variables.

The **LISTSYMS** instruction allows you to do this.

Notes

1. **PRINTSYM** may also be used to print the contents of the Symbolic Variable table. **PRINTSYM** prints the table to the Jol listing file, instead of at the terminal.

2. You can also use the **WRITE** instruction to display the values of symbolic variables immediately at the terminal. For example:

   WRITE '%NAME';

   displays the current value of **%NAME** immediately at the terminal.

Examples

1 **LISTSYMS;**

LISTSYMS lists the names and contents of all symbolic variables.

LOADGO - *Execute Command*

Purpose

The **LOADGO** Command is used to invoke the loader which loads the program and executes it.

```
LOADGO    { dsid-list             }   'parms' options ;
          { data-set-name-list    }

where

     'parms' is a parameter to be passed to the user program.

     and options are:-

          EP  [=]  entry-point-of-users'-program

          {  PRINT        {data-set-name | dsid }  | NOPRINT  }
          {  SYSOUT       * | class                           }

          LIB (auto-call-list)

          [  COBLIB    ]
          [  FORTLIB   ]
          [  PLIBASE   ]

          DYNAMIC | NODYNAM
          MAP | NOMAP
          RES | NORES
          CALL | NCAL
          LET | NOLET
          TERM | NOTERM
          LINECNT (line-count)
```

Example of the LOADGO Command

1 **LOADGO (PROG1.OBJ PROG2.OBJ)**
NAME (TESTPGM) PLIBASE;

The object data sets **PROG1.OBJ** and **PROG2.OBJ** are read by the **LOADER** and any external references satisfied by the **PLIBASE** automatic call library. Program **TESTPGM** (which must have been declared earlier) is executed.

LOADVOL - *Execute Command*

Purpose

The **LOADVOL** Command will restore the contents of specified disk volumes created by the **DUMPVOL** Command.

LOADVOL *volume1 [,volume2 ....] options* ;	
options	***defaults***
BACKDSN = *high level of tape data set name*	**JOL.BACKUP.OF.** *volume(0)*
FROMUNIT = *tape-device-name*	**TAPE**
TOUNIT = *disk-device-name*	**DISK**

for single volume restores, you may also code

options		***defaults***
TO =	*specific volume name if to a different volume*	*same as that of the dumped volume*
GEN =	***generation number*** *of tape to be restored*	**(0)**
PURGE		**NOPURGE**
COPYVOLID		**-**

The LOADVOL Command can be used in two ways:-

1 To restore several disks to the same volumes from which they were copied.

2 To restore a volume to a different disk volume and optionally rename it to that of the original dumped volume.

Notes

1 **BACKDSN** specifies the first part of the name of the tape data set on which the volume was copied. It defaults to **JOL.BACKUP.OF***volume***(0)**. Therefore, if volume **DISK01** had been copied to tape with the **DUMPVOL** command, the name of the latest version of that volume on tape would be **JOL.BACKUP.OFDISK01(0)**.

2 **PURGE** must be specified if the receiving volume contains **VSAM** data spaces, or data sets that have not yet expired, otherwise the **LOADVOL** Command is terminated.

Examples of the LOADVOL Command

1 **LOADVOL DASD01;**

This reloads the disk volume **DASD01** from tape; the tape's data set name is **JOL.BACKUP.OFDASD01(0)**. Any data on **DASD01** is destroyed and replaced by whatever data is on the tape.

2
```
LOADVOL DASD02 TO SPARE PURGE
        COPYVOLID;
```

This loads the latest backup tape of volume **DASD02** to a disk called **SPARE**. Before loading, it will scratch all data sets from volume **SPARE**. After the reload has taken place, the volume will be renamed **DASD02** because the **COPYVOLID** parameter was specified.

MACRO - *Immediate Prototype Statement*

Purpose

The **MACRO** statement defines the parameters that may be coded on a Macro Invocation. See the *Jol Reference Manual* or the *Jol General Information Manual* for a full description of writing Macros or Commands.

```
        macroname: MACRO [optional parameters];

    where optional parameters are:

        [ { %parameter-keyword=['initial value']        } ]
        [ { %substitution-keyword                       } ]    [, ...]
        [ { positional keyword                          } ]
```

Notes

1 There must be an **END** to correspond to the **MACRO** statement.

2 There are 4 types of parameters that may be used in a Macro:

a) **Keyword Assignments** are used when the user of the macro codes a keyword followed by a value; for example **UNIT = 3380**. This is coded in the macro prototype as **%UNIT=,**. When **%UNIT** is referenced within the macro, it would contain **''(null)** or **'3380'**, or, in fact, whatever value was coded after **UNIT**.

b) **Substitution Keywords** can be used to detect if a word is coded on the Users command. For example, to see if the word **TEST** was coded, code **%TEST** in the macro prototype instruction. If the user of the macro coded **TEST** on the macro invocation, **%TEST** would be set to **'TEST'** otherwise **''(null)**.

c) **Pointer Keywords** can be used to determine which **%LIST** item contains a particular keyword. This is coded by coding the name without equal or percent symbols. See d) below.

d) **Positional Parameters**. Any Parameter coded on an invocation statement is stored in symbolic locations commencing **%LIST(1)** to **%LIST(n)**, unless it is a keyword parameter. **%LIST** items may be accessed directly, or when used in conjunction with Pointer Keywords, it is possible to determine the previous **%LIST** item.

Example

A Macro is to be set up to allocate space for a Data Set. In keeping with the Jol philosophy, it is required that the user be able to code the parameters in as free-format as possible.

The syntax of the macro command is to be as follows:

```
    ALLOCSPC data-set-name n { CYLS | TRKS } UNIT = unit;
```

By using the Prototype Statement shown on the following page, the User can code the parameters required for the macro in virtually any order.

```
ALLOCSPC : MACRO (CYLS,TRKS,%UNIT=DISK);
```

The User of your command can now write any of the following statements:

```
    ALLOCSPC TEMP.DATA.SET 10 CYLS UNIT SYSDA;
or  ALLOCSPC TEMP.DATA.SET 10 TRKS UNIT SYSDA;
or  ALLOCSPC TEMP.DATA.SET UNIT=SYSDA 10 TRKS;
or  ALLOCSPC 10 CYLS UNIT SYSDA TEMP.DATA.SET;
or  ALLOCSPC 10 TRKS UNIT SYSDA TEMP.DATA.SET;
or  ALLOCSPC UNIT=SYSDA 10 TRKS TEMP.DATA.SET;
or  ALLOCSPC UNIT=SYSDA 10 CYLS TEMP.DATA.SET;
```

Example of Results of the ALLOCSPC Command

If the User codes:

```
ALLOCSPC TEMP.DATA.SET 10 CYLS UNIT SYSDA;
```

then:

%CYLS will be equal to **3** (that is, the 3rd **%LIST** item or **%LIST(3)** is the word **CYLS**).

%UNIT will equal **SYSDA** (the default of **DISK** has been replaced by the word **SYSDA**).

and the **%LIST** array will contain the following:

%LIST(1) contains the *data-set-name* or **TEMP.DATA.SET** in the example above.

%LIST(2) contains the number **10**.

%LIST(3) contains the words **CYLS**. Remember that the *Pointer Keyword Parameter* **CYLS** contains the number **3**, which *points to this %LIST item*. It can be considered as an index into the **%LIST** array.

To find the number of **CYLS** specified by the User is done as simply as coding the following:

```
%LIST (%CYLS-1)
```

Referencing **%LIST (%CYLS-1)** in the example above will return **10** (the number coded before the **CYLS** keyword).

Effective use of the various Jol Macro Parameters makes writing installation or private macro commands a simple process.

The *Jol Reference Manual* and *Jol General Information Manual* contain full details on how to extend the Jol Language by writing Jol Macros or Commands.

MAIN - *Execute Commmand*

Purpose

The **MAIN** command informs the **ASP** and **JES3** processors details about the job. You may indicate which type of Operating System your job must run with, or which computer your job is to run on, and other details such as the maximum number of printed lines the job is allowed to produce.

```
        MAIN          options  ;

    where options are:

        CARDS         =          maxcards

        LINES         =          maxlines

        { WARNING | CANCEL | DUMP }

        CLASS         =          jobclass

        FAILURE       =  { RESTART | CANCEL | HOLD | PRINT }

                              { ANY | JGLOBAL | JLOCAL     }
        SYSTEM        =       { ASP                        }
                              { main-name                  }

        TYPE          =       { ANY | MVT | VS2 | system-id }
```

Notes

1 If you execute this command on a **JES2** system, the **JOBPARM** command will be used instead with the parameters specified on the **MAIN** command.

2 The **MAIN** command may be coded anywhere in the Jol program.

Example of the MAIN Command

1
```
MAIN CARDS 2, LINES = 20, CANCEL
           TYPE = VS2;
```

The **MAIN** command is used to inform the system that there will be a maximum of 2000 cards and 20000 lines of print produced by the job. If these limits are exceeded, the job is to be cancelled. The system the job is to run on is a **VS2** system.

MERGE - *Execute Command*

Purpose

The **MERGE** command allows up to 5 data sets to be **MERGE**d into another data set using parameters either described in the **FIELDS** parameter or those described in the dataset referenced by the **USING** parameter.

```
MERGE   { dsid     }  &    { dsid    }     [& ...]
        { dsname   }       { dsname  }

        TO { dsid | dsname }

        { MERGE FIELDS  =  merge parameters   }
        { USING { dsid  | dsname }            }

where:

        FIELDS = (p1,m1,f1,s1, . . . pn,mn,fn,sn)
```

Notes

1 Up to five (5) Data Sets may be merged with the Merge command.

2 The **USING** parameter specifies that a Data Set contains the **MERGE** parameters. The data set may have been created in the current job, or, it may be an **OLD** data set.

3 The **FIELDS** parameter specifies the **FIELDS** to be used for collating and merging the input data sets.

The format is:

FIELDS = **(**$p_1,m_1,f_1,s_1, \ldots p_n,m_n,f_n,s_n$**)**

where

P	specifies the *position* within the record.
M	specifies the *length* of the field.
F	is the *format* of the field and may be **CH | ZD | ID | FI | BI | FL | AC | CSL | CST | CLO | CTO | ASL | AST | AQ**
S	specifies the desired *sequencing* **A|D|E**

Examples

1 **MERGE DATA1 & DATA2**
TO DATA3
USING MERGPARM;

The data referred to by **DATA1** and **DATA2 DSID**s is to be merged and placed in the data set referred to by **DATA3**. The parmeters for the **MERGE** will be found in the data set referred to by the **DSID MERGPARM**.

2 **MERGE SORTED.ONE & SORTED.TWO**

TO OUTPUT

USING MERGE.CONTROL(MERG001);

Two data sets, **SORTED.ONE** and **SORTED.TWO,** are to be merged into the data set described by the **DSID** called **OUTPUT**. The parameters for the **MERGE** will be found in member **MERG001** of the data set called **MERGE.CONTROL**.

ON ERROR - *Execute Command*

Purpose

The **ON ERROR** command will allow a copy of the processing program's storage to be taken if an abend error occurs.

```
              { SNAP   }
ON ERROR      { DUMP   }   [ SYSTEM ]   [options]   ;
              { MDUMP  }
```

options	default
TO PRINTER *classname*	**PRINTER A**
HOLD	**NOHOLD**

where

classname may be from **A** to **Z** or **0** to **9**.

SNAP, **DUMP** or **MDUMP** specify that the program's storage is to be written to the specified device.

SYSTEM specifies that the system nucleus and a trace table (if available) is to be produced in addition to the program storage area. If **SYSTEM** is not specified and **DUMP** or **SNAP** is specified, only the program storage area is written.

When a program has an error, it will often terminate with an **ABEND**. Suppose you allocate 1 track for for an output data set, but attempt to write more data than will fit in the track. In this case, the system will terminate your program with a **System Abend - D37**. Other type of errors issue different abend codes.

The **ON ERROR** command allows you to specify that you want a dump of the program if an abend occurs, it can be used to specify that no dump is to be taken. Coding:

```
ON ERROR;
```

requests that no dump be taken, and coding:

```
ON ERROR SNAP;
```

requests a dump if an error occurs.

Notes

1 Only one **ON ERROR** should be issued. If multiple **ON ERROR** commands are issued, the details from the last one are used.

2 The **ON ERROR** command should be coded before any **RUNs** or **Commands**.

Examples of the ON ERROR ommand

1 **ON ERROR;**

The **ON ERROR** cancels any previous **ON ERROR** commands, and no dump will be taken if a programming or system error occurs.

2 **ON ERROR SNAP TO PRINTER B HOLD;**

If an **Abend** occurs, the dump will be written to a printer with **CLASS B**. The dump will not be printed until the operator specifies it to be printed because **HOLD** was specified.

OPCNTL - *Execute Instruction*

Purpose

The **OPCNTL** instruction will output the '*text*' as an 80 character card image in the position specified. This instruction is designed to be used in Macros, and provides the capability for creating **HASP, ASP** or **JES** commands.

```
OPCNTL    'text'    { AFTER }     JOBCARD  ;
                    { BEFORE }
```

The **OPCNTL** instruction is designed to allow the creation of control cards for **HASP, ASP** and **JES** systems. It is designed to be used in **Macro** commands (such as the **MAIN** command), and should not be used to create **JCL** statements.

Notes

1 Only one control card can be output before the generated **JOB card**.

2 If neither the **AFTER** or **BEFORE** card option is specified, the card will be placed after any **JCL** produced for the last **RUN** statement or after any **JCL** produced for a **DELETE** or **SCRATCH** instruction.

3 If the text is greater than 80 characters, it will be truncated. If it is less than 80 characters, it will be padded with blanks.

4 If an apostrophe is required, code two apostrophes and they will be converted to one in the generated **JCL**.

4 JCL cards should not be output using the **OPCNTL** instruction.

5 If the **DYNAM** compiler option is specified (that is, run the job without producing JCL), the results are undefined.

Examples of the OPCNTL Instruction

1 **OPCNTL '/*PRIORITY 9' BEFORE JOBCARD;**

The card '**/*PRIORITY 9**' will be placed before the generated Jobcard.

2 **OPCNTL '/*SETUP (DA01)' AFTER JOBCARD;**

The card '**/*SETUP (DA0l)**' will be placed after the Jobcard.

OPEN or OPENFILE - *Immediate Instruction*

Purpose

The **OPEN** instruction allows Data Sets to be read or written with the **READ, WRITE, GETFILE** and **PUTFILE** instructions.

```
        OPEN FILE (     {  ddname  |  filename }    )
   or
        OPENFILE  (     {  ddname  |  filename }    )

              INPUT | OUTPUT | UPDATE  ;
```

Opening Files

Before a Data Set can be **READ** from or **WRITTEN** to, the file must be opened. The **OPEN** or **OPENFILE** instruction opens the Data Set for subsequent processing.

Notes

1. The file must have been previously allocated with the Jol or TSO ALLOC instruction, or with the JCL used to invoke Jol.

2. The file must only contain character information. Binary fixed, decimal, floating point numbers are not yet supported.

3. The maximum record length that may be READ or WRITTEN is 253 characters, which is the maximum permissable length of symbolic parameters.

4. If the file contains Variable length records, the symbolic parameter length is set equal to the length of the record read.

   For Fixed length record files, the size of the symbolic parameter is set equal to the length specified in the file definition.

5. See also the ALLOC, READ, GETFILE, PUTFILE, WRITE, CLOSFILE and FREE instructions.

6. %LASTCC is set to zero if the file was opened correctly, otherwise it is set to a non-zero value.

Example of the OPENFILE Instruction

1.
```
 ALLOC F(INPUT) SYS1.CONTROL.LIB(%DAY);
                                        /* Allocates member depending on the day */

  OPEN F(INPUT) INPUT;    /* Open the file */
  IF %LASTCC ^=0
       THEN EXIT 'OPEN FOR DATA SET FAILED');

  IF ^ EOF(INPUT) THEN
  DO;
       GETFILE INPUT;
       ... perform processing required.
  END;
```

PANEL - *Immediate Instruction*

Purpose

The **PANEL** instruction allows the user to define Full Screen Panels, with or without SAA/CUA compatible Action or Menu Bars. When the **PANEL** instruction is executed the screen is cleared and the information specified is displayed on the screen.

Facilities are provided to display text, display text and allow replies to be entered, and display text with default replies. MenuBars and Help lines may be defined with the appropriate parameters. Up to 23 lines of text and replies may be displayed with the one **PANEL** instruction.

PANEL '*heading*' **MENUBAR (**'*menu items*'**)**
(list of sub parameters)..
HELPLINE ('*help text*'**)**
;

PANEL OPTION [CAPS | ASIS | AUTOSKIP | NOSKIP];

PANEL REREAD FROM *symbolic-name ['error-message']* **;**

where **MENUBAR** consists of a set of keywords choices that represent groups of related actions Users may request,

and **HELPLINE** is a line of text that is automatically centered and displayed on the last line of the terminal,

and where **sub parameters** are:

1. **(***[color, attribute]* **'*text*'** *...***)**

   The *text* is displayed on the screen. Replies may not be entered.

2. **(***[color, attribute]* **'*text*'** *...*, ***symbolic-name, maximum-length*)**

   The *text* is displayed on the screen. Replies may be entered. The maximum length of the reply allowed is indicated in the *maximum-length* field. If a reply longer than the maximum is entered the cursor will move to the next unprotected field.

3. **(***[color, attribute]* **'*text*'** *...*, ***symbolic-name, maximum-length,***
   [color, attribute] **'*default-reply'*** *...***)**

   The text is displayed as 2 above. A *default reply* is placed in the area normally reserved for the reply. The cursor will be placed at the beginning of the default reply, and may be overwritten.

4. **(***[color, attribute]* **'*text*'** *...*, ***symbolic-name, maximum-length,***
   [color, attribute] **'*default-reply'*** *...*,
   message-row, message-column,
   reply-row, reply-column)

   The *text* and *default reply* is displayed as in 3 above. The *message-row, message-column, reply-row, reply-column* are used to position the text and replies anywhere on the screen.

Other Options:

PANEL OPTION *[option]* ;

PANEL REREAD [FROM *variable ['error message']]* *where* ***options*** *are:*

1. **SHOWFIELD | NOSHOWFIELD**

   Data Entry Fields are underlined when the next Panel instruction is executed, thus showing the User the length of each input field. **NOSHOWFIELD** is the default.

2. **AUTOSKIP | NOAUTOSKIP**

   When **AUTOSKIP** is enabled, the Cursor automatically skips to the next Entry Field when the current Entry Field has been filled. **NOAUTOSKIP** is the default.

3. **CAPS | NOCAPS | ASIS**

   When **CAPS** is enabled, data is converted to Capital Letters on entry. **CAPS** is the default.

and

PANEL REREAD FROM *variable ['error message'];*

The **REREAD** option places the cursor at the screen position occupied by *variable* and re-enters input mode. Typically data is validated, and the **REREAD** option is used to prompt the User to correct data if errors were detected.

General Notes

1. If the **PANEL** is executed by a background job, the default(s), if any, are used. No attempt is made to display data on any screen.
2. Any reply areas and default replies are highlighted. On color screens, the replies will be a different color from the messages.
3. It is recommended that specific row and column numbers are not used, or are used sparingly, as it is then difficult to insert new lines at a later time, or alter the display.
4. To force the display to skip a line, code a slash (/) between the parameters.
5. To turn the **PANEL** command off, the Jol command **FS OFF** may be used. This turns the **PANEL** off, and it behaves as if it were running in a background job; no data is displayed on the screen, and the defaults (if any) are used instead of input from the terminal.

   FS ON will enable terminal processing once more.
6. If a **PFK** (Program Function Key) is pressed instead of the Enter key, **%SYSPFK** is set to the number of the **PFK** key pressed.
7. The **PA2** or **ESC** key, if used, causes the screen to be refreshed. If a mistake is made entering data, pressing the **PA2** key will redisplay the screen as it was before any data was entered.
8. Temporary color changes may be made by coding **'*text*'** as:

   HIGHLIGHT | HI | BOLD | NORMAL '*text*'

   or

   [**BRIGHT**] *color* '*text*'

   where *color* may be:

   BLACK | RED | GREEN | YELLOW | BLUE | MAGENTA | CYAN | WHITE | INVISIBLE | UNDERLINE | FLASH | REVERSE

The default colors may be specified with the **RED, BLUE, BLACK** and other color instructions.

Colors may be disabled with the **BW** Compiler Option, or with the **JOLOPT BW;** instruction

In addition, the following *attributes* may be used:

BOLD, HI	Uses the color specified with the *color* **HILIGHT** instruction.
CENTRE, CTR	Centres the text on the screen.
NORMAL	Reverses any color or attribute information, reverts to defaults.
BLINK, FLASH	Any following text blinks.
UNDERLINE	Any following text is underlined.
NORMAL	Reverses any color or attribute information, reverts to defaults.
REVERSE	Displays the text in Reverse Video.
INVISIBLE	Makes the text and replies invisible.
LM *number*	Set the Left Screen Margin.

9. Coding **CTR** before any color options forces the text to be Centered.
10. Coding **BOX** or *color* **BOX** as the first parameters on the **PANEL** instruction draws a box around the Panel.
11. Coding **LM** *number* (**L**eft **M**argin) *before* any text sub-_parameters moves all text to the right by the specified amount until another **LM** sub-command is found.

MenuBar Notes

1. Action or Menu Bars may be specified by coding the keyword **MENUBAR** ('*keyword list*').

   For example, coding **MENUBAR('Files Dir')** will display the words **Files** and **Dir** on the top line of the terminal.

   The User may type the letter **F** or **D** in the input field provided to select the appropriate action, or move the cursor to the keyword and pressing enter.

   The symbolic variable **%SYSMENU** will contain the *capitialised keyword in full* or null (''). **%SYSMENU** can the tested with an **IF** statement, and action taken based on the value returned by the User. Typically, a **POPUP** instruction will follow to allow the User to make further choices.
2. If a keyword contains an uppercase character, then that character becomes the *hotkey*.
3. If all the words are coded in lowercase, then the first letter of the word is assumed to be the *hotkey*.
4. Braces ({}) may be coded around any letter in the keyword to specify the *hotkey*.

Examples of the PANEL Instruction

1. To create a **PANEL** to request information to backup a data base, the following may be used:

```
Panel (Center Cyan 'Data Base Backup Menu')
  LM 10 /* Set Left Margin to 10 */
  // ('Enter the name of the Data Base',DATA,44)
  / ('Enter Volume Number of Tape',Tape,44,'SCRTCH')
  ('Fast or Slow Backup?',Backup,4,'FAST')
```

The **PANEL** above will produce a screen similar to:

```
                    Data Base Backup Menu

      Enter the name of the Data Base     _

      Enter Volume Number of Tape        SCRTCH
      Fast or Slow Backup?                FAST
```

The cursor is positioned at the first reply position and the details are entered.

The **TAB** key skips the cursor to the next input field.

When the data has been entered and the **RETURN** key pressed, the data entered by the terminal user is stored in the Symbolic Variables for later use.

2. The following example shows how to make a screen similar to that on the front cover.

```
panel option caps;
panel
   menubar ('Files Applications BuildJob Compile Run/Submit'
        ' Options eXit Help'
   (ctr %hedcolor 'Universal Command Language Primary Menu')
   (ctr 'Place the first character of the command you wish to select in'
            ' the box')
   (ctr 'or code any Command directly in the command line.')
   (hi ' Command === > ',command,60,'%commstrt') /* Leave last command */
   (' The following Jol menus are available:')
   (hi ' A ' normal 'Run Other Programs, Spreadsheets,'
       ' Wordprocessing etc.')
   (hi ' B ' normal 'Builds a Job to Execute Later.')
   (hi ' C ' normal 'Compiles Programs (Asm, C, Basic and PL/I).')
   (hi ' F ' normal 'Browse, Copy, Rename or Delete Files; '
       ' Drop to Unix/DOS/TSO.')
   (hi ' H ' normal 'Help and Learn Facilities.')
   (hi ' S ' normal 'Submit Jobs and Programs '
        'to Background or Mainframes.')
   (hi ' O ' normal 'Configure Jol or Set Dos Options'
        ' (TIME, DATE etc).')
   (hi ' R ' normal 'Runs Programs or Jobs.')
   (hi ' X ' normal 'Leave Jol.')
       helpline('Press F1 for Help Screen; F3 to Return To Jol Command Mode')
     ;
```

After the panel has been displayed, code similar to the following may be used to decide on actions to be taken based on the keys pressed by the User.

... check for SYSPFK keys

....

... check for Menubar Selection

```
If sysmenu='EXIT' `then exit quit;
If sysmenu='APPLICATIONS' `then $applic;
If substr(sysmenu,1,3)='RUN'
then do;
      popup
      ( 'Job ')
      ( 'Program ',exec /* program */)
      ;
      if syspop='JOB'
      then do;
            runjob;
            goto endjob;
      end;
end;
```

Another example, checking for input errors, can be found in the **GOTO** instruction. Further examples of **PANEL** instructions can be found in the *Teach Yourself Jol (CAIJOL)* course.

PLI - *Execute Command*

Purpose

The **PLI** command is used to invoke the PL/1 Compiler to convert PL/1 source to object code. The object code may be input to the **LINK** or **LOADGO** command and executed.

PLI { *dsid* | *data-set-name* **}** ***[*** *options* ***]*** ;

where ***options*** *are*:

PRINT **{** *data-set-name* | *dsid* **} | NOPRINT**
or
SYSOUT * | *class*

DECK **{** *dsid* **|** *data-set-name* **} | NODECK**

LIB **{** *dsid* **|** *data-set-name* [|| ...] **} | NOLIB**

MDECK | MD { *dsid* | *data-set-name* **} | NOMDECK | NMD**

OBJ **{** *dsid* **|** *data-set-name* **} | NOOBJ**

ASTER | NOASTER

ATTR | A (SHORT | FULL | S | F) | NOATTR | NOA

{ CHARSET | CS } (48 | 60 | EBCDIC | E | BCD | B)
or **CHAR48 | CHAR60**

NOCOMPILE | **NOC (W | E | S)**
or **COMPILE | C**

CONTROL (*password*)
COUNT | NOCOUNT | NCT | CT
DUMP | NODUMP | DU | NDU
ESD | NOESD
EBCDIC | BCD
FLAG (I | W | E | S) **| F(I | W | E |** S)
FLOW (*number1, number$_2$*) **| NOFLOW**
GONUMBER | NOGONUM | GN | NGN
GOSTMT | NOGOSTMT | GS | NGS
INCLUDE | NOINCLUD | INC | NINC
INSOURCE | NOINSOUR | IS | NIS
INTERRUPT | NOINTER | INT | NINT
LINECNT (*number*) **| LC**(*number*)
LIST | NOLIST
LISTSTMT (*number1, number2*)
LMESSAGE | SMESSAGE | LMSG | SMSG
MACRO | NOMACRO | M | MN
MAP | NOMAP

MARGINI | MI ('*character*')
or **NOMARGINI | NMI**

MARGINS | MAR ((*number1,number2,number3* **)**
NEST | NONEST
NUMBER | NONUMBER | NUM | NONUM

OFFSET | <u>NOOFFSET</u> | OF | NOF

OPT | OPTIMIZE (0 | 2 | 3 | **TIME** | **<u>NOOPT</u>)**
<u>OPTIONS</u> | NOOPTINS | OP | NOP
<u>PANEL</u> | NOPANEL
REENTRANT | <u>NORENT</u> | RENT | NRENT
SEQ (*number1*, *number2*)
SIZE (*number* **|** *number* **K**)
<u>SOURCE</u> | S | NOSOURCE | NS
SPACE | NOSPACE | SP | NSP
<u>STMT</u> | NOSTMT
<u>STORAGE</u> | NOSTORAGE | STG | NSTG
SYNTAX | NOSYNTAX (**W | E | S**)
TERM | <u>NOTERM</u>
XREF (**<u>SHORT</u> | FULL | S | F**)

Notes

1. Under TSO, **SYSOUT** defaults to **SYSOUT = %TSOCLASS** so that you can view the data at the terminal. Otherwise, **SYSOUT** defaults to the installation defined **SYSOUT** class.

2. **OBJ** defaults to a temporary data set **&OBJ**. It may be referenced by the **DSID OBJ** for the **LINK** command.

Example

1
```
PLI SOURCE (VALIDATE)
     INCLUDE('INSTLN.INCLUDE');

LINK OBJ LOAD = TEST.LOAD (VALIDATE)
               PLILIB;

RUN VALIDATE;
```

POPUP - *Immediate Instruction*

Purpose

The **POPUP** instruction allows the user to define a SAA/CUA compatible Pull-down Selection Panel. When the **POPUP** instruction is executed, the choices coded overlay the current Panel.

Up to 20 lines of text may be displayed with the one **POPUP** instruction.

POPUP [**AT** *screen-column*]
('*selection-text' [, optional-command]***)**
...
;

where *selection-text* consists of a set of phrases or choices that represent an action Users may request.

selection-text' may be:

1. **(***[color, attribute]* **'*text*'** *...***)**

The *text* is displayed on the screen. The User may choose the action by using the cursor, or by use of the appropriate *hot-key*.

2. **(***[color, attribute]* **'*text*'** *...* **, *command*)**

The *text* is displayed on the screen. The User may choose the action by using the cursor, or by use of the appropriate *hot-key*. *When the choice is selected, the* ***command*** *is executed.*

Notes

1. Action choices are specified by coding a list of phrases or sentences in a list.

   For example, coding:

   POPUP ('Add to Data Base')

   ('Delete from Data Base');

   will display the choices in a box on the terminal.

   The User may type the letter **A** or **D** in the input field provided to select the appropriate action, or move the cursor to the line of text describing the action and pressing enter.

   The symbolic variable **%SYSPOP** will contain the *capitialised phrase in full* or null (''). **%SYSPOP** can be tested with an **IF** statement, and action taken based on the value returned by the User.

   If the phrase contains an uppercase character, then that character becomes the *hotkey*.

2. If all the words are coded in lowercase, then the first letter of the word is assumed to be the *hotkey*.

3. Braces ({}) may be coded around any letter in the keyword to specify the *hotkey*.

4. If the **POPUP** is executed by a background job, the default(s), if any, are used. No attempt is made to display data on any screen.

5. If the **AT** *screen-column* option is not used, the **POPUP** will be displayed underneath the appropriate **PANEL** MenuBar.

6. To force the display to skip a line, code a slash (/) between the parameters.

7. To turn the **POPUP** command off, the Jol command **FS OFF** may be used. This turns the **POPUP** off, and it behaves as if it were running in a background job; no data is displayed on the screen, and **%SYSPOP** will contain a null or empty string.

   FS ON will enable terminal processing once more.

8. If a **PFK** (Program Function Key) is pressed instead of the Enter key, **%SYSPFK** is set to the number of the **PFK** key pressed.

9. The **PA2** or **ESC** key, if used, causes the screen to be refreshed. If a mistake is made entering data, pressing the **PA2** key will redisplay the screen as it was before any data was entered.

10. Temporary color changes may be made by coding **'*text*'** as:

    [**BRIGHT**] **BLACK | BLUE | CYAN | GREEN | MAGENTA | RED | WHITE | YELLOW** '***text***'

    The default colors may be specified with the **RED, BLUE, BLACK** and other color instructions.

    In addition, the following *attributes* may be used:

BOLD, HI	Uses the color specified with the *color* **HILIGHT** instruction.
CENTRE, CTR	Centres the text on the screen.
NORMAL	Reverses any color or attribute information, reverts to defaults.
BLINK, FLASH	Any following text blinks.
UNDERLINE	Any following text is underlined.
NORMAL	Reverses any color or attribute information, reverts to defaults.
REVERSE	Displays the text in Reverse Video.
INVISIBLE	Makes the text and replies invisible.
LM, RM	Set the Left and Right Screen Margins.

Example of the POPUP Instruction

To create a **POPUP** to request information to backup a data base, the following may be used

```
popup
              ('Backup'          ,backup)
              ('Copy'            ,copy)
              ('Delete'          ,del)
              ('Format Disk or Tape' ,format)
              ('Load/Restore'    ,restore)
              ('Merge Files'     ,merge)
              ('Sort Files'      ,sort)
              ('Rename'          ,rename)
  ;
```

The **POPUP** above will produce a screen similar to:

Backup
Copy
Delete
Format Disk or Tape
Load/Restore
Merge Files
Sort Files
Rename

The cursor is positioned at the *hotkey* entry position awaiting user input.
When the data has been entered and the **RETURN** key pressed, the data entered by the terminal user is stored in the Symbolic Variables for later use.

PRINT - *Execute Command*

Purpose

The **PRINT** command copies the data specified to a printer. Options are provided so that the data may be printed using special forms or at special destinations.

PRINT { *dsid* | *data-set-name* } *[options]* ;

options		*default*
CLASS = **A** to **Z, 0** to **9** or *****		*****
[**ON**] *n* [**PART**] [**STATIONERY**] *or* **FORM** = form-id		**1**
COPIES = *number-of-copies*		**1**
{ DEST | ROUTE } = *destination*		
FREE [AT] {CLOSE | END }		-
HOLD		-
OUTLIM		-
record format		
RECFM = *record-format*	-	-
LRECL = *record-length*	-	-
BLKSIZE = *block-length*	-	-

Notes

1. As supplied, the standard system sequential copy utility is used in the **PRINT** command to transfer the data to the printer. Your installation may alter the **PRINT** command to use another utility.

2. If the data to be printed does not contain print control characters, a new page will be commenced when the previous page has been filled. The length of the page to be printed may be altered with the **MAIN** or **JOBPARM** commands.

Examples

1 **PRINT REPORT;**

The **PRINT** command will allocate a spooled printer and the data referred to by **DSID REPORT** will be transferred to the printer. Because the printer is a spooled printer, the data will actually be printed at a later time.

2 **PRINT JOL.INCLUDE(SALES), COPIES 3;**

The contents of member **SALES** in the **JOL.INCLUDE** library will be printed. **COPIES 3** was specified, and so three copies of the data will be printed.

3 **PRINT PAY.PRINT.FILE(0) FORM CHECK HOLD;**

The contents of the latest version of the **PAY.PRINT.FILE** generation data set are to be printed on a special form called **CHECK**. In addition, the **HOLD** option specifies that printing is not to actually commence until the operator starts it.

PRINTSYM - *Immediate Instruction*

Purpose

The **PRINTSYM** instruction prints the entire Symbolic Variable table on the Jol listing file. The variable names and their current contents are listed.

```
PRINTSYM  ;
```

Printing the Symbolic Variable Table Data

As Jol is compiling your program, it stores the names of all symbolic variables in a table so that symbolic names can be replaced when necessary. The names and current contents of the variables are stored in the table.

Sometimes, it is desirable to know what variables are currently available to Jol and the contents of those variables. For example, when testing a macro program, you may wish to see what variables have been used, and the effect of any of your Jol instructions on those variables.

The **PRINTSYM** instruction allows you to do this.

Notes

1 **LISTSYMS** may also be used to view the contents of the Symbolic Variable table. **LISTSYMS** lists the table in sorted order, directly to the screen.

2 **PRINTSYM** copies the data to the Jol listing files. You can use Browse or Edit to look at file when your compilation is complete.

If your are running Jol interactively, you can request Jol to print the data immediately at your terminal instead of to a data set. For example:

JOL * PRINT(*)

invokes Jol and allows you to enter instructions directly. The Jol listing file is displayed immediately at the terminal.

3 You can also use the **WRITE** instruction to display the values of symbolic variables immediately at the terminal. For example:

WRITE '%NAME';

displays the current value of **%NAME** immediately at the terminal.

Examples

1 **PRINTSYM;**

PRINTSYM lists the names and contents of all symbolic variables.

PROTECT - *Execute Command*

Purpose

The **PROTECT** command is used to Add/Replace/Delete or List the Password of a data set.

PROTECT *data-set-name*

[**ADD** *password*]
[**REPLACE** *old-password new-password*]
[**DELETE** *password*]
[**LIST** *password*]
[**DATA** *data-string*]
[**PWREAD | NOPWREAD**]
[**PWRITE | NOWRITE**]
[**VSAM**] ;

Example

```
PROTECT PAYROLL.MASTER.FILE;
```

PUTFILE - *Immediate Instruction*

Purpose

The **PUTFILE** instruction is used to write a record to a file previously opened with the **OPENFILE** instruction. Records are written from the symbolic variable with the same name as the filename.

```
PUTFILE  filename ;
```

The **PUTFILE** instruction writes a record from the symbolic variable whose name is specified following the **PUTFILE** instruction. The name is also used as the **DDNAME** or **FILENAME** that the Data set was allocated with.

Notes

1. The file must have been **OPEN**ed with the Jol **OPENFILE** instruction.
2. **%LASTCC** is set to zero if a record was written successfully, otherwise it is set non-zero.
3. The **WRITE** instruction offers more flexibility than the **PUTFILE** instruction, as data may be written from any Symbolic Variable.
4. The file must only contain character information. Binary fixed, decimal, and floating point numbers are not yet supported.
5. The maximum record length that may be read or written is 253 characters, which is the maximum permissable length of symbolic variables.
6. If the file is a Variable file, the record length is set equal to the length of the symbolic variable. If the data is enclosed in quotes (**TYPE = 'LIT'**), the quotes are removed before output.

   For Fixed length record files, the symbolic variable is padded with blanks if the length is less than the file's record or length, or truncated if it is too long to be contained in one record.
7. See also the **ALLOC, OPENFILE, READ, GETFILE, WRITE, CLOSFILE** and **FREE** instructions.

Examples

1

```
OPENFILE OUT OUTPUT;                    /* Open the file */
%OUT='THIS IS DATA TO BE WRITTEN';
PUTFILE OUT;
```

The data contained in the symbolic variable **OUT** is written to the file known as **OUT**.

2

```
%FILE='FILE1';

OPENFILE %FILE OUTPUT;                  /* Open the file */

SET %FILE = 'THIS IS DATA TO BE WRITTEN';

PUTFILE %FILE;
```

In the above example, **%FILE** is used as an indirect variable. Using the **WRITE FILE** instruction, the above could have been written more simply by:

```
OPENFILE FILE OUTPUT;
WRITE FILE(%FILE) FROM('THIS IS DATA TO BE WRITTEN');
```

READ - *Immediate Instruction*

Purpose

The **READ** instruction is used to read data into a symbolic variable from either a file previously opened with the **OPENFILE** instruction, or from the Terminal. Records are returned one at a time.

> **READ** *symbolic name* ;
>
> *or*
>
> **READ FILE (***filename***) INTO (***symbolic-variable*);

General Notes

1. The maximum record length that may be **READ** is 253 characters.
2. The names do not have the **%** symbol preceding them; this allows **READ** instructions to be created/modified with symbolic variables.

Specific Notes for TERMINAL Input

1. Use the **WRITE** instruction to notify the user of the expected input.
2. The input may be a string of characters. If a quoted string is entered, the **TYPE** function will return the value **LIT;** similarly for numbers and character strings.
3. The **READ** instruction is ignored in a background or **BATCH** job.

Specific Notes for FILE Input

1. The file must have been **OPEN**ed with the **OPENFILE** instruction.
2. **IF EOF(*filename*)**; may be used to determine if the file is at the end-of-file.
3. **%LASTCC** is set to zero if a record was read successfully, otherwise it is set non-zero.
4. Files may be copied/merged with a combination of **READ**, **WRITE** and **REDO** instructions, with processing on the contents of the file being performed.
5. The file must only contain character information.
6. For Fixed length record files, the size of the symbolic variable is set equal to the length specified in the file; for Varaible length files it is set equal to the length of the record read.
7. See also the **ALLOC, GETFILE, PUTFILE, WRITE, CLOSE** and **FREE** instructions.

Examples

1

```
IF %SOURCE = ''
THEN DO;
    WRITE 'ENTER SOURCE STATEMENT LIBRARY';
    READ SOURCE;
END;
```

If the symbolic variable **SOURCE** is *null* (that is, blank or empty) then the **WRITE** instruction will write a message to the terminal. The **READ** instruction then waits for the user to enter the data and stores it in the symbolic variable **SOURCE**.

2
```
OPENFILE INPUT INPUT; /* Open the file */
IF ^ EOF(INPUT) THEN
DO;
    READFILE INPUT;
    ... perform processing required.
END;
```

REDO - *Immediate Instruction*

Purpose

The **REDO** instruction gives the user a looping facility.

> **LABEL** *label*;
>
> *any **JOL** code*;
>
> **REDO** *label [FOREVER]* ;
>
> *where*
>
> ***label*** is a 1 to 7 character label name.
>
> The keyword **LABEL** must appear in the Jol code to identify that the user may wish to **REDO** that section of the code.

The **REDO** statement allows you to **REDO** a set of instructions **within a macro** or **an included member**; it allows you to repeat Jol code in the preprocessor phase.

You can **REDO** from any nested level of a **DO** group to a lower level, for example:

```
LABEL A;
IF statement
THEN DO;
    Jol statements
    . . .;
    IF expression
    THEN DO;
        Jol statements
        . . .;
        IF . . .
        THEN DO;
            REDO A;
        END;
    END;
END;
```

With this example the **REDO** instruction is coded at the third level and refers back to a first level Jol statement.

Notes

1. See also the **GOTO** instruction.

2. The **REDO** instruction cannot be used for execution type statements, for example:

   STEP01: RUN PROG1;
   IF PROG1 > 0 (e.g. did not not run)
   THEN REDO STEP01 ;

   is **incorrect** and will not be allowed in Jol.

3. To stop possible continuous looping there is a limitation of 50 loops in a **REDO** command. An error message will appear on the screen and allow the user to stop or continue. If Jol is running as a batch job, the loop will stop after fifty iterations.

   Use the **FOREVER** option to repeat the loop indefinitely.

Example

Suppose you require a tape number to be entered by the user of your Jol program. You decide that the tape number must be from 1 to 25, and wish to validate that the User has entered a number within that range. The following code will be repeated until the correct value is entered.

```
      PANEL ('TAPE NUMERICS MUST RANGE FROM 01-25')
           ///     /* Space Screen Down */

           ('ENTER TAPE NUMBER =====>',ANS1,2,'10')
           ;

  LABEL VALIDATE;
      TYPEANS1 = TYPE(ANS1);

      IF      TYPEANS1 ^= 'NUM'
      |       ANS1 > 25 | ANS1 < 10
      THEN DO;
             PANEL REREAD FROM ANS1  /* Place Cursor at ANS1 */
                 'TAPE NUMBERS MUST BE 01 to 25';
                          /* Error Message is at Top of Screen */
                REDO VALIDATE;
      END;
```

If the terminal Operator does not type in a number then the **PANEL** is redisplayed again and the error message appears on the screen. If the Operator types in a number that is too small or too large then the **PANEL** is also redisplayed.

The **REREAD FROM** clause places the cursor at the beginning of the field in error as a convenience for the Operator. The message that follows is displayed on the top of the terminal.

You may have more than one error message on the Panel and verify each field before redisplaying the screen rather than every time an error is detected. You can also put the error message above or below the corresponding error and thus not confuse the Operator where the error is as this example shows:

```
ERR01=' ';
ERR02=' ';
LABEL SCR01;

PANEL ('TAPE NUMERICS MUST RANGE FROM 10-25')
('                              %ERR01')
('ENTER TAPE NUMBER =====>',ANS1,2,'10')
('                              %ERR02')
('IS THIS A BACKUP JOB ==>',ANS2,1,'N') ;

ERR01=' ';   /* Clear Error Message 1 */
ERR02=' ';       /* Clear Error Message 2 */

TYPEANS1 = TYPE(ANS1);

IF  TYPEANS1 ^= 'NUM'
|    ANS1 > 25 | %ANS1 < 10
THEN ERR01='TAPE NUMBERS MUST BE IN THE
         RANGE OF 01 to 25';

IF ANS2='N' | ANS2='Y'
THEN ;

ELSE ERR02='ANSWER Y OR N ONLY';

/* See if any messages to be displayed  */

IF ERR01 ^= '' |  ERR02 ^= ''
THEN REDO SCR01;
```

The **PANEL** will be re-displayed until the correct values have been input.

REGISTER - *Immediate and Execute Command*

Purpose

The **REGISTER** command is used to Register information about programs to Jol. After registration, Jol knows which compiler to call when a compilation is required (see **COMPILE** command), the name of the Load Module library, and the internal filenames (or DDNAMES) that are used for input and output when the program is to be executed (see **EXEC** command).

REGISTER program **{ PROG | PROGRAM }**
options ;

When **REGISTER**ing programs, the following options apply:-

LANG	-	**COBOL | PLI | ASM | FORT**.
FUNC	-	'comments regarding the program function'.
SOURCE	-	Source module library name containing the program source code.
LOAD	-	Load module library which contains an executable copy of the program will be saved when it is compiled and linked.
FILES	-	A list of internal program file names or DDNAMES followed by **READS, WRITES**, **UPDATES** or **MODS,** or **MAY READ, MAY WRITE, MAY UPDATE** or **MAY MOD**.
COMPOPT	-	A list of compiler options to be used when the program is compiled.
LINKOPT	-	A list of linkage editor options to be used when the program is linked.
AUTOCALL	-	A list of libraries which will be searched by the linkage editor to include installation subroutines. There is no need to specify **SYS1.COBLIB** for **COBOL,** or **SYS1.PLIBASE** for **PLI** etc., as they will be included automatically.
REPLACE	-	If coded, causes the program to be re-registered, in other words the current registration is overwritten by the new specifications.

Notes

1. When used interactively, you will be prompted for the required information.

2. A Program can only be **REGISTERED** once unless the **REPLACE** option is used to re-register the entire program. This will then replace the existing **REGISTRATION**.

3. When coding the **FILES** parameter you must code the **DDNAME** followed by a keyword indicating whether the **DDNAME** is to **READ** or **WRITE** etc. For example:

   ```
   FILES (SYSUT2 WRITES, SYSIN READS)
   ```

   specifies that file **SYSUT2** will write to a data set or data set identifier and file **SYSIN** will read a data set when the program is executed by an **EXEC** command.

4. The order in which the files are specified in the **FILES** parameter is not critical. It is recommended that all output files be specified first, followed by input files.

5. When using the **EXEC** command to execute a program, the list of data sets or data set identifiers to be used by the program to be specified in **EXACTLY THE SAME ORDER** as the **DDNAMES** or filenames were specified in the **REGISTER** command.

   Thus:

   ```
   REGISTER PROG1
            FILES (SYSUT2 WRITES, SYSIN READS)  ;
   ```

   forms a list for the **EXEC** instruction in the order:

   ```
   SYSUT2
   SYSIN
   ```

   A subsequent **EXEC** instruction must be coded in this way:

   ```
   EXEC PROG1
            OUTPUT.DATA.SET,INPUT.DATA.SET;
   ```

   The **EXEC** will then map **'OUTPUT.DATA.SET'** with file **SYSUT2** (the first file specified in the **REGISTER** command) and **'INPUT.DATA.SET'** with file **SYSIN** (the second specified in the **REGISTER** command).

Example

1
```
REGISTER UPDATE PROGRAM
        FUNC 'THIS PROGRAM UPDATES SALES MASTER'
        LANG PLI
        SOURCE TEST.SOURCE
        LOAD TEST.LOAD
        FILES   (OUTMAS WRITES,
                      PRINTER WRITES,
                      INMAS READS,
                      TRANS READS)
        COMPOPT 'LIST,XREF'
        LINKOPT 'LIST,XREF,MAP'
        AUTOCALL 'INSTLN.SUB.ROUTINES';
```

The example above registers program **UPDATE** to Jol. The **FUNCT** text is stored in the **JOL.PROGRAM.REGISTER** so that the program's function can be seen easily.

- The **LANG** parameter specifies that the program is written in **PLI**.

- Coding **SOURCE** = **TEST.SOURCE** specifies that the program source code is stored in the **TEST.SOURCE** library.

- **TEST.LOAD** is the Load Module data set that the program will be saved in when the program is link edited. It is also the **default** library from which the program will be loaded when it is executed.

- The **FILES** parameter specifies all the **DDNAMES** or internal file names that the program will use and indicates whether they be used for input, output and so on. In this case **DDNAMES OUTMAS** and **PRINTER,** will be written to while **INMAS** and **TRANS** will be read.

- The **COMPOPT** specifies standard **default** compiler options to be used whenever the **COMPILE** command is used.

- **LINKOPT** and **AUTOCALL** are link editor options and automatic call libraries respectively. The **INSTLN.SUB.ROUTINES** automatic call library will be searched for any pre-written subroutines the installation may provide.

RENAME - *Execute Command*

Purpose

The **RENAME** command is used to rename a data set or a member of a partitioned data set. It can also add an alias to a name. The data set, if cataloged, is recataloged under the new name.

```
RENAME   old-dsname  new-dsname  [ALIAS]
                                 [VOL volume UNIT unit] ;
```

Examples

1 **RENAME 'SYS.PROCLIB(TESTPROC)'**
(NEWPROC);

Member **TESTPROC** is renamed to member **NEWPROC** in library **SYS1.PROCLIB**.

2 **RENAME 'TEST.DATA.SET1'**
'OLD.TEST.DATA';

3 **RENAME 'DEPT1.*.ASM'   'DEPT2.*.ASM';**

RETURN - *Execute Command*

Purpose

The **RETURN** command writes the message to the Job's Log and terminates the job immediately.

RETURN	'*message*'	;

After execution of the **RETURN** instruction, the remainder of the job is flushed, and no further processing is possible. Even if coded the **IF ERROR** does not receive control.

Notes

1. The message must be less than 100 characters.

2. The message is not displayed on the Operator's Console. To do this, use the**TYPE**, **SIGNAL ERROR** or **STOP** instructions.

Examples

1

```
RETURN 'SIMPLE EXAMPLE OF RETURN';
```

After executing the **RETURN,** the message '**SIMPLE EXAMPLE OF RETURN**' will be displayed on the Job's Log, and processing will be terminated immediately.

2

```
RUN DAILY;
IF DAILY ^= 0 THEN
     RETURN '**** WEEKLY PROCESSING WILL NOT RUN ****';
ELSE SUBMIT WEEKLY;
```

The program **DAILY** is executed, and if it returns a non zero returncode, it means that **WEEKLY** processing does not have to be performed. In that case, the **RETURN** instruction is executed, and it places a message for Audit purposes on the Log, and terminates the job.

If **DAILY** did return 0, the next job (**WEEKLY**) is submitted to the system for execution.

ROUTE - *Execute Command*

Purpose

The **ROUTE** command directs all printed or punched output to a specified Printer or Punch.

```
ROUTE   { PRINT  | PUNCH }

            [TO]

            { LOCAL  |  RMTn  |  PRINTERn  |  PUNCHn  }
```

Notes

1. This command creates a **ROUTE** card for **HASP** or **JES2**, and a **FORMAT** card for **ASP** or **JES3**.

2. The names **RMT*nn*** **PRINTER*n*, PRINTER*nn*** and **PUNCH*n*** are installation defined.

Examples

1 **ROUTE PRINT TO LOCAL;**

The **ROUTE** command instructs the Operating System to print all the results **LOCALly**.

2 **ROUTE PRINT TO RMT3;**

The **ROUTE** command directs the Operating System to print all the results at the destination known to it as **RMT3**.

RUN - *Execute Instruction*

Purpose

The **RUN** instruction loads and executes the program described in the **Program** Declaration statement. All the data sets required by the program are made available before the program is executed.

```
[label:]

     RUN   program-id  [PARM] [=]   {number | parameters }
```

Notes

1 All new data sets are **DELETEd** at their last use in the Job unless a **KEEP** or **CATALOG** instruction is executed for them.

Therefore, it is possible to delay **KEEP**ing or **CATALOG**ing a data set until later in the job, or even until the end of the job, thus simplifying restarts.

2 Any data sets used or created by the program are *automatically retained* or *passed* if they are required by a later program or instruction.

If **PROGl** is run, and it produces data set **A.B.C**, then another program can read data set **A.B.C automatically**; that is, there is no requirement on your part to ensure that the data set will still be available for the second program as Jol does this function for you. For example:

```
DCL TESTDS DS
      A.B.C 5 TRACKS DISK;           /* Define data set */

DCL PROG1 PROG                       /* Define program */
      SYSUT2 WRITES TESTDS;          /* to create A.B.C */

RUN PROG1;                           /* Execute it */

LIST TESTDS;                         /* List A.B.C */

IF MAXCC=0 THEN                      /* If all successful, */
         CATALOG TESTDS;             /* Catalog data set */
```

Because all new Data Sets are Deleted after their last use unless a **KEEP** or **CATALOG** instruction is executed for them, if the **CATALOG TESTDS** instruction is not executed, the data set **A.B.C** is deleted.

3 **Return Codes**: When a program has finished executing, it returns a value in register 15 to Jol or the Operating System. This '*return code*' or program result can be tested with an **IF** statement, and, subject to the result of the **IF**, other statements may be skipped or executed. See the **IF** statement description.

4 **Errors**: When a program returns the number 16, or a number greater than 2000, Jol considers this to be an error and will terminate the job at that point *unless* a **STOP WHEN** command has been issued.

An **ABEND** will produce a dump of your program and selected control areas if you have the **SYSUDUMP** or **SYSABEND** files allocated. The **ON ERROR** command will either allocate these files or not, as requested.

5. The *parameter* information, if coded, cannot exceed 100 characters. A long parameter may be coded up to, and including, column 72 and then continued in column 1 of the next card.

   Parameters containing special characters or blanks must be coded in *apostrophes* (single quotes). If any of the special characters are themselves apostrophes, then each must be shown as two consecutive apostrophes. Only one is passed to the program.

   If one of the special characters is a *percent sign* and you are not defining a symbolic parameter, code two consecutive percent signs in its place, for example **PARM ='3462%%05'**. Only one is passed to the program.

   To execute a program with a Parameter of **'PARM'**, the parameter must be coded as

   ```
   PARM = 'PARM';
   ```

   Other examples of Parameters:

   RUN PROG 10; is a simple **RUN** instruction with a simple parameter.

   RUN LINK 'MAP,LIST'; shows a character string as parameter.

   RUN MYPROG 'THIS SHOWS TWO ''S'; shows two apostrophes.

6 When you run the same program more than once in a job, it may be necessary to test the return codes from the **RUNs** individually. To do this, code a label on each of the **RUNs,** and test the label names instead of the program name that you ran.

   For example, suppose program **PARMCC** is a program that merely returns to the operating system whatever was coded in the Parameter field. That is, running:

   ```
   PARMCC PARM 10;
   ```

   will return **10** to the operating system on completion.

   The test for individual runs may be done like this:

```
          DCL PARMCC PROG;
STEPONE: RUN PARMCC 20;
STEPTWO: RUN PARMCC 30;

          IF STEPONE=20
          THEN DISPLAY 'STEPONE RETURNED 30';

          IF STEPTWO=30
          THEN DISPLAY 'STEPTWO RETURNED 30';

          IF STEPONE=15
          & STEPTWO=90
          THEN DISPLAY 'INVALID CODES';
```

```
ELSE DISPLAY 'CODES ARE VALID';
```

7 To re-execute the same program with different data sets, you must make multiple declarations of the program and define the required files in each. The declarations must then be given unique labels and these, not the program names, are **RUN.**

You can reduce the amount of coding necessary by the use of the **LIKE** parameter **-** it may be used to copy details from a previous program declaration.

Example of Running a Program more than once:

PROG1 reads a data set on file **SYSUT1,** and returns a return code of 0 if the file is Fixed Blocked, or a 4 if it is Variable Blocked, otherwise an 8.

```
RUNONE: DCL PROG1 PROG
                SYSUT1 READS DSIDONE;

RUNTWO: DCL PROG1 PROG
                SYSUT1 READS DSIDTWO;

        RUN RUNONE;

        RUN RUNTWO;
```

NB. If you find that a particular program is being executed in this way often, it is a **PRIME** candidate for being a command or **Macro**. This will ensure that it is easy for all users to execute. The section on "Writing New Commands" explains this concept in detail.

Examples of the RUN instruction

1 **RUN ASMFC 'RENT,NOXREF';**

The program definition **ASMFC** is being **RUN,** and any data sets defined in the **ASMFC** program definition will be allocated before control is passed to the program.

In this example, the parameters **RENT** and **NOXREF** are being passed to the Assembler Program.

2
```
 %SORTPARM = 'CORE=MAX';    /* Set Sort Parameter */
RUN SORT '%SORTPARM';
IF SORT ^= 0 THEN
STOP 'SORT FAILED';
```

The symbolic variable **SORTPARM** is set to **'CORE=MAX'**, and then the program declared as **SORT** is **RUN**.

3 **RUN REPORT '%DAY';**

The symbolic variable **DAY** is passed as a parameter to program **REPORT**. **%DAY** is automatically set to **MONDAY, TUESDAY** and so on, thus the program can print the day without extensive calculations.

SAVESYMS - *Immediate Command*

Purpose

The **SAVESYMS** command stores specified symbolic values into a member of a partitioned data set. These symbolic values may be **INCLUDE**d in a subsequent compile, thus providing the facility to communicate with other Jol procedures, or the same procedure at a later time.

SAVESYMS *symbolic1 [,symbolic2 ...]*
IN '*data-set-name(member)*' ;

where

symbolic-variable-name-list	is a symbolic name or a list of names separated by commas, without the **%** preceding them.
data-set-name	specifies the name of an **existing** data set or library into which the data will be saved.

Any **symbolic variable** that has been used during the current compilation may be saved in an external data set. Subsequent runs can reuse the saved or previous values in issuing an **INCLUDE** specifying the member name the symbolics were saved in.

By displaying these values, perhaps in a **PANEL**, the user will automatically know the data that was used for the prior run. The values saved may then be used on a subsequent run to initialize values to those that were set on a previous run.

Notes

1 The data is stored as a series or Symbolic Variable Assignments. Thus:

```
SAVESYMS DAY IN '%SYSUID.JOL(X)';
```

will result in member **X** containing a statement similar to:

```
%X='MONDAY';
```

To retrieve the data, simply **INCLUDE** the member(s).

2 The information can also be used by another job, and so information may be passed from job to job.

3 If subsequent saves are made to the same data set or member, the new data will destroy the data that was previously saved.

To retain the previous data, you must copy it to a different data set, or use a different member name. For example:

```
SAVESYMS RUN, DOLLVAL IN
         '%SYSUID.JOL(%DAY)';
```

will automatically save the data for symbolic variables **RUN** and **DOLLVAL** in the Jol data set, but the member name will change depending on the day the job is used.

Examples of the SAVESYMS command.

1 **SAVESYMS A, B IN '%SYSUID.JOL(LASTRUN)';**

will save the values of **A** and **B**in the users private Jol data set in the member called **LASTRUN**.

To retrieve the data that was saved, issue the following:-

```
INCLUDE LASTRUN;
```

SCRATCH - *Execute Instruction*

Purpose

The **SCRATCH** instruction is used to Scratch data sets. The data contained in the data set is lost, allowing the free space to be used for other data sets.

```
SCRATCH       {  dsid    }        [ (generation-number) ] . . .
              {dsname}

 [FROM [VOL] volume [UNIT] unit]

 [ALWAYS] ;
```

Data sets that the job reads (that is OLD data sets) are normally kept when the job finishes. To Delete a Data Set, the **DELETE** or **SCRATCH** instruction is used.

Notes

1. All **OLD data** sets are automatically kept unless a **DELETE** or **SCRATCH** instruction is executed, but all **NEW** data sets are scratched by the Operating System, unless they are **KEPT** or **CATALOGED**, or the data set has a Retention period (see **RETAIN**) or was **PROTECTED** (see **PROTECT** Command, **PROTECT** and **READ ONLY** Parameters).

2. The '*dsname*' form of the instruction is not recommended for long production jobs as the **DSID** format allows the Data Set Name to be overridden; thus a data set name may be altered in one place **-** the **DECLARE-** and all references to the **DSID** will have the new Data Set Name.

   However, using the '*dsname*' form is a very convenient method for deleting data sets for housekeeping purposes.

3. The **FROM** option, if specified, must be coded after the **DSID** or **DSNAME** list. It is particularly useful for housekeeping of direct access volumes.

4. When the expiration date or retention period has not, or may not have, expired use the **ALWAYS option** to override the specified date and scratch the data set.

5. The **SCRATCH** instruction should be used if it is known that the data set(s) are not cataloged. The **SCRATCH** instruction will not attempt to uncatalog these data set(s) as would the **DELETE** instruction. To use the **DELETE** instruction in this case would produce an unnecessary **ERROR message**.

6. **Temporary Data Sets** or data sets with no name are automatically deleted after the step that used them last.

7. Use the **UNCATALG** instruction to remove a catalog entry from the catalog.

8. If the data set is on tape, the tape will not actually be mounted on the system. If the data set is on a **Direct Access device**, the **VOLUME** must be mounted so that the data set can be **SCRATCHed** from it.

Examples

1. **DCL DSID1 DS MY.DATA.SET;**

```
    SCRATCH DSID1;
```

   The system catalog will be requested to supply volume and device class information from the catalog, and the data set **'MY.DATA.SET'** will be scratched from the volume.

2. **DELETE GENER.DATA.SET(-1)**

```
DELETE    GENER.DATA.SET(-1)
          GENER.DATA.SET(19);
```

   Two generations of the Data Set **'GENER.DATA.SET'** are to be scratched. The catalog will be requested to return the name and volume of the last but one data set in the group, and this will be deleted.

   Absolute generation number 19 will also be deleted.

   If either, or both, the data sets reside on Direct Access Devices, the volume(s) will be mounted (if they are not already) so that the data sets may be physically scratched from the volume; if on tape, the tapes will not be mounted and the instruction is ignored.

3. **DELETE GROUP1.TEST1**

```
DELETE    GROUP1.TEST1
          GROUP2.TEST.MASTER
          GROUP9.ACCOUNT.FILE
          GROUP10.TEST.OUTPUT
    FROM VOL 222222 UNIT 3330-1;
```

   The four data sets will be **SCRATCHed** from volume 222222. If any of the data sets are not on the volume, a warning message will be given but the job will continue. This is an excellent method of scratching data sets from direct access volumes for housekeeping purposes as it requires far fewer control cards than the IBM utility, IEHPROGM.

SEND - *Execute Command*

Purpose

The **SEND** command sends a message to a terminal user.

```
SEND  'text' [ TO ] userid-list  ;
```

Displaying Status Information

The **SEND** command can be used to advise a terminal user about the status of a job, or other conditions.

Notes

1. The *text* must not be greater than 110 characters in length.
2. The message is also displayed on the Job Log.
3. If the user(s) is not logged on, the message is lost.
4. If no *user-id* is specified, the message will be sent to the name specified in the **%SYSUID** symbolic variable.

Example

1 **SEND 'SUBMITTING PAYROLL JOB2 NOW' TO SCHEDULER;**

The user known as **SCHEDULER** will receive the message.

SETUP - *Execute Command*

Purpose

The **SETUP** command displays a message on the Operator's Console to indicate which Volumes (Disk or Tape) are required by the job.

```
SETUP     { VOL   }     { volume        }  ;
          { VOLS  }     { (volume-list) }
```

The **SETUP** command can be used to provide an orderly progression of jobs using the same volumes. Each time the machine is requested to dismount a volume and load another, your job is delayed, and often others as well. Using the **SETUP** comMand, the Operator and Scheduler have an opportunity to optimize the throughput of the machine.

Notes

1. This command generates a **SETUP** card for **HASP** and **JES2** systems.

2. **SETUP** cards may be generated automatically by Jol. Check with the Systems Programming Department.

3. **JES3** or **ASP** systems do not require the **SETUP** details of jobs as they themselves determine the setup requirements and schedule your job accordingly.

Example

1 **SETUP VOLS (TAPE01, DISK99);**

The Operator will be informed that volumes TAPE0l and DISK99 will be required for the job. The job will be put on the **HOLD** queue until it is released by the Operator.

2
```
%VOLUMES = 'D1SK8,222222';
      . . .
SETUP (%VOLUMES);
```

The symbolic variable is set to the required volumes, and then, perhaps after many other Jol instructions, the **SETUP** command is issued. The **SETUP** command will inform the Operator that **DISK8** and **222222** will be required for the job and therefore the job will be placed on the hold queue until the Operator has mounted the required volumes and released the job from the **HOLD** Queue.

SIGNAL ERROR - *Immediate and Execute Instruction*

Purpose

The **SIGNAL ERROR** instruction is used to indicate an error has occurred. If it is used at execution time the message will be written to the Job Log; if performed at compile time, the message will appear with other Jol messages.

SIGNAL [ERROR] *severity*, '*message*'

where:

message	may be any message less than 100 characters long.
severity	indicates the severity of the error. It must be from zero to four (0-4).

When the **SIGNAL ERROR** is executed in the **Compile Phase**, the severity has the following effect:-

Severity	Meaning	Effect
0	**Information only**	Message placed on Jol Compiler Listing File and the Terminal.
1	**Warning**	Message placed on Jol Compiler Listing File and the Terminal.
2	**Minor Error**	Message placed on Jol Compiler Listing File and the Terminal.
3	**Severe Error**	Message placed on Jol Compiler Listing File and the Terminal. The job is placed in the **HOLD** status; the operator must release it.
4	**Terminal Error**	Message placed on Jol Compiler Listing File and the Terminal. The job is not allowed to start unless the **LET** compiler option was specified; same as using **STOP** instruction.

When executed at **run time** (that is, after an **IF** statement), the severity has the following effect:-

Severity	Meaning	Effect
0	**Information**	Message placed on Job Log only; similar to **DISPLAY**.
1	**Warning**	Message placed on Job Log; similar to **DISPLAY**.
2	**Minor Error**	Message placed on Job Log and written to Operator; similar to **TYPE**.
3	**Severe Error**	Message placed on Job Log and written to Operator; Operator asked to reply **YES** if job is to continue, **NO** if it is to be stopped.
4	**Terminal Error**	Message placed on Job Log and written to Operator; Job stops immediately; **IF ERROR** not actioned; similar to **STOP**.

Notes

1. The message must be less than 100 characters long.
2. If there are any apostrophes, two apostrophes must be coded; only one will appear on the produced message.
3. The severity must be from 0 to 4.
4. If the **SIGNAL ERROR** terminates a job, any **IF ERROR** instructions are ignored **-** the job stops immediately.
5. Jol automatically stops a job when any program returns a 16 or greater than 2000. Thus, whenever a program returns a 16 or greater than 2000 the job is terminated. In **PL/1** the **EXIT** or **STOP** instructions stop the program; these return 2000 or greater to Jol, which terminates the entire job.

   The **STOP WHEN instruction** may be used to alter the default of **16** or **>** 2000.

Examples of the SIGNAL ERROR Instruction

1. **SIGNAL ERROR 1, 'SIMPLE EXAMPLE OF SIGNAL ERROR';**

   After executing the **SIGNAL ERROR**, the message **'SIMPLE EXAMPLE OF SIGNAL ERROR'** will be placed in the Jol Compiler Listing. Because the severity specified was one, the job will be allowed to start. The return code from the compiler will be 4.

2.
   ```
   IF %LIB = 'SYS1.SVCLIB'
   THEN SIGNAL ERROR 3,
           'SYS1.SVCLIB IS TO BE RE-ORGANIZED, RELEASE WHEN
           MACHINE IDLE';
   ```

   The **IF** statement tests the symbolic variable **%LIB**, and if **%LIB** contains the characters **'SYS1.SVCLIB'** the **SIGNAL ERROR** instruction will be executed, and the message will be placed on the error log.

   In addition, the job will be placed in the **HOLD** status and the operator will have to release the job manually.

Note that the above message may be written to the operator at execution time, if desired. The **SIGNAL ERROR** must be preceded by an execution oriented **IF**.

SHOWDIR - *Immediate Instruction*

Purpose

The **SHOWDIR** instruction displays a sorted list of directory entries, commencing from the current directory level. The cursor may be used to select a directory. The current path is changed to the directory chosen.

```
SHOWDIR  ;
```

Notes

1. **SHOWDIR** provides a convenient method of moving between directories. After a **SHOWDIR** instruction, **%SYSDIR** contains the name of the current directory.

2. The **GETDIR** instruction may be used to find the current directory. After a **GETDIR** instruction, **%SYSDIR** contains the name of the current directory.

Example

1 **SHOWDIR;**
WRITE 'Current Directory is %sysdir';

The **SHOWDIR** instruction displays the sorted list of directories. The User then selects a directory from the list, and the **WRITE** instruction displays the selected directory.

SHOWDSN - *Immediate Instruction*

Purpose

The **SHOWDSN** displays a sorted list of data set names from the current directory. The cursor may be used to select either a data set or a directory. If a directory name is chosen, a temporary change is made to that directory, and data set or file names displayed from that path. This process continues until a data set name is selected.

SHOWDSN *[path\file-search-pattern]* **;**

Notes

1. **SHOWDSN** provides a convenient method of selecting data sets or file names. After a data set name has been selected, symbolic variables contain the name and other information.

2. If the optional *pathname\file-search-pattern* is not specified, the default is ***.***, or all files are displayed.

3. The **GETFIRST** and **GETNEXT** instructions may also be used to select data sets.

   Coding **GETFIRST *.*** ; returns the first file name in the current directory. Coding **GETNEXT** ; then returns the next file name. **%LASTCC** is set non-zero when there are no more file names.

4. The following symbolic variables are set when **SHOWDIR, GETFIRST** or **GETNEXT** instructions are used.

%SYSDIR	contains the current directory.
%SYSDSN	contains the *data-set-name*. All blanks are removed.
%SYSDSN2	contains the *data-set-name*. No blanks are removed and therefore it is a simple matter to parse the file name into *filename, file-type*.
%SYSDSN3	contains the *data-set-name, filesize, date* and *time* information. For ease of sorting, the *date* is in YYYY-MM-DD format.

Note: If the data set is changed, the word **CHANGED** is added after the time.

Example

1 **SHOWDSN *.PLI;**

```
X=INDEX(SYSDSN3,'CHANGED');
IF X>10
THEN WRITE 'File %SYSDSN Has Been Modified';
```

The **SHOWDSN** instruction displays a sorted list of **PLI** data set names. After a data set has been selected from the list, the **WRITE** instruction displays a message if the file has been modified.

SORT - *Execute Command*

Purpose

The **SORT** command allows up to 5 data sets to be **SORT**ed into another data set using parameters either described in the **FIELDS** parameter or the parameters described in the data set referenced by the **USING** parameter.

```
        SORT    {  dsid     }  [ ||   { dsid     }    ...]
                { dsname  }           { dsname  }

                TO  {  dsid  |  dsname  }

                {  FIELDS  = ( sort parameters) [FORMAT=f]      }
                {  USING  { dsid  | dsname  }                   }

                {  number TAPES                            }
                {  number DISKS [number  CYLS]             }

                OPTIONS | OPTS  = 'options'

            ;

    where:

            FIELDS = (p1,m1,f1,s1, . . . pn,mn,fn,sn)
```

Notes

1. If no **TAPES**, **DISKS** or **CYLS** are specified, the standard Jol default of 3 disk work areas of 5 cylinders is used. The defaults are installation-dependent and therefore may vary from installation to installation.

2. If ***n*** **TAPES** are specified a Tape Sort is generated using the number of tapes specified as Sort Work Tapes.

3. If ***n*** **CYLS** are specified the size of the disk work areas are altered to the specified amount.

4. If ***n*** **DISKS** are specified the number of work areas used is as specified.

5. The maximum number of input data sets that may be sorted is *five* (5) and all input data sets must be on like devices; that is, one data set may not be on a Tape and another on a Disk.

6. The **USING** parameter specifies that the data set referred to contains the **SORT** parameters. The data set may have been created in the current job, or it may be an old data set.

   The parameters must be in the same format as required by the **SORT**.

7. The **OPTIONS** parameter can contain any valid extra sort parameter information, such as **'EQUALS'**. This information will be added to the generated **SORT** parameter field.

8. If the **SORT** is not successful, a non-zero return code will be issued. You can use an **IF** to check for this condition.

9. The **FIELDS** parameter specifies the **FIELDS** to be used for sorting, collating and merging the input data sets. The format is:

   The format is:

   FIELDS = ($p_1,m_1,f_1,s_1, \ldots p_n,m_n,f_n,s_n$)

   where

P	specifies the *position* within the record.
M	specifies the *length* of the field.
F	is the *format* of the field and may be **CH | ZD | ID | FI | BI | FL | AC | CSL | CST | CLO | CTO | ASL | AST | AQ**
S	specifies the desired *sequencing* **A | D | E**

Examples

1
```
DCL DSID2 DS PAYROLL.SORTED.TRANS
             VB 300, 8000
             10 CYLS DISK;

SORT         PAYROLL.TRANS.FILE
             TO DSID2
             FIELDS=(
                      10,2,CH,A,      /* COMPANY CODE */
                      25,5,FI,A);     /* STATE CODE */
```

The data set referred to by **DSID1** is to be sorted and placed in the data set referred to by **DSID2**. The parameters for the **SORT** are specified using the **FIELDS** parameter.

2
```
SORT A TO B
             50 CYLS 10 DISKS
             USING SORT.CONTROL(S001);
```

The data set referred to by **DSID A** is to be sorted and placed in the data set referred to by **DSID B**. The **SORT FIELDS** are to be read from member **S001** of the **SORT.CONTROL** data set.

Ten **DISK** work areas of fifty cylinders each has been requested for the **SORT** to use as work areas.

3
```
DCL DSID1 DS PAYROLL.TRANS.FILE;
DCL DSID2 DS PAYROLL.SORTED.TRANS
                      VB 300, 8000
                      10 CYLS DISK;
SORT DSID1 TO DSID2
            FIELDS=(
             10,2,CH,A,               /* COMPANY CODE */
             25,5,FI,A);              /* STATE CODE */
```

The data set referred to by **DSID1** is to be sorted and placed in the data set referred to by **DSID2**. The parameters for the **SORT** are specified using the **FIELDS** parameter.

STARTAT - *Execute Instruction*

Purpose

The **STARTAT** instruction causes all executable instructions to be skipped from that point until an executable instruction with the named label is found. It is similar to a **GOTO**.

STARTAT	*label* ;
where	
label	is the name of an *executable* instruction where execution is to recommence.

You may wish to skip steps in a job because:

- You are restarting the job at some point other than the beginning.
- You have a long job containing steps that are normally run monthly (for example), and you do not want to run all the program steps on a daily basis.
- You may be testing an individual program in a job, and the **STARTAT** and **STOPAT** instructions will allow you to execute an isolated segment of your Jol program. The **STARTAT** instruction can be used to skip over all the steps you do not want to execute.

Notes

1. The **STARTAT** instruction assists **RESTART** situations.
2. Only **IF**, **DO**, **END**, and **DECLARE** statements are actioned after a **STARTAT** instruction; however all other instructions are checked for syntax.
3. The **STARTAT** applies to the Compiler Phase and Execute Phase **-** all Preprocessor Statements are executed as usual.
4. The **STARTAT** instruction need not be the first instruction in a Jol program. You may run programs, then issue a **STARTAT** to skip steps, and continue processing later. If you do this the **STARTAT** is more like a **GO TO** instruction.
5. Multiple **STARTATS** may be issued, but each must be satisfied before any subsequent ones are issued.
6. The **STOPAT** instruction can also be used in combination with the **STARTAT** to execute a small segment of your program.
7. The **STARTAT** instruction is a directive to the **Compiler**to start processing Jol instructions when the statement with the label is found. Because it is a compiler directive, you cannot issue a **STARTAT** after an ***execution*** oriented **IF** statement. If you do so the **STARTAT** will be actioned as though no **IF** had been coded.

Examples

1
```
STARTAT STEP9;
        . . .

STEP9:          RUN SALES;
```

All steps after the **STARTAT** instruction are skipped until the labeled instruction with **STEP9** as the label is found.

2
```
IF %RESTART = 'REPORT'
THEN DO;
        DCL MASTER DS SALES.MASTER(0);
        STARTAT REPORT;
END;

ELSE DCL MASTER DS SALES.MASTER(+1)
                VB 200,7294
                3380
                10 CYLS;
        . . .

UPDATE:         RUN UPDATE;

                CATLG MASTER;

REPORT:         RUN REPORT;
```

The above example shows that if **%RESTART** contains **'REPORT'** then the **MASTER DSID** is to be declared, and it is to use the latest **(0)** generation of the group of data sets called **SALES.MASTER**. Additionally, the **STARTAT REPORT** instruction is issued, and so processing will then be skipped until the label **REPORT** is found.

If **%RESTART** is not **REPORT**, then the **MASTER DSID** is declared as a new generation. Because the **STARTAT** instruction will not be executed, the **RUN UPDATE** and **CATALOG MASTER** instructions will be executed.

3.
```
%RESTART = 'STEP10';
        ...

IF %RESTART = 'STEP10'
THEN DO;
        RUN RESTORE;            /* RESTORE DATA SET */
        STARTAT %RESTART;
END;
```

In the example above the symbolic **%RESTART** is set to **STEP10**. Later, an **IF** statement is used to test the value of the symbolic variable. In this case, if it is **STEP10** then the program **RESTORE** will be run to restore data set, and then the **STARTAT** is issued with the symbolic variable **%RESTART** as its subject. The **%RESTART** is replaced by the current contents of **%RESTART**, and so the **STARTAT %RESTART** becomes **STARTAT STEP10**.

STOP - *Immediate and Execute Instruction*

Purpose

The **STOP** instruction is used to terminate a job immediately and place a message on the Job's Log. The message is also displayed on the **Operator's Console**.

After execution of the **STOP** instruction, the remainder of the job is flushed, and no further processing is possible. The **IF ERROR** does not receive control.

In *execute mode*, the job is **terminated immediately**. In *compiler mode*, the job is not allowed to start execution.

```
STOP   number | 'message'          ;
```

Notes

1. The message must be less than 100 characters.

2. If the **STOP** is executed after an **execution IF** statement, the job will be terminated at that point. For example:

   IF SORT = 16 THEN STOP 'SORT RETURNED 16';

3. If the **STOP** is executed **unconditionally** or after a **preprocessor IF**, the message will be output as a Jol Compiler message. The job will not be allowed to commence execution unless the compiler option **LET** has been specified. This can save valuable machine resources. For example:

   ```
   IF %NUMBER<8 | %NUMBER>35
       THEN STOP '%NUMBER IS OUT OF RANGE:-        RESUBMIT
   WITH CORRECT VALUE';
   ```

   If **%NUMBER** is **not 8-35**, the job is not allowed to start.

4. Jol automatically terminates a job when any return code of 16 or greater than 2000 is detected. Thus, whenever a program returns a code of 16 or more than 2000, the job is terminated. The **STOP WHEN** instruction may be used to alter the default of 16 or greater than 2000.

   In **PL1**, the **EXIT** or **STOP** instructions will terminate a program and return greater than 2000, thus terminating the job.

5. The **IF ERROR** is not actioned if a job is terminated by a **STOP** instruction.

Examples

1 **STOP 'SIMPLE EXAMPLE OF STOP';**

After executing the **STOP**, the message **'SIMPLE EXAMPLE OF STOP'** will be displayed with the Jol Compiler **Error Messages**(if any); the job will not be allowed to start to execute unless the compiler option **LET** has been specified.

2 **RUN SORT 'CORE=MAX';**

```
IF SORT ^= 0
THEN STOP 'SORT FAILED-CHECK AND RESUBMIT';
```

The program or step **SORT** is **RUN** with a parameter of **'CORE=MAX'**. If it fails for any reason other than an **ABEND**, it will return a non-zero, and the instruction after the **IF** instruction will be executed; this is a **STOP** and so the message **'SORT FAILED-CHECK AND RESUBMIT'** will be displayed on the Operator's Console and the Job Log, and the job will be terminated.

3

```
RUN VALIDATE;
IF VALIDATE = 0
THEN DO;
    CATALOG NEWTRANS;
    SUBMIT UPDATE;
END;

ELSE STOP 'ERROR IN VALIDATE JOB, CORRECT AND RESUBMIT';
```

The **VALIDATE** program is executed; if it runs without errors, the data set **NEWTRANS** is cataloged and the next job **UPDATE** is submitted for execution.

However, if **VALIDATE** fails, then a message is given to the operator requesting him to make corrections and then resubmit the job.

4

```
IF %DAY='SATURDAY' | %DAY = 'SUNDAY'
THEN
STOP 'THIS JOB DOES NOT RUN ON %DAY';
```

The Jol statements above simply test the initialized special variable **%DAY**; if it is **SATURDAY** or **SUNDAY**, the **STOP** instruction is executed, thus prohibiting the job ever starting execution.

If it is not a Saturday or Sunday, the **STOP** will not be interpreted, and, if no other errors were found, the job would be submitted to the system for execution.

STOPAT - *Execute Instruction*

Purpose

The **STOPAT** instruction causes all executable instructions at and following the label not to be executed. Execution ceases at this point.

> **STOPAT** *label* ;
>
> *where*
>
> *label* is the name of an *executable* instruction where execution is to stop.

Stopping the Job

You may stop a Jol job by using:

- A **RETURN** instruction.
- A **STOP** instruction.
- A **STOPAT** instruction.
- A **STOP WHEN** instruction.
- A **STOPAT** instruction.
- Or by reaching the end of the job.

The **STOPAT** instruction is a **Compiler Directive** instruction, and after the specified label is found, that instruction, and all others following, are ignored.

Notes

1. The **STOPAT** instruction is a **Compiler Directive** instruction, and after the specified label is found, that instruction, and all others following, are ignored.

2. Because the **STOPAT** is a compiler instruction, you cannot code it after an **Execute** time instruction. You can code it after an Immediate or Preprocessor oriented **IF** statement.

Examples

1 **STOPAT COPY**;

The job will terminate at (before) the **COPY** instruction.

2
```
IF %RESTART = 'STEP9'
THEN DO;
      STARTAT RECOVER;
      STOPAT STEP10;

END;
```

If the symbolic variable contains the value **STEP9** then the **STARTAT** and **STOPAT** instructions will be executed.

STOP WHEN - *Execute Instruction*

Purpose

The **STOP WHEN** instruction is used to terminate the job when certain return codes, or ranges of return codes, are issued.

STOP WHEN *condition* ***[| condition...]*** ;

where *condition* is:

ANY c*ondition-operator* *number* from **0** to **4095**

and c*ondition-operator* and meaning is one of the following:

=	equal to
⌉=	not equal to
^=	not equal to
<	less than
<=	less than or equal to
⌉<	not less than
^<	not less than
>	greater than
>=	greater than or equal to
⌉>	not greater than
^>	not greater than

1.

Notes

2. You may test the return code after every instruction, if you wish, and determine whether the job should continue or stop. Rather than do this repeatedly, the **STOP WHEN** instruction automatically tests a set of specified conditions after every **RUN** instruction or command that uses the **RUN** instruction such as **COPY** or **SORT**.

3. The Jol default is:

   STOP WHEN ANY=16 | ANY>2000;

   Thus whenever a program returns a code of 16 or more than 2000, the job is terminated. In PL1, the **EXIT** or **STOP** instructions will terminate a program and return greater than 2000, thus terminating the job.

4. You may not execute a **STOP WHEN** instruction after an **IF test;** use the **STOP** after **IF** tests.

5. The maximum number of tests you may request is **eight**.

6. You may code one **STOP WHEN** instruction per Jol job.

7. The **IF ERROR** is not actioned if a job is terminated by a **STOP WHEN** instruction. No further processing may take place.

Examples

1 **STOP WHEN ANY = 10;**

When any executed program returns a value of 10, the job is terminated.

2 **STOP WHEN ANY <=20 | ANY >=50;**

If any return codes less than 20 or greater than 50 are returned, the job terminates. That is, the only valid return codes are within the range 20 to 50 inclusive.

STORPROC - *Execute Instruction*

Purpose

The **STORPRO** Ccommand stores procedures or control cards in a partitioned data set or sequential data set. The **STORPRO** Ccommand can be used in a batch environment where it is impossible or undesirable to save procedures into a data set with TSO or SPF.

> **STORPROC** ***dsid* | *dsname*** IN ***dsid* | *dsname*** **;**

Notes

1. This command uses the **IEBUPDTE** utility to copy the required data to the data set.

2. Any **IEBUPDTE** control cards may be used by enclosing them in quotes.

3. Jol enques the data set using the same enque names as **SPF**. Therefore you can save data in a data set even if it is currently being used by a User under TSO/SPF.

4. See also the **UPDATE** command.

5. You can also use the **COPY** command to copy data into a data set. However, the **COPY** will not number the data for you.

Examples

1 **STORPROC MY.DATASET IN PROD.DATASET;**

The card image file in **MY.DATASET** is copied to **PROD.DATASET** using **IEBUPDTE**. Any control cards for **IEBUPDTE** are actioned.

2 **STORPROC MY.PDS(MEMB1) IN PROD.PDS(MEMB1);**

The card image file in member **MEMB1** of the partitioned data set or library **MY.PDS** is copied to the corresponding member in **PROD.PDS** using **IEBUPDTE**. Any control cards for **IEBUPDTE** are actioned.

3 **STORPROC** '*data ...* '
'*data ...* '
'*data ...* ' **IN SYS2.CONTROL(MEMB10);**

The card images are copied to **SYS2.CONTROL** and placed in member **MEMB10**.

SUBMIT - *Execute Command*

Purpose

The **SUBMIT** command allows the compiling of subsequent jobs at any point within the currently executing job. This provides the facility to effectively schedule dependent job streams. That is, the dependent job is not compiled or placed on the system job queue until the current job has reached a point where it is desirable or safe to run the dependent job.

By using this technique, a linkage can be created which controls the sequence of running a number of jobs.

Note: Symbolic Parameters may be passed from job to job.

```
                 {  member-name [ , member-name ...]  }
SUBMIT           {  USING   dsid | data-set           }

                 [SYMS = 'symbolic-variable-name-list']

                 {   LC            }
                 {   LINECOUNT     }    [=]  line-count

                 [PI]     [NPI]
                 [PM]     [NPM]
                 [PE]     [NPE]
                 [PJ]     [NPJ]
                 [LET]    [NOLET]
                 [PO]     [NPO]

  ;
```

The **SUBMIT** command may also be used to submit **JCL** code direct to the system by including the desired JCL code within quotes.

```
SUBMIT 'JCL code ....';
```

The **SUBMIT** command is used to transmit a new job to the Operating System, ready for execution.

Notes

1. The job may be a Jol job, or a **JCL** job. If the job is a Jol job, the Jol compiler is called to convert the program to instructions ready for the machine to execute. If the job is a **JCL** job, the card images are passed through to the Operating System unchanged, apart from the replacement of Jol symbolic variables.

2. The *member-name-list* must consist of one or more members separated by blanks or commas. This member (or members) is directed to be **INCLUDED** as part of the compiler input. Up to five (5) members may be specified. Each member must contain its own **JOB** Definition statements.

The users **%SYSUID.JOL** data set is automatically concatenated to the Jol input data sets.

3. The **USING** parameter allows a section *data-set-identifier* to be specified that contains the Jol program code to be input into the Jol compiler. The Jol program in the data set may be generated by an earlier program in the job stream, or it may be any other card image data set containing Jol statements.

4. If neither the **USING** parameter or the *member-name-list* is specified, the input is assumed to be a data-set-identifier named **JOLIN**.

5. The **SYMS** parameters allows the passing of Symbolic Variables from job to job. Initial values are assigned to Symbolic Variables, in the same manner as the *** JOL card**, or by using the parameter to the Jol compiler as described in the section dealing with Compiler Options.

   Thus the **first** job in a network may have values assigned to Symbolic Variables, and these values may then be passed to all the other jobs in the network without Operator or Programmer intervention.

   For example:

   JOB1 contains a **SUBMIT** command requesting **JOB2** to be compiled and placed on the system queue ready to execute. **JOB1** had a symbolic variable initialized and **JOB2** also requires that the same symbolic variable be initialized to the value contained in **JOB1**.

   ```
   %SYM='VALUE';
   SUBMIT JOB2 SYMS='SYM=%SYM';
   ```

   The current value of **%SYM** will be passed to **JOB2** when the **SUBMIT** command is executed.

6. The **PRINTOPT**, **LINECNT** and following parameters may be used to set conditions for the Jol compiler. They are described under the section on Compiler Options.

7. Symbolic Variables or Parameters may also be coded in the JCL submitted when the JCL format of the command is used.

   These will be resolved before the job is submitted.

   In fact, the entire JCL statement itself may be a symbolic parameter, thus enabling the generation of highly tailored JCL.

   Up to ten JCL statements may be submitted with one **SUBMIT** command. Multiple **SUBMIT** commands are possible.

Examples

1 **SUBMIT JOB1, JOB2; /* PUT JOBS ON SYSTEM QUEUE */**

Members **JOB1** and **JOB2** are translated by Jol into executable instructions, and submitted to the Operating System for execution.

```
IF MAXCC<8
        THEN SUBMIT JOB10;
        ELSE SIGNAL ERROR 3, 'JOB10 NOT ON QUEUE,
        CRITICAL ERROR DETECTED';
```

This example describes a typical **SUBMIT command.**

If the maximum completion code from all the programs that have executed in the jobs is less than 8, then **SUBMIT JOB10,** otherwise display a message on the Operator's Console for information.

2
```
%DATASET1='%SYSUID.INPUT';
%DATASET2='%SYSUID.OUTPUT';
. . .
SUBMIT      '//JOB1 JOB (account etc)'
            '//     EXEC PGM=IEBCOPY'
            '//IN1  DD   DSN=%DATASET1,DISP=SHR'
            '//OUT  DD   DSN=%DATASET2,DISP=(,CATLG),'
            ' etc ...';
```

The Symbolic Parameters are replaced before the job is submitted.

TEXIST - *Execute Command*

Purpose

The **TEXIST** command tests if a data set exists. Based on the return code of the **TEXIST** command, you can take appropriate action, for example, recover a data set or submit a job.

TEXIST ***dsid* | *data-set-name* } ;**

Testing for the Presence or Absence of a Data Set

It is often desirable to know if a data set exists, or not. Frequently, a job needs to take special action if a particular data set exists. For example, if data set **X.Y** exists you could:

- Submit a job.
- Enter a restart procedure.

Notes

1. You can code a label on the TEXIST command, or use LASTCC to test the return code.

   The return code are:

0	If the data set exists.
4	If the data set does not exist.

1. The data set is allocated as a MOD data set, and an internal operating system indicator is then tested to see if the data set was allocated as a new data set, or an old data set. If it was allocated as a new data set, the return code is set to 4, otherwise 0.

Examples

1

```
DCL OLDMAST DS PAYROLL.UPDATES;
TEXIST OLDMAST;

IF LASTCC = 0 THEN SUBMIT JOB6;
ELSE TYPE 'JOB6 NOT SUBMITED';
```

The DSID **OLDMAST** (data set **PAYROLL.UPDATES**) is checked to see if it exists. If it does exist, **JOB6** is submitted. **JOB6** presumably uses the data contained in **PAYROLL.UPDATES**.

2

```
TEST:  TEXIST 'PAYROLL.UPDATES';

IF TEST = 0 THEN SUBMIT JOB6;
ELSE TYPE 'JOB6 NOT SUBMITTED';
```

This example is the same as the one above, except it is coded in a different manner. If data set **PAYROLL.UPDATES** exists **JOB6** is submitted, otherwise a message is displayed on the Operator's console.

TYPE - *Execute Instruction*

Purpose

The TYPE Instruction types a message on the operators console, and on the job's log.

```
TYPE   { number   |   'message'   } ;
```

The **TYPE** instruction is most often used to inform Operators about certain situations or conditions. It can be used, for example, to inform the Operator that a particular job will not execute for some reason, or that a critical error occurred in the job,and it will be terminating before reaching the normal end of job.

Note

The message must be a character string constant or number not greater than 100 characters in length.

Examples

1 **TYPE 10;**

The message:

JOLE10-01 '10'

will appear on the Operator's Console and the job's system log.

2 **RUN VALIDATE;**

```
IF VALIDATE = 0 THEN
DO;
      CATALOG NEWTRANS;
      SUBMIT UPDATE;
END;

ELSE TYPE 'ERROR IN VALIDATE JOB, CORRECT AND
RESUBMIT';
```

The **VALIDATE** program is executed; if it runs without errors, the data set **NEWTRANS** is cataloged and the next job **UPDATE** is submitted for execution.

However, if **VALIDATE** fails, then a message is given to the operator requesting him to make corrections and then resubmit the job.

UNCATALOG - *Execute Instruction*

Purpose

UNCATALOGing a data set removes the name of the data set from the catalog. It does not **SCRATCH** the data set, and so if you want to use the data set again, you must code the volume and unit information. You may use the **CATALOG** instruction to place data set name and volume information in the catalog.

{ UNCATALOG | UNCATLG | UNCAT}

{ *dsid* | *dsname* **} [** *(generation-number)* **] . . .**

[FROM [VOL] volume [UNIT] unit]

Notes

1. See page 33 for an overview of the **CATALOG, DELETE, KEEP, SCRATCH** and **UNCATALOG** instructions.

2. The *data-set-name form* allows you to **UNCATALOG** data sets easily, but use of the data-set-identifier (**DSID**) form permits you to change the data set name in only one place (the **DECLARE**) and still have the new name uncataloged without any further changes to your Jol program.

3. However, using the '*dsname'* form is a very convenient method for uncataloging data sets for housekeeping purposes.

4. If the data set(s) is not cataloged, information messages will indicate this, but the job will still run.

5. If you know a data set is **SCRATCH**ed (i.e., does not exist), the **UNCATALOG** instruction may be used to remove the catalog entry rather than the **DELETE** instruction.

6. Otherwise, use the **DELETE** instruction rather than the **UNCATALOG** instruction. **DELETE** will issue an information message if the data set has already been scratched, but will still perform the **UNCATLOG** function thus keeping the catalog contents correct.

7. The volumes containing the data sets will not be mounted to perform the **UNCATALOG** instruction.

8. Some Tape Library systems use the catalog to determine if a tape may be overwritten or placed in the scratch pool. Thus uncataloging a tape data set may be equivalent to deleting the data set, that is, the data on the tape may be overwritten.ô

9. When qualified data set names (for example **PAYROLL.MASTER**) are uncataloged, unnecessary index levels are automatically deleted from the catalog.

Examples

1 **DCL DSID1 DS MY.DATA.SET;**

UNCATALOG DSID1;

The system catalog will be requested to supply volume and device class information from the catalog, and, if it was in the catalog, the name will be removed from the system catalog with the **UNCATALOG** instruction.

2

```
UNCATALOG GENER.DATA.SET(-1)
          GENER.DATA.SET(19);
```

Two generations of the Data Set **'GENER.DATA.SET'** are to be uncataloged. The catalog will be requested to return the name and volume of the last but one data set in the group, and this will be uncataloged. Absolute generation 19 will also be uncataloged.

3.

```
UNCATALOG GROUP1.TEST1
          GROUP2.TEST.MASTER
          GROUP9.ACCOUNT.FILE
          GROUP10.TEST.OUTPUT

               FROM VOL DA992 UNIT 3330-1;
```

The four data sets will be **UNCATALOGED**, but not **SCRATCHED** from volume **DA992**. No check is made to see if the data sets actually exist on the volume. If any of the data set names are not in the catalog, a warning message will be given but the job will continue. This is an excellent method of uncataloging many data sets for housekeeping purposes, as it requires far fewer control cards than the IBM utility, **IEHPROGM**. To perform this action in JCL is not possible, because if one, or more, of the data sets is missing, none of the others will be uncataloged, and the job will be terminated immediately.

UPDATE - *Execute Command*

Purpose

The UPDATE command updates card image data sets in partitioned or sequential data sets. UPDATE can be used in a batch environment where it is impossible or undesirable to update data sets with TSO or SPF.

> **UPDATE { *dsid* | *dsname* } WITH { *dsid* | *dsname* };**
>
> *or*
>
> **UPDATE { *dsid* | *dsname* } WITH '*data* ' ['*data*'] ... ;**

Notes

1. This command uses the **IEBUPDTE** utility to update the data set.
2. Any **IEBUPDTE** control cards may be used by enclosing them in quotes.
3. Jol enques the data set using the same enque names as SPF. Therefore you can save data in a data set even if it is currently being used by a User under TSO/SPF.
4. See also the **STORPROC** command.

Examples

1 UPDATE MY.DATASET WITH NEW.DATASET;

The card image file in **MY.DATASET** is updated with the update instructions found in **NEW.DATASET** using the system utility **IEBUPDTE**. Any control cards for **IEBUPDTE** are actioned.

2 UPDATE MY.PDS(MEMB1) WITH PROD.PDS(MEMB1);

The card image file in member **MEMB1** of the partitioned data set or library **MY.PDS** is updated by control statements found in member **MEMB1** of data set **PROD.PDS**. Any control cards for **IEBUPDTE** are actioned.

3
```
UPDATE    MY.DATASET WITH
     'data ... '
     'data ... '
     'data ... '  ;
```

MY.DATASET is updated as indicated in the data statements.

VALIDATE *- Immediate Instruction*

Purpose

The **VALIDATE** command tests a symbolic variable to be a specific value or within a specific range. It is typically used in conjunction with the Jol **PANEL** instruction to check the values entered by a user. Should the tests fail, an error message is displayed to the user showing the field in error, and the required input.

The user then has the opportunity to correct the input.

VALIDATE *symbolic-variable-name*
test1 **[/] [***test2***] ;**

where

symbolic-variable-name
is a symbolic name, without the % preceding it.

test1	specifies the characters the variable is to be validated with. This can also be used to test one or more characters by using the / symbol.
test2	specifies the range when using *test1*.

Examples

1. **VALIDATE DAY MONDAY;**

   will test that %DAY is equal to 'MONDAY'.

2. **VALIDATE YEAR 198/;**

   will test that **%YEAR** begins with 198.

3. **VALIDATE MONTH 6 12;**

   will test that %MONTH is a number in a range between 6 and 12.

4. **VALIDATE ACCOUNT 600/ 800/;**

   will test to see that the first three characters of %ACCOUNT is within the range 600 to 800**.**

WRITE - *Immediate Instruction*

Purpose

The **WRITE** Instruction is used to write a message to a terminal when running Jol interactively, or to write data to a file previously opened with the **OPENFILE** instruction. Records are written one at a time from the symbolic variable to the terminal or tothe named file.

> **WRITE** **'***message***'** **[AT** *row,column* **] ;**
>
> *or*
>
> **WRITE FILE (***filename***) FROM (***symbolic variable***);**
>
> *or*
>
> **WRITE FILE (***filename***) FROM ('***literal string***');**

General Notes

1. The maximum text length that may be written is 253 characters.

2. The names do not have the **%** symbol preceding them; this allows a **WRITE** instruction to be coded with a symbolic variable containing the name of another symbolic variable **-** after replacement the correct symbolic variable will have the value written to the terminal or the file.

Specific Notes for TERMINAL Output

1. A **WRITE** Instruction is usually used to write a message to an interactive terminal; a **READ** Instruction is then used to accept data back from the user. Decisions can then be made on that data.

Specific Notes for FILE Output

1. The file must have been previously **OPEN**ed with the Jol **OPENFILE** instruction.
2. **%LASTCC** is set to zero if a record was written successfully, otherwise it is set non-zero.
3. The **PUTFILE** instruction may also be used to write data to a file.
4. The file will only contain character information. Binary fixed, decimal, and floating point numbers are not yet supported.
5. If the data is enclosed in quotes (**TYPE = 'LIT'**), the quotes are removed before output.
6. If the Record Format of the file is Variable or Variable Blocked, the record length is set equal to the length of the symbolic variable.

   For Fixed length record files, the symbolic variable is padded with blanks if the length is less than the file's record or length, or truncated if it is too long to be contained in one record.
7. The GOTO instruction may be used to copy the entire contents of the file. Files may be copied with a combination of READ, WRITE and REDO Instructions, with processing on the contents of the file being performed.
8. See also the **ALLOC, OPENFILE, READ, GETFILE, PUTFILE, CLOSFILE** and **FREE** Instructions.

Examples of Output to a Terminal

1 **WRITE 'THIS TERMINAL IS LOGGED ON AS %SYSUID';**

If the terminal is logged as as '**SYSMAINT'** then the message:

'THIS TERMINAL IS LOGGED ON AS SYSMAINT'

will be displayed**.**

2

```
IF %SOURCE = ''
THEN DO;
      WRITE 'ENTER SOURCE STATEMENT LIBRARY';
      READ SOURCE;
END;
```

If the symbolic parameter **SOURCE** is ***null*** (that is, empty) or blank, then the **WRITE** Instruction will write a message to the terminal. The **READ** Instruction will then wait until the user has entered the data and store the data into the symbolic variable **SOURCE.**

Example of Output to a File

1.

```
OPENFILE FILEOUT OUTPUT;   /* Open the file for OUTPUT */
%VAR = 'THIS IS DATA TO BE WRITTEN';
WRITE FILE(FILEOUT) FROM(VAR);
```

The data from the symbolic variable **VAR** is written to the file **FILEOUT.**

2.

```
OPENFILE FILEOUT OUTPUT;   /* Open the file for OUTPUT */
WRITE FILE(FILEOUT)
      FROM('THIS IS DATA WRITTEN ON %DAY');
```

The data from the literal data in quotes. Because **%DAY** is included in the string, it is changed to the current day before the data is written to the file **FILEOUT.**