Concepts and Facilities

Jol

Universal Command Language

Version 5.1

CONCEPTS AND FACILITIES MANUAL

Open System Command and Retrieval (OSCAR) Pty. Ltd.

Jol Command Language "*Concepts and Facilities Manual".*

Sixth Edition (November, 1999)

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

Open System Command and Retrieval (OSCAR),
16/38 Kings Park Road,
West Perth
Australia 6005.

Telephone **(61)-8-9324-1119**
Email **oscarptyltd@ozemail.com.au**

PREFACE

This Concepts and Facilities Manual presents a general overview of Jol - The Universal Command Language. It details Jol's operation and functions, and some of its more outstanding features. Jol is fully documented in the **Jol Reference Manuals** and **User Guides.**

This manual is aimed more for Programming Management than for Technical Programming staff. Due to the very nature of Jol, there will be a number of sections in this manual that become very technical, especially the sample Jol jobs. The overall readability of this manual will not be affected by these sections. However, they are necessary in providing a more complete picture of the Command Language, and providing for the more technically oriented a needed balance of technical information.

The major aims of this Concepts and Facilities Manual are:-

(1) To provide you with a broad understanding of Jol. It will provide you with sufficent information to understand Jol's uses, Jol's capabilities, and Jol's benefits.

(2) To clearly demonstrate that the benefits of the powerful Jol Command Language meet the needs of today's market place.

(3) To outline all the new developments and changes that have been made to Jol.

Over the last few years, Jol has experienced enormous developments and changes that have been geared to increasing its power, flexibility, proficiency and ease of use. For example, Jol is now twice as powerful as when it was initially released, as measured by its proficiency in operations. Jol can now use IBM's Dynamic Allocation so that jobs can run under TSO or in Background, with or without the use of JCL.

In addition, Scheduling and Networking Facilities have been added that allow you to specify which jobs are to run on which days, and which jobs are to run concurrently. This facility works equally well with Jol jobs or JCL.

The recent addition of a *Data Base of Data Sets* attributes reduces the code required to write JCL to approximately one-quarter that required for JCL, and provides compatability with UNIX. Jol automatically merges information from the data base, and creates all data sets (including VSAM data sets) when necessary.

We have also developed a Personal Computer version of Jol that can generate JCL for MVS and soon DOS/VSE and UNIX.

The total Jol package is now accompanied by an interactive "Teach Yourself Jol" kit, that can be run on any IBM Personal Computer.

For a different view on the new and more powerful Jol Universal Command Language, review the "Answers to Questions" booklet; and for a Financial

Appraisal, work through the "Evaluation Plan and Financial Work Sheets".

Summary of Amendments

SUMMARY OF RECENT IMPORTANT DEVELOPMENTS TO Jol

Data Base of Data Sets

- Data Set attributes can be stored in a *Data Base*. Jol automatically merges information from the data base, and creates all data sets (including VSAM data sets) when necessary.

  This reduces the code required to write JCL to approximately one-tenth that required for JCL, and provides compatability with UNIX.

TSO Support.

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

Scheduling and Networking Facilities.

- Facilities to specify which jobs are to run on which days, and which jobs are to run concurrently. Works equally well with Jol jobs or JCL.

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.

  In addition, a **CALL** instruction will execute any problem program at Compile Time.

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.

ASSOCIATED DOCUMENTATION INCLUDES:

- **Jol Reference Manual**
- **Jol Reference Guide**
- **Jol Answers to Questions**
- **Jol Evaluation Plan and Financial Work Sheets**
- **Jol Conversion Utilities**
- **Jol Planning and Installation Guide**
- **Jol System Programmer Guide**
- **Jol Program Logic Manual**

INTRODUCTION

Command Languages

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

Use of Command Languages spreads across all areas of Data Processing. Without them, we have no means of communicating with the Operating System. Inefficient use of a Command Language can have disastrous effects on corporations' computing resources.

Today, not all computer Users are Computer Technicians. Therefore, it is extremely important that the Command Language used is simple and easy to use. Additionally, with the increasing expansion of computer resources and the growing number of software packages being acquired by each installation, the Computer Users are increasingly needing more power and flexibility in their Command languages. Jol is unique in meeting these new standards for Command Languages.

Jol uses a simple, flexible, concise and English-like command structure to communicate with your operating system and to effectively control data, programs, and events. It is easy to learn, easy to use, and easy to change. With these features, Jol allows Users to better utilize their skills, experience, and creative abilities enabling them to be more proficient than was ever previously possible.

Jol is 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 with the flexibility to solve the most complex type of requirements in a logical straightforward manner. By combining the flexibilities of these modern procedural languages with many new features, Jol provides a simple, powerful, and flexible INTERFACE to the operating system.

In addition, Jol coexists with JCL, and interfaces with contemporary development techniques, such as top down design, step level refinement, structured coding, and prototyping. Jol's many features focus on the End User, programming maintenance and development, production (e.g., operations and scheduling), management control, machine utilization, job scheduling and job networking.
Jol has some 40 commands. These 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.

Jol has many other highlights. Management can use Jol to monitor jobs and trap inefficiencies before jobs begin to execute. Systems Programmers can alter the inefficient code and make it more efficient through Exits. Data set protection facilities are also offered.

The data set attribute data base allows the data manager great flexibility - data set attributes can be changed without altering any of the Jol command language scripts.

Currently Jol complements the powerful IBM MVS systems (TSO and ISPF) by providing, for example, simplified instructions and access to Calendar facilities at the Command Language level.

The Personal Computer version of Jol produces either JCL or a special *pseudo-code*, so that jobs can be initialized on the Personal Computer and then submitted to the mainframe for execution.

Language translators are available to convert MVS, DOS and X8 JCL to Jol.

Functions

The power, simplicity and flexibility of Jol are highlighted by the list below of some of Jol's instructions, and in turn by the **"PRINT"** example.

With Jol you can:

- **Schedule** and **Network** Jobs. You specify which jobs are to run on which days, and which jobs are to run concurrently. This facility works equally well with Jol jobs or JCL.š
- **RUN** programs.
- **PRINT** data sets.
- **COPY** data sets or volumes containing data.
- **CATALOG** or **DELETE** data sets.
- **Test Return Codes, Error conditions** and **Symbolic Variables**.
- **SUBMIT** other jobs to the system.
- **ALLOCATE, READ and WRITE** data sets.
- **List Catalogs**.
- **Write** your new Jol Instructions.
- and more.

For example, to **PRINT** a data set you could say:

PRINT JOL.INCLUDE(PAYROLL);

which will print on the printer member **PAYROLL** of the **JOL.INCLUDE** data set.

To **SORT** a card image data set, and **CATALOG** and **PRINT** it may be coded as:

```
DCL CARDS *;
cards
EOF;

SORTSTEP:
    SORT CARDS                  /* Sort cards as specified  */
        TO SORTED.CARDS         /* to Output Data Set */
        FIELDS=(10,10,CH,A);    /* over fields */

    IF SORTSTEP=0               /* If sorted successfully */
    THEN DO;
            CATLG OUTPUT;       /* then catalog data set */
            PRINT OUTPUT;       /* and Print it. */
    END;
```

Figure 1-1. A small Jol Job Example

Note particularly that the output data (**SORTED.CARDS**) has no attributes specified. Jol finds the attributes, space and volume from the data set data base. This provides compatibility with UNIX and other operating systems.

```
DCL CARDS *;
cards
EOF;
DCL OUTPUT DATA SET     /* Define output data set */
        SORTED.CARDS    /* Define name */
        FB 80,800       /* Define record format */
        100 RECORDS     /* Define space */
        SYSDA;          /* Define unit */

SORTSTEP:
        SORT CARDS              /* Sort cards as specified  */
        TO OUTPUT               /* to output */
        FIELDS=(10,10,CH,A);    /* over fields */

        IF SORTSTEP=0           /* If sorted successfully */
        THEN DO;
                CATLG OUTPUT;       /* then catalog data set */
                PRINT OUTPUT;       /* and Print it. */
        END;
```

Figure 1-2. A small Jol Job Example

This second example shows how you can specify data set attributes within your Jol program instead of using the data set data base to retrieve the data set attributes.

Additionally, the Jol Macro facility allows an installation to tailor the Command Language to its own requirements. Often, Users suggest additions that can be used by other organizations. These are then passed on to all installations in the form of Newsletters, and sometimes incorporated into later releases of The Universal Command Language.

Jol Concepts and Facilities Manual - Preface

Structure of this Concepts and Facilities Manual

This Concepts and Facilities Manual has been divided into the following sections:

Section One: Introduction.

Section Two: Benefits of Jol.

Outlines the benefits of Jol to your overall organization, with an emphasis towards those benefits that reduce costs and increase your productivity.

Section Three: Using Jol.

A general overview on how to operate Jol. Different methods of using Jol and the general format of Jol jobs are outlined. The Computer Aided Instruction facility in learning Jol is also outlined. Throughout this section the power, the extensive features and the simplicity of Jol are highlighted.

Section Four: General Characteristics of Jol.

This section outlines the general characteristics in operating Jol. It outlines some further features that affect the general operations of Jol. It basically extends the logic of Section Three, but in greater detail.

Section Five: Additional Facilities

This section outlines additional facilities of Jol. Again with an emphasis towards User Benefits and advantages of Jol over alternative Command Languages.

Section Six: User Exits

This section outlines one of the most important additional facilities of Jol - User Exits.

Section Seven: Summary of Jol's Instructions

This section presents a list of instructions available in Jol. This section also presents a sample list of Macro Command Instructions that are available with Jol.

Jol Concepts and Facilities Manual - Preface

BENEFITS OF Jol

The Command Language to Complement Powerful Computers and Software

The benefits of Jol are numerous, outstanding and unequalled by any existing Command Language or OS Interface.

After reviewing other Command Languages and developing Jol to its current standard, we can confidently claim: "After more than two decades, Jol has found and brought the basic benefits of Computers back to your business - the place they should have never left."

Dramatic improvements have been made to IBM computer hardware, operating design and performance. Computers have gone from 1 million instructions per second to more than 27 million instructions per second. Operating systems have gone from performing one task to performing multiple tasks.

During this very same time, the Command Language has been neglected. Little has been done to enhance its development and in turn the productivity of the Programmer. To further exacerbate the predicament, the Applications Programmer must now know in the order of a dozen different languages in order to create an application - and this number is increasing. These languages are artificial, non-English like, and all different.

This neglect of the Command Language, has resulted in many non-productive hours of learning programs and the disproportional growth of I.T. departments with the hiring of many extra staff. All this has resulted in a reported annual cost of billions of dollars to corporations.

Jol the Universal Command Language goes a long way to solve many of these problems relating to the interface with your computer. Jol even goes as far as converting many of these problems to big advantages for the User and your organization. The range and scope of the benefits that Jol brings to the Users of IBM and compatible computers are so great that we have yet to find an installation where the benefits of Jol have not greatly outweighed its cost.

Jol's Benefits are Real

No organization uses computing facilities unless those facilities are going to prove beneficial.

Benefits happen due to a saving of time, space, or money. A major criterion that must be met by any product, is the benefits must outweigh whatever problems may appear. With some computers and software the question of real benefits is a marginal one.

This does not apply to Jol.

Evaluating Jol

When an evaluation of a new software product is commissioned, the following criteria is generally the basis of the evaluation:

1. Does the product suit the installation?
2. Does it fulfill a current need?
3. Does it eliminate a known problem area?
4. Does it provide new benefits in the saving of time: machine time, people

time and learning time?

5. Does it save money?

This "Benefits of Jol" section demonstrates how Jol fulfills all these requirements. As every current User of Jol has reported, the savings are generally in hundreds of thousands of dollars per annum (sometimes reaching millions), and these savings can be realized soon after Jol is installed.

Below are listed some of the important areas where Jol provides real benefits. Obviously the importance and relevance of each area will differ according to the requirements of each installation. What is certain, is that Jol will save you resources - computer time, people time, and money.

Easy to Use

Jol is an English-like Command Language. Jol's similarity to other high level languages means that Users rapidly become proficient in its use, consequently allowing all its benefits to be reaped quickly.

Jol is easy to use.

Jol's English-like free form and procedural format make it easy to use, even for non-D.P. personnel.

Jol does not require the same painstaking attention to coding detail that JCL requires. This means that procedures can be coded with greater speed, while still achieving a much reduced incidence of Programmer error.

One of the most important design features of Jol is the need for extraneous words and details has been eliminated, allowing free form coding wherever possible. JCL is syntax restrictive and relies heavily on Keywords, Sequencing and Positional Parameters to obtain the information it requires to successfully execute a job. On the other hand, Jol has made most Keywords redundant and utilizes contextual recognition wherever possible. This is illustrated in the example below where four statements are written in Jol and JCL:

Jol	**JCL**
5 TRKS	**SPACE=(TRK,(5))**
2,1 MINS	**TIME=1** with elapsed time of 2
VB 100,7294	**DCB=(RECFM=VB,LRECL=100, BLKSZE=7294**)
IF A=10 | B=2	**COND=((10,NE,A),(2,NE,B))**

Even considering the above example and all the other comparisons to be made between Jol and JCL throughout this General Information Manual, it must be accepted that JCL is a powerful and efficient tool on IBM and Facom machines. However, one of the primary intentions of Jol is to provide the User with easier access to this tool.

Easy to Learn

Since Jol presents the Programmer with a distinctly simplified and logically structured view of the operating system, Jol can be learnt in a relatively short time.

Experience has shown that most Programmers can learn Jol from a three day course. New Jol Programmers are often writing complex control procedures within a week or two of their Jol course. Consequently, the Users' stresses associated with Learning a new language are reduced to a minimum, and cost savings begin accruing immediately.

Immediate Cost Savings

Jol's free format language syntax evokes sighs of relief as Applications Programmers begin to write their job control statements as easily as they write their normal job stream. As a result, the computer installation gains immeasurably through greater Programmer confidence and efficiency, as these Programmers need no longer rely on Systems Programmers to get their jobs running.

And Systems Programmers enjoy the extra hour or two a day they obtain from not having to correct myriads of errors in more junior Programmers' work. Jol allows Systems Programmers to actually get on with their own real work more effectively than ever before.

Computer industry surveys have shown that up to 20% of a Programmer's time is eaten up by maintaining existing JCL and writing new JCL. Taking the current average cost of employing a Programmer at $50,000 per year (this includes salary, insurance, payroll taxes, terminal, facilities etc.), $10,000 of this goes to maintaining and writing JCL. Using Jol, this $10,000 will be reduced to $3,000 - a saving of $7,000 per Programmer. If you have 25 Programmers in your installation, this could amount to $175,000 per year either saved or put to better use.

More Cost Savings - Re-Runs Run Out

JCL has inherent coding difficulties, like the need for exact brackets, commas and asterisks in a very demanding order. JCL's coding difficulties have caused re-runs of programs to become an occupational hazard at all IBM installations. Everyone knows they exist, but no-one likes to admit they exist. It's not the I.T. Managers' fault, but it's very difficult to explain to company Directors that re-runs are an IBM fact of life. We could almost guarantee that your installation is having 8-10 hours of re-runs a week.

According to surveys around 10 hours of re-runs occur each week due to JCL errors. If it costs around $300 per hour to run a computer, it is easy to see that many installations could lose up to $3,000 a week in this one area alone. As Jol will save approximately 80% of these re-runs, around $120,000 can be saved each year - not to mention Operator savings, and the prestige of being able to get

the work out on time, every time.

Jol has extensive pre-compile error checking facilities. These facilities can virtually eliminate re-runs caused by JCL errors. The saying at our User sites is: "If it's on the queue, it'll run!"

Scheduling and Networking

Jol contains a Scheduling and Networking System that is extremely simple to use and written *entirely in high level Jol Code*. This powerful facility allows you to easily make changes to the system to suit your installation. Using the facility, it is possible to program job schedules for days, months, or even years in advance. All your existing jobs can be started without changes. Furthermore, Jol can be used to schedule your de-bugged and already tested jobs written in **JCL**. Full Jol symbolic variable processing is allowed, even with JCL jobs.

The Jol Schedule Data Set contains various members that can be used to specify which jobs are to run on **WORKDAYS, HOLIDAYS, MONDAYS, TUESDAYS**, etc., and jobs for particular dates such as **JUL12**. At a time suitable to you, Jol examines this data set and prepares all the appropriate jobs for execution.

To specify which jobs are to be started, simply code the name of the member containing the job in the **JOL.SCHEDULE** data set. For example, to indicate which jobs are to be executed daily may be done by coding the names of the jobs in member **WORKDAY**, as shown below.

```
PREPARE BACKUPS;
PREPARE ACCOUNTS;
PREPARE PAYROLL;
```

Figure 2-1: Indicating Which Jobs are to Run.

The jobs named are submitted for execution on the appropriate days. Jobs may contain any Jol instruction, and include networking instructions. For example, even daily jobs can examine the Jol calendar or accept information from the terminal and dynamically alter themselves before submission.

Planning Ahead

Jobs in member **WORKDAY** are submitted on a daily basis. In addition, other members of the library are examined based on dates. The following members of the schedule data set are executed on the appropriate days:

WORKDAY	Executed only on normal working days.
HOLIDAY	Executed only on non-working days.
WEEKEND	Executed only on Weekends.
SUN	Executed on **SUNDAY**s.
MON	Executed on **MONDAY**s.
TUE	Executed on **TUESDAY**s.
WED	Executed on **WEDNESDAY**s.
THU	Executed on **THURSDAY**s.
FRI	Executed on **FRIDAY**s.
SAT	Executed on **SATURDAY**s.

Figure 2-2: Members to be Examined on Weekdays.

In addition, the following members of the schedule data set are executed *(if present)* on the specified days:

JAN01	Executed on January 1st.
JAN02	Executed on January 2nd.
...	
FEB01	Executed on February 1st.
MAR01	Executed on March 1st.
APR01	Executed on April 1st.
MAY01	Executed on May 1st.
JUN01	Executed on June 1st.
JUL01	Executed on July 1st.
AUG01	Executed on August 1st.
SEP01	Executed on September 1st.
OCT01	Executed on October 1st.
NOV01	Executed on November 1st.
DEC01	Executed on December 1st.

Other jobs can be submitted on the last working day, the last day, the last working day -1 and so on.

It is also an easy matter to have jobs submitted every second week, should you so desire. A member called **SPECIAL** has prototype code for these types of jobs.

Figure 2-3: Month and Day Members Containing Scheduling Information.

Once a day, Jol examines the Schedule data set. If it is not a holiday, it examines the **WORKDAY** member; if it is Saturday or Sunday, the data in these members is used instead, otherwise member **HOLIDAY** is used. Additionally, any other member such as **JUL14** is also examined, if appropriate. The jobs found are then submitted.

Other Useful Facilities for Use in Scheduling

Additionally, Jol allows:

- Read and Write access to existing or new data sets at compile time.

  These data sets may contain other scheduling information or further details of work to be run on particular days, or under particular circumstances, or both. Jol can create tailored job streams. These are created on finding the appropriate data in the schedule.

- Access to the system calendar.

  Jol allows the date (year, month, day) and time to be accessed. Using this data, Jol can create tailored job streams.

- Testing for the existence of data sets.

  Jol can test if a data set exists. If a data set does exist, it is possible to submit other jobs to run, or take a different path through the current job.

- Symbolic Parameters or Variables can be passed from Job to Job.

Networking of Jobs

Often it is desirable to process two or more jobs in parallel, and after those jobs have reached completion, start one or more jobs.

To prepare a Jol network is a simple task. You can use your previous Jol or JCL jobs without alteration. Prepare your network commands and place them in your Jol program. These commands typically submit Jol and/or JCL jobs.

A typical network job might be that jobs **JOB1** and **JOB2** are to run, and after both jobs have completed **JOB3** is to commence.

This can be represented as shown below:

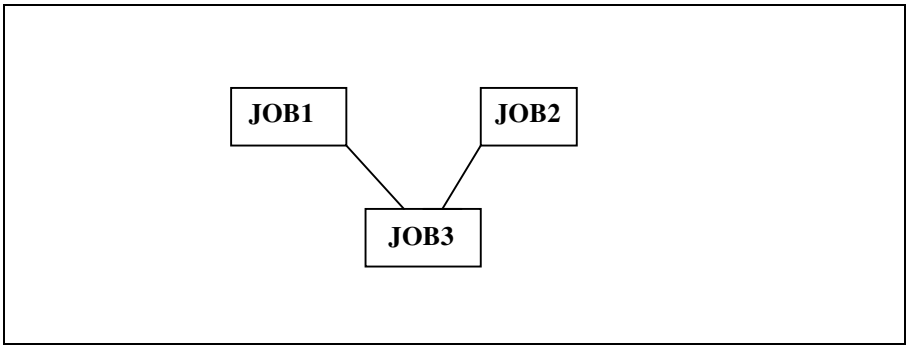

Figure 2-4: A Small Job Network

This may be coded in Jol in the following manner:

```
NETWORK ONE;
      SUBMIT JOB1;
      SUBMIT JOB2;
      SUBMIT JOB3 AFTER JOB1 & JOB2 ENDED;
ENDNET;
```

Notice the Jol instructions **NETWORK** and **ENDNET**. The **NETWORK** instruction is followed by the *name* of a network. This defines a unique network so that you can submit the same jobs in different networks, and still allow Jol to network the correct jobs. The **ENDNET** informs Jol that any instructions following are not part of the network.

You can also use other Jol instructions such as **PANEL** in a network. Any instruction will be executed as normal, except for **SUBMIT**, which is copied to a work file for possible re-execution at the end of each job.

For example, the jobs above may require the Operator to provide Symbolic Variables for the jobs.

For example:

```
PANEL ('ENTER TODAY''S DOLLAR RATE',DOLLVAL,7);

NETWORK ONE;
      SUBMIT JOB1 SYMS('DOLLVAL=%DOLLVAL);
      SUBMIT JOB2;
      SUBMIT JOB3 SYMS('DAY=%DAY')
            AFTER JOB1 & JOB2 ENDED ;
ENDNET;
```

Figure 2-5: A Small Network with PANEL and Symbolic Variables.

A more complex example of Networking is represented below:

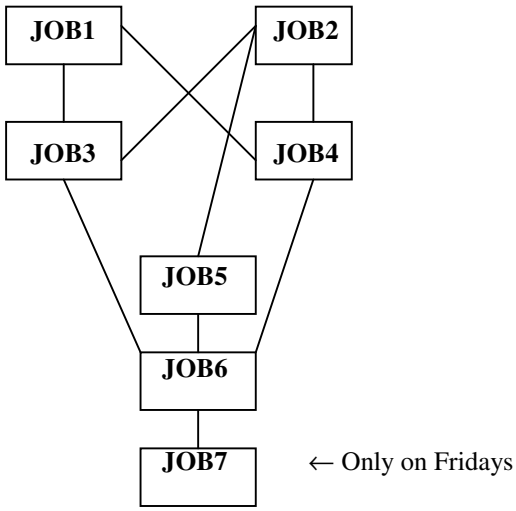

Figure 2-6: An Advanced Network.

In this example, **JOB1** and **JOB2** may proceed in parallel. The other jobs must execute in the following sequence:

- **JOB3** can commence processing after *either* **JOB1** or **JOB2** have ended.
- **JOB4** cannot commence until *both* **JOB1** and **JOB2** have finished.
- **JOB5** must wait for **JOB2** to end before it can start.
- **JOB6** must wait for **JOB3, JOB4** and **JOB5** before it can start.
- **JOB7** must wait for **JOB6** to end, and it is to be executed *only on Fridays*.

One possible method of coding the example above is presented:

```
NETWORK TWO;
      SUBMIT JOB1;
      SUBMIT JOB2;
      SUBMIT JOB3 AFTER JOB1 OR JOB2 ENDED;
      SUBMIT JOB4 AFTER JOB1 AND JOB2 ENDED;
      SUBMIT JOB5 AFTER JOB2 ENDED;
      SUBMIT JOB6
            AFTER JOB3 & JOB4 & JOB5 ENDED;
      IF %DAY='FRIDAY' THEN
            SUBMIT JOB7 AFTER JOB6 ENDED;
ENDNET;
```

Figure 2-7: Coding the Advanced Network.

Submitting Dependent Jobs

In addition to the Jol **NETWORK** instructions, the Jol **SUBMIT** Command ensures the correct order of processing interdependent jobs is observed. Other jobs may be **SUBMIT**ted at any time - even before the end of the current job.

Usually, the source text for **SUBMIT**ted jobs can be found in a library.

Please note that ASP and JES3 have facilities allowing the creation of a "Job Net". Jobnets are supported by Jol.

Increased Efficiency and Better Computer Utilization

Through the basic design features of Jol, a far greater degree of efficient utilization of computer resources is possible. For example:

- Catalog searching is performed at compile time, thus minimizing elapsed time, and providing better error checking.
- Data sets are declared only once in a Jol job, hence the catalog needs to be searched only once, irrespective of the number of times the data set is used.

*For the **Dynamic Allocation** option:*

- When Jol is using Dynamic Allocation instead of generating JCL, it monitors the progress of the job at the end of each program execution, and frees resources as soon as possible.
- Jobs can run immediately under TSO, thus providing immediate results.
- Dynamic Allocation can be faster than JCL.

*For the **JCL Generate** option:*

- Jol generates optimized JCL.
- The total number of OS steps is reduced, because the Jol processor has a transient scheduler/initiator incorporated in it. All catalog and scratch statements can be performed in a single OS step.

Jol Users have reported a 3% - 10% reduction in computer time as a result of the extra efficiency Jol provides over JCL, CLISTS and ISPF. Jol Users have also reported that, in some instances, this reduction can be even greater.

In addition to these savings, the job validation checks applied by Jol, substantially reduce the incidence of jobs failing due to Control Language coding errors. In general, it is not unreasonable to expect a 75% reduction in the occurrence of these failures.

Increased Management Control

Jol provides management with an opportunity to monitor all jobs entering a system and hence, to exercise a degree of control that has never been previously available.

By incorporating User supplied routines into the Jol processor, Jol makes the following management control facilities available:

- Control of authorized use of the machine.
- Control of access to data sets. Sensitive information can be protected by ensuring that only authorized personnel (or departments) read or write data sets using authorized programs.
- Strict enforcement of standards and conventions for data set names, program names, etc.
- Determination of which programs are using which resources.
- Application of efficient default values for missing information and the detection and correction of inefficient specifications.
- Provision of an easy means of transferring jobs to other computers in case of hardware failure.
- Printing of System Status Information (SSI) for each program. This is a program audit facility that shows precisely which program has been executed.

Increased Error Detection

As previously outlined, Jol substantially reduces the number of job failures caused by incorrect job control information. However, some errors will still occur. Jol's comprehensive set of validation checks ensures that all errors will be isolated at the commencement of the job. The errors will be clearly defined for the User by over 400 informative error messages. The messages clearly define the error and its location so that the User knows exactly where and which error has occurred.

In addition to the normal syntax checks that are applied to each Jol statement, the internal consistency of the total job procedure is examined. If any inconsistencies are detected, Jol either corrects the procedure or does not submit the job for execution.

Jol also has an error detection facility that allows Operators to simply find errors. Execution messages are placed at the beginning of the job, so that pages of JCL do not have to be scanned to find messages or errors.

Easy Modification of Programs

The attributes that make Jol easy to learn and easy to use, also make it easy to modify. In many cases, a change in a cataloged text can be made simply by the use of an INCLUDE instruction or by changing a macro in the Jol MACRO library.

Jol Enhances TSO

Jol's **PANEL** instruction provides further time and cost savings, particularly to the MVS User who depends on CLISTS.

The **PANEL** instruction provides a powerful method for inputting data to Jol under an interactive system such as TSO, TSO/ISPF or a Personal Computer. The text is displayed on the screen including all entries and defaults. The User

can modify the default entries supplied by Jol and then enter all the data to Jol. The Jol code can then be modified using the data from the screen.

Allocating, Reading and Writing Data Sets

Like TSO, Jol has instructions to Allocate, Open, Read and Write data sets. Data Sets can be copied or created.

Typical uses of these instructions include:

- Data Entry, using the Jol **PANEL** instruction to input data that may then be used by an update program.

- Reading a machine readable schedule, then submitting jobs based on that schedule.

These features provide greater Programmer productivity and greater computer efficiency. Each is discussed in greater detail in the "Summary of Jol's Instructions" section of this Concepts and Facilities Manual.

Other Jol Features to Further Increase Programmer Productivity

Programmer time spent on maintenance and development is a necessary cost in running an OS Data Center. However, a significant portion of this time is normally spent in performing non-programming related functions. Jol provides the following features that substantially reduce the time and training required to carry out these non-programming related functions.

Relational Capability

Jol's powerful macro facility is a set of instructions stored in a macro library. This macro facility can be used to create Jol instructions tailored to the Users' specifications. This facility provides a transparent error free interface for the Programmer and User to existing software products. Additionally, Jol macro instructions such as **SORT, COPY, PRINT, COMPILE** and **COMPRESS** are available to reduce the redundant labor efforts normally associated with these common routines.

Change Facility

Jol's comprehensive Command structures reduce the labor intensive practices associated with OS jobstreams. With Jol, all program and data set definitions are defined only once. This feature dramatically reduces the time required to implement changes.

Registration Capability

Jol enables Users to record all attributes of a program once (e.g. name, language type, source location, function, load library name, file names, compiler options, link options, autocall libraries, and copy libraries). This facility provides standard enforcement and swift accessibility to the attributes of the program.

Library Facility

Jol provides an include capability similar to the COBOL copy verb which allows common or static code to be stored in the **INCLUDE** library. The **INCLUDE** library can contain program, data set, and data card definitions which can optionally be defined in a single member or as individually included members. The current values of these definitions can be defined with symbolic variables allowing temporary changes to the procedures, and enabling Multiple Users to share the member containing these definitions.

Dependent Job Submission Facility

In addition to the full Jol Scheduling Facility mentioned above, Jol enables Users to submit subsequent jobs at any point within the job currently executing. This process provides a facility to effectively schedule dependent jobstreams. The dependent job is not submitted 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.

Logic Capability

Jol contains **IF, THEN, ELSE, AND/OR,** and **DO** logic. This logic provides the Programmer with a powerful tool for manipulating programs in jobstreams based on variable conditions, calendar information, and ABEND situations.

Enhanced Data Center Operations

An efficient production environment is vital for the smooth operation of a Data Center. Jol provides the following features to help you achieve the right work environment.

On-Line Data Entry for Job Submission

Jol enables Users to create '**PANELS**' or pre-formatted screens for inputting data to Jol or other programs. Effective use of this feature insulates the User from most of the tiresome details of entering information pertinent to a job(s), and at the same time, provides an efficient low overhead method of doing so. Facilities are provided to:

- Display text.
- Display text and allow replies to be entered.
- Display text with default replies.

Calendar Facility

Jol has an in-built calendar which can be used to select jobs or parts of jobs on particular dates, days, months or years.

Rerun/Restart Facility

By using symbolic variables, Jol gives you total flexibility in organizing restarts with simple instructions. This facility allows you to commence the job execution at any point other than the beginning of a job, and also allows you to stop the job at any point other than at the end of a job.

Standards Enforcement

Jol provides the means for the consistent enforcement of standards and for the enforcement of standards across all required levels. User supplied routines can be incorporated into the Jol processor when it is generated. Jol makes available to these routines all the job control information pertinent to a particular job. These routines can then examine this information and override any element if necessary.

Communication Ability

Jol provides a facility to communicate with the Operator or User, and to place messages on the System Log of the job. This communication can take 2 forms:

1. Simple instructions to the User.
2. Request a specific reply from the User.

Easy Overriding

All Jol definitions can be overridden. To override a job, program, data set, or symbolic definitions, simply code another definition before the existing definition. Jol gives the first definition preference. In some cases no modification of the Jol text is required. When processing needs to be transferred to a back-up machine with a different hardware configuration, the Jol processor can automatically perform all the necessary global changes. Similarly, generic changes resulting from alterations in the specifications of a public or shared program, can be optionally accomplished by changing a catalog text in the Jol 'INCLUDE' library or by changing a macro prototype in the Jol 'MACRO' library.

Efficient Computer Usage

Maximum utilization of the Operating System resources is essential in achieving the smooth and economical running of a Data Center. Jol comfortably achieves this criterion, and easily surpasses other Command Languages and OS interfaces with the following group of facilities:

Catalog Management

Catalog searching is minimized and only needs to be performed at submission time. Data sets are defined only once in a Jol job, hence the catalog needs to be searched only once regardless of the number of times a data set is used.

Tape Access Management

Jol ensures that tape data sets are left positioned correctly, even in cases where several data sets are created or read on one tape volume.

Reduction in Job Steps

Even when the JCL generate option is used, all **CATALOG** and **SCRATCH** statements are performed in the one OS step due to a transient Scheduler/Initiator being incorporated in the Jol Processor. Sometimes, several programs can be executed in the same OS job step - further reducing the OS Initiation/Termination overheads.

Reduced Overheads in Symbolic Translation

Symbolic usage is greatly reduced because all Jol definitions are defined once. Additionally, all symbolics are resolved prior to submission - further reducing the OS Reader/Interpreter overheads.

No Hooks to the Operating System

Installation of Jol is simple and does not require an IPL. Additionally, User Exits can be incorporated into Jol to enforce standards and provide User defined facilities without concern to the operating system. This ensures that an operating system upgrade does not necessitate a reapplication of these exits.

Jol Concepts and Facilities Manual - Preface

USING Jol

Overview in Using Jol

Simplicity in use, is the first feature of Jol to be noticed by the first time User. Power, speed and Jol's extensive range of facilities are next noticed.

In summary, to use Jol, the User firstly specifies 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.

Before the Jol Compiler allows the job to execute, it runs detailed validation checks on the instructions and declarations that have been made available. Thus saving computer time that could be lost through job failures. It checks that no execution of an instruction will cause the job to fail at any stage, and that there are no data sets or libraries missing.

If no serious errors are detected, Jol will allow the operating system to schedule the job for execution in a background region, or the job may be run immediately under TSO.

Jol may be executed interactively or in a background region. Executing Jol interactively allows the use of full screen **PANEL**s to input data. This data can then be used to create different job streams or as input to your programs. This process is described as Preprocessing the Jol text, and is described in detail in a later section of this manual.

Computer Aided Instruction in Learning Jol

A Computer Aided Instruction (CAI) package for Jol is provided to assist in the learning of Jol. The CAI package provides a complete teach yourself facility, in individual selectable units, with tests at the end of each unit for self evaluation. The CAI package is interactive and uses full screen **PANEL**s to display the data.

To run the CAI package, simply enter the following Command:

CAIJOL;

This command invokes the CAI, and then the CAI displays the following menu screen:

CAI MENU

Jol introduction and overview	=1
Jol language structure	=2
Declaring jobs, programs, and datasets	=3
Symbolic variables	=4
Disposition processing	=5
IF statement and REDO instruction	=6
Compiling and running your programs	=7
Dataset management commands	=8
Communications and error handling	=9
Job processing facilities	=10
Macro facilities	=11
Extra commands	=12
Terminal Users Guide	=13

Figure 3-1. CAI Menu

Notes relating to Jol's CAI package:

1. A special version of the CAI Jol Course is available for the IBM Personal Computer.
2. CAIJOL runs interactively under TSO or the Personal Computer.
3. PFK keys are used to provide further flexibility.
4. A test option is provided at the end of each unit for self evaluation.
5. The CAI package is expandable to incorporate any User-supplied information pertinent to the site.

Preparation of Jol Input

Two methods are provided for preparing input for Jol. The first is a series of formatted TSO screens which are menu driven. As a general rule, only certain key parameter fields are required for preparing input for Jol on these screens.

The second method allows the input to be prepared using any type of facility which provides fixed length 80 character records, such as a keypunch, TSO, CMS and other TP Monitors.

The two input methods can be intermixed.

The use of either input method removes the need for knowledge of OS syntax, keywords, format requirements, parameter positioning and sub-parameter sequences.

You can concentrate strictly on defining the values of the variable parameters. The variable parameters are the jobname, program names, names of data files, and attributes of data files. These must be supplied with each job if the files or data sets are not cataloged or are being newly created. Other data fields may be optionally used to make use of various OS facilities.

The technical documentation that accompanies the Jol system contains detailed explanations of all fields.

Methods of Using Jol

Two methods are provided in using Jol. Jol can be used either in Dynamic and Interactive mode, or in Batch mode.

- The Interactive method operates via a Terminal Processing monitor. With this method, Jol becomes conversational and can be menu driven. A "HELP" facility and the Computer Aided Instruction (CAI) facility are also provided. To further simplify the use of Jol, Menus or "PANELS" can be created by Users.

- The Batch method involves executing a supplied procedure and selecting the required Jol text member. The Jol input can be prepared using any type of facility which will provide fixed length 80 character records, for example, keypunch, TSO/ISPF, CMS, CICS and other TP monitors.

Both methods make full use of all Jol logic facilities.

General Format of a Jol Program

A Jol job is made up of one or more statements, instructions or commands.

Statements may **DECLARE** or **DEFINE** variables, data sets, programs, etc. Instructions and Commands cause some action to be taken with the items **DECLARED** or **DEFINED**.

In general the following procedure is followed:

1. Code all variable definitions.

2. Code all the data set and program definitions.

3. Code the instructions and commands to manipulate the defined data.

Figures 3-1 and 3-2 below show examples of how a Jol job can be layed out and documented.

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

STEP10:
        SORT    PAYROLL.TRANS
                TO PAYROLL.SORTED.TRANS
                FIELDS = (69,2,CH,A) ;

        IF STEP10 = 0
        THEN DO ;
             CATLG OUTPUT ;
             SUBMIT PAYUPDT ;
        END ;
```

Figure 3-2. A condensed Jol Job Example.

The **SORTJOB** above sorts the **PAYROLL.TRANS** file to **PAYROLL.SORTED.TRANS** using the sort fields specified in the **FIELDS** parameter.

The attributes, volume and space requirements for the output data set are stored in the *data set* data base. This data set is usually maintained by the Data Set Administrator, and separates the data set attributes from the Jol program.

The data set attributes may have been specified as:

```
PAYROLL.SORTED.TRANS
        FB 80,800 5 TRKS
        SYSDA VOL=WORK01 ;
```

```
/*  THIS JOB PERFORMS A SORT ON THE DATASET
      'PAYROLL.TRANS' AND PRODUCES A DATASET
      CALLED 'PAYROLL.SORTED.TRANS'.

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)'    /* Specifies Accounting Information
                                   */
    NAME = JOLUSER              /* Specifies the Programmer Name */
    CLASS = A                   /* Allocates the Job Class */
    USER = '??????'             /* Provides USERID for RACF */
    PASSWORD = '????????' ;     /* Provides PASSWORD for RACF */


STEP10:                         /* Stepname */
    SORT PAYROLL.TRANS          /* SORT Command */
    TO PAYROLL.SORTED.TRANS
    FIELDS = (69,               /* Start of Field */
              2,                /* Length */
              CH,               /* Character */
              A) ;              /* Ascending */

    IF STEP10 = 0               /* SORT OK? */
    THEN DO ;
      CATLG PAYROLL.SORTED.TRANS;
      SUBMIT PAYUPDT;           /* Submit the next job */
    END;
```

Figure 3-3. A Fully Documented Jol Job Example.

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

1. Definitions, instructions, and commands can be intermixed. However, clarity is greatly improved if the suggested procedure is followed.
2. Symbolic variable processing can be performed anywhere in the job.
3. Defined items do not have to be used in any instructions, thus allowing them to be placed in a member of a library and **INCLUDE**d. At the same time the instructions to the items can be coded following the **INCLUDE** instruction.
4. Data set identifiers or DSIDs can be allocated to all data set definitions. These DSIDs can then be used in instructions, commands and program definitions. This means that any details about a data set need only be defined once, and thereafter every reference to the DSID automatically picks up the required information.
5. Comments can be coded anywhere in the jobstream where a blank space is allowed.

Automatic Scheduling

While Jol can be used as a simple replacement or enhancement package for JCL and CLISTS, Jol has other facilities that have been designed to assist with the scheduling of jobs. For example, Jol allows:

- Read and Write access to existing or new data sets.

  These data sets may contain schedules or details of work to be run on particular days, or under particular circumstances, or both. Jol can create tailored job streams. These are created on finding the appropriate data in the schedule.

- Access to the system calender.

  Jol allows the date (year, month, day) and time to be accessed. Using this data, Jol can create tailored job streams.

- Testing for the existence of data sets.

  Jol can test if a data set exists. If a data set does exist, it is possible to submit other jobs to run, or take a different path through the current job.

Networking

Many Installations have a routine enabling a job to start another job, thus ensuring that only after the first job has reached a certain point the second one begins execution. Usually, the second job is placed on the System Input Queue in a HOLD status. The first job issues a START or ACTIVATE command to tell the Operating System that it may now begin the execution of the second job.

While Jol fully supports such methods, it offers a further and more powerful method for Networking. Jol allows the invocation of itself at any time from a currently executing Job to submit a second job, rather than having the second job on the System Input queue.

The advantages of this method are:

- Space is not wasted on the System Input Queues.

- If the first job fails, the Operators do not have to concern themselves with the second job, as it will not have been placed on the System Input queue.

- The Operator cannot accidently activate the second job, because it isn't on the System Input Queue.

- Symbolic Parameters or Variables can be passed from Job to Job.

Submitting Other Jobs

The Jol **SUBMIT** Command ensures the correct order of processing interdependent jobs is observed. Other jobs may be **SUBMIT**ted at any time - even before the end of the current job.

Usually, the source text for **SUBMIT**ted jobs can be found in a library.

Passing Symbolic Variables to Other Jobs

Symbolic parameters can be passed from Job to Job. Symbolic parameters are often used to pass information to programs. An example of information that can be passed to programs, is a date or other information that is to be printed as part of a report. Many jobs may require the use of the same information. By using the **SUBMIT** Command, it needs to be defined only once, and then all other jobs

will have the same value or the same information.

Shell (Australia) have several jobs that are initially started with a START command. Through the Console, the first Jol job is issued the values of some Symbolic Variables. The first job then uses SUBMIT to generate the second job, and it also passes all the original symbolic Variables through to the second job. This process may be repeated infinitely. For example:

```
JOB1 : JOB;
        process
  IF MAXCC < 8 | LASTSTEP=0
  THEN DO;
        CATLG all new data sets;
        SUBMIT JOB2
              SYMS('DOLLVAL=%DOLLVAL,
                    DATE=%USERDATE');
  END;
```

If the maximum Return Code issued by any of the programs is less than 8, or the **LASTSTEP** is equal to 0, then the second job, **JOB2**, is allowed to begin execution. The second job can only begin execution after any data sets created in the first Job have been Cataloged in the System Catalog.

Scheduling According to Date

In addition to Jol's full Scheduling and Networking Facility, Jol has an internal calendar that sets up the current date (year, month and day). As indicated above, the date can be easily accessed by Symbolic Variables and passed onto the program. Therefore, not only does Jol allow you to submit jobs depending on conditions detected in other jobs, it also allows you to submit jobs depending on the date. It then becomes possible to program job schedules for days, months, or even years in advance.

Furthermore, Jol can be used to schedule your de-bugged and already tested jobs written in the IBM Job Control Language (JCL).

For example, to submit a job on every Thursday of the week only requires the following code:

```
IF %DAY='THURSDAY'
THEN SUBMIT required job;
```

You can be more specific and request that Jol submit the job only on the 30th of June. To do this requires the following code:

```
IF %DAYNO = 30
& %MONTH = JUNE
THEN SUBMIT JUNEJOB;
```

Using Jol with JCL

Scheduling can also be performed with JCL jobs. For example:

```
IF %DAY='WEDNESDAY'
THEN SUBMIT
    '//JOB1 JOB'
    '//PAYROLL EXEC PAYROLL,DAY=%DAY';
```

Not only is it possible to use the Jol date facilities, but you can also extensively use the Jol preprocessor when using JCL. It is often desirable to write all JCL in one central location and execute the procedures at remote computer sites.

In theory this is good practice, but in practise the remote installations invariably must make some alterations to the distributed JCL due to differing local requirements. The following example illustrates how to alter space requirements dependent upon whether the procedure is to be executed in Chicago or Kansas. The installation dependent catalog values are also correctly inserted in a system utility control card.

```
IF %LOCN='CHICAGO'
THEN DO;
        %SPACE='SPACE=(CYL,199)';
        %CATVOL='CHICAT';
        %CATUNIT='3350';
END;

IF %LOCN='KANSAS'
THEN DO;
        %SPACE='SPACE=(CYL,50)';
        %CATVOL='KANSAS';
        %CATUNIT='3380';
END;
```

later...

```
SUBMIT      '//JOB1 JOB (account etc)'
            '// EXEC MAIN,SPACE="%SPACE"'
            '//IEHLIST.SYSIN DD *'
            ' LISTCAT CVOL=%CATUNIT=%CATVOL';
```

which results in either:

```
    //JOB1 JOB (account etc)
    // EXEC MAIN,SPACE='(CYL,199)'          *
    //IEHLIST.SYSIN DD *
      LISTCAT CVOL=3350=CHICAT               *
```

or

```
    //JOB1 JOB (account etc)
    // EXEC MAIN,SPACE='(CYL,50)'           *
    //IEHLIST.SYSIN DD *
      LISTCAT CVOL=3380=KANSAS               *
                                             *Altered Lines
```

The above example highlights the power of the Jol Pre-processor while still allowing you to use JCL.

GENERAL CHARACTERISTICS OF Jol

A Control Language acts as an interface between the Operating System of the computer and its User.

The basic task that Jol performs is reduce the complexity of this interface to its most simple and logical form, something no other existing interface has managed to accomplish.

While making your computing system easier to use, Jol also adds a number of very desirable features to your system. These include: read and write access to data sets; the ability to check dates; and an extremely powerful macro facility that allows your installation to easily write new Jol instructions designed specifically for your organization.

Jol delivers all the above more efficiently than all other current methods!

Command Languages depend on the conventions and formats utilized in three main areas, and Jol excels in all three:

- Written Language.
- Language Syntax and Structure.
- Language Clarity and Semantics.

Written Languages

Jol statements have a free format and are written in a style used by most Programmers. There are no fixed columns for the commands and statements may be continued on to other lines and so on. In fact, the Jol language format is similiar to PL/I, PASCAL and "C".

Jol statements can be written over as many card images as necessary, without using any continuation card conventions.

However, if required several separate statements can be written on one card. Please note, that only one statement per card is recommended.

Statements are made up of a series of words or symbols which are delimited by space characters or punctuation symbols.

Each statement is ended by a semi-colon. Colons are used to specify statement labels.

Other punctuation symbols such as parentheses, commas and equality signs may be freely used to enhance the legibility of statements.

Language Syntax and Structure

Basically, Jol uses an English like command structure, although menus can also be used.

Wherever possible, Jol uses syntax that is the nearest equivalent to normal conversational English. This is achieved regardless of whether the language used is to define or declare jobs, programs or data sets, or used to execute programs, request operating system services, or communicate status information.

The syntax of Jol allows the definition of symbolic variables and the assignment of values to them. The syntactic form:

IF.....**THEN**.....**ELSE**.....

provides a control structure which may be used to test the values of, and the relationships between, symbolic variables and Return Codes. This syntactical form also allows different branches of the program code to be made after an IF statement.

Language Clarity and Semantics

Any computer language that will be used by more than one person must clearly express the work that must be done, and be easy to read by someone other than the author. Jol has both these qualities.

As in most programming languages, Jol allows the definition of entities such as data sets and programs, and provides simple instructions to use the defined variables.

Through Jol three principal entities can be identified and defined. These being: jobs, programs (or tasks) and data sets.

The static relationship between these entities are expressed by Jol through declarative statements. For example, the core storage requirement, time required to execute individual jobs, etc.

The dynamic relationships and interactions between the entities are expressed through procedural instructions. The principal elements of these procedures are the sequences of actions which can occur and the relative controlling conditions under which they occur.

A clear distinction between static and dynamic elements is maintained throughout Jol, enabling the control of overall system behavior to be expressed in a lucid and straightforward manner.

Definitions

Instructions manipulate jobs, programs and data sets. Definitions specify the static resources and data sets that can be used by the instruction.

Program Definition

A statement of the form:

DEFINE *name* **PROG** *options*;

defines a program. Options available include phrases such as:

internal-file-name **READS** *data-set-name*

These options bind the file names in programs to the data sets defined in the Jol data set data base or in the System Catalog. For example, to define the program called **UPDATE** which is to be found in the cataloged **PAYROLL.LIBRARY** program library:

```
DEFINE UPDATE PROGRAM,
     LIB=PAYROLL.LIBRARY,
     MASTIN      READS   ACCNTS.MASTER(0)
     MASTOUT     WRITES  ACCNTS.MASTER(+1)
     SYSPRINT    WRITES  PRINTER;
```

Note: Program definitions are typically stored by Jol and used with the **EXEC** statement.

Printer Output Definition

A statement of the form:

DEFINE *name* **PRINTER** *options*;

defines the relevant printer output. Options available incude: multi-part paper, special forms, record size, block size and format. For example:

```
DEFINE MULTI PRINTER 2 COPIES;
```

Card Image Input Definition

Any card image input can be included as part of a Jol program. Jol allows card image files to be read any number of times. MACRO Prototypes and Jol source text stored in an INCLUDE library can Å*both* contain card files.

The User also has the option of requesting Jol to replace any Symbolic Variables in the text of a file with their current values. This facility is particularly useful when creating control statements for utility purposes, for example:

```
DCL CARDFILE * REPLACE;
     SORT FIELDS=(%FIELDS)
EOF;
```

The current value of **%FIELDS** will be copied to the card image file.

Data Set Definitions

There are three classes of data sets: **OLD, NEW** and **TEMPORARY**.

Old Data Sets

An old data set is one which already exists *before* the job is run. It continues to exist after the job is run, unless a **SCRATCH** or **DELETE** instruction is executed for it.

New Data Sets

A new data set is one which is created within the job. It is automatically **SCRATCH**ed after its last use in the job, unless a **KEEP** or **CATALOG** instruction is executed for it.

Temporary Data Sets

A temporary data set is used for working storage in the job and is then automatically **SCRATCH**ed after its last use in the job.

A statement of the form:

DEFINE *data-set-identifier* **DATA SET** *options*;

defines a data set. The options available are numerous and relate to format, record and block sizes, unit type, volume, etc.

With Jol version 5, all data set attributes such as Record Format, Space and Volume may be saved in the data set data base for *automatic* inclusion by Jol when it requires such details - typically for new data sets.

You may also use *installation defined models* such as **SMALL, LARGE** for complete flexability, and to further reduce coding requirements.

Free Format and Optional Keywords

In many cases keywords may be coded or left out when a data set is defined. For example, defining a data set with dsname **YOUR.DATA.SET** on Volume **111111** as a Variable Blocked Data Set with 5 Cylinders and 1 Cylinder Secondary Allocation may be done in the following ways:

```
DCL  DSID1 DATA SET
        DSN=YOUR.DATA.SET VB 100,7294
        VOL 111111 UNIT 3350
        SPACE=5,1 CYLS;
```

or

```
DCL    DSID1 DS
        YOUR.DATA.SET
        VB 100,7294
        3350 111111
        5,1 CYCLS;
```

or

```
DCL    DSID1 DS
        VB 7294, 100
        DISK 5,1 CYLS
        VOL 111111
        YOUR.DATA.SET;
```

You will notice that the **VB** specification has 100,7294 or 7294,100 coded after it. Jol takes the largest number as being the Block Size, unless SPANNED records have been specified.

If you need to override part or all of the data set information, you only need to redeclare the **DSID** (Data Set Identifier), and the part overriding will then be carried through the entire generated JCL, no matter how many times the data set is used. This makes overriding extremely simple.

ADDITIONAL FACILITIES

One of Jol's features that has been repeatedly emphasized throughout this manual is the extensive range of User benefits and facilities available. Due to the very nature of these facilities, it is not possible to present detailed technical descriptions and explanations of these in this Concepts and Facilities Manual, or even make you aware of them all. However full details are available on request.

Presented below are only brief descriptions of some of the more important of these facilities.

Data Base of Data Set Attributes

Version 5 of Jol adds a data base of data sets to Jol, and makes a typical Jol job a series of **EXEC** or **RUN** instructions, with **COPIES, CATALOG** and other instructions *only*. With this facility, Mainframe jobs become as easy as running programs on MS/DOS or UNIX.

For example, to execute a program called **UPDATE** you can code:

```
Exec  Update Payroll.Master(0),
             Trans.Action(0),
             Payroll.Master(+1);

If Update = 0
then do;
       Catalog Payroll.Master(+1);
       Submit Job2;
end;
else Stop 'Error in Job';
```

Note: "Old Style" Jol programs may continue to be used, and the data base will be accessed for new data sets, and information merged with any current declares for data sets.

The addition of the Jol data set data base facility:

- Greatly reduces the training required for new JCL Programmers and increases their productivity even further.
- Further reduces the amount of coding required to run a Jol job. By removing the necessity to code program and data set declares, most Jol jobs are reduced by an average of 75% or more. Executing a program now only requires writing a line or two of code for each program.
- Provides a set of commands that will function in a similiar way on Mainframes, Personal Computers and Unix systems.
- Assists in creating an environment whereby the Data Manager can have more (and separate) control over data set placement and other attributes.

- Adds full VSAM support (with the exception of automatically deleting data sets at the end of a job).

Language Extension - Jol Macros or Commands

The Jol language can be tailored to an installation's specific requirements by using the Jol MACRO language to create new and *customized* Jol statements. The macro prototypes for these statements are stored in the JOL.CMDLIB library.

Although not discussed in this manual, Jol has facilities to incorporate Assembler, COBOL and PL/I routines into the Jol Compiler. These routines then become part of the Jol Language - tailored to your own requirements.

Macros

The Jol macro language does not impose an arbitrary fixed syntax for the invocation of macros, such as a fixed position and order for elements of the parameter list. The syntax of the macro can be designed to reflect the semantics of the statement. Jol allows keyword parameters, which may be coded anywhere in the invoking statement, and positional parameters, which may be located in a fixed position or relative to a keyword.

Furthermore, the macro prototype can be designed to allow many variations in the syntax actually used in the invoking statement. The only real restriction is that the name of the macro must be the first word in the statement.

The macro Language is a natural extension of the Compile Time Facilities.

It allows you to add new Jol instructions as you require them, and as simply as you do with Cataloged Procedures for OS normally. It also permits the use of any *previous* Macro Command, thus giving you the opportunity of virtually executing a procedure within a procedure.

Nearly all Jol code can be made into Macros by placing an extra MACRO statement at the beginning of the Jol text to define keywords, defaults, etc and placing an **END** at the end of the code, and then storing it in the Macro Library. Whenever a non-standard Jol instruction is then encountered, the code in the Macro will be read and executed if appropriate.

Source Text Library

Jol provides a facility for the storage of Jol source text in a library. This is known as the **INCLUDE** library. The library text can contain any Jol statements as well as *input card image data sets*.

A statement of the form:

INCLUDE *member-name*;

copies the text from the specified member into the Jol input stream, replacing the **INCLUDE** statement. It is then processed as if it were an original part of the input stream.

Applications of this facility include the storage of complete production job procedures, including any changeable card input data sets; and the storage of common procedures such as those used to invoke utility programs.Ý

The **INCLUDE** member statement may be used to conveniently include a series

of data set definitions. This allows the installation to set up various members of the library with data set definitions. Anyone who wants to run part of that set of programs, need only include the data set definition member to save coding out the required data sets.

Unlike JCL, data cards can be **INCLUDE**d in the text together with other text. You can even replace parts (or all) of these data cards with the current values contained in Symbolic Variables.

An **INCLUDE**d member may **INCLUDE** another member. This process can be performed to a depth of eleven levels.

Compile Time Facilities

All Jol programs are firstly compiled and then submitted to the host operating system for execution.

The compile stages may be divided into three main stages:

1. The MACRO or PREPROCESSOR phase.
2. The COMPILE phase.
3. The GENERATE phase.

While Jol is interpreting the Macro instructions in preparation for the COMPILE phase, you can alter selected parts of your program. These include:

1. Defining Symbolic Variables and initializing them.
2. Testing the current values and contents of individual Symbolic Variables.
3. Changing the values within Symbolic Variables.
4. Indicating the sections of the source program to be compiled.
5. Including source text from a library.

The output from this Macro phase consists of the updated source statements, which is then input to the compile phase to create the instructions necessary for the Operating System to execute the job.

Jol allows the User to freely intermix preprocessor and macro statements with the rest of the instructions within the job.

Symbolic Variables

Compared with JCL's Symbolic Parameter facilities, Jol's symbolic parameter facilities have been greatly extended to allow the values of symbolic variables to be:

- Tested.
- Changed.
- Added, subtracted, multiplied, divided and concatenated to other variables or to literal constants.

By using Jol's extended forms of symbolic parameter processing, it is possible to code large Jol jobs that have a large number of steps, some of which are executed

on a daily, monthly or yearly basis.

By examining a variable, such as the System Date, it is possible to generate *only* the required JCL for that particular run.

One example of this facility's application, is the creation of unique data set names which contain the date they were created on, or they can contain the name of the program they were created by.

In this field, the **IF** statement in Jol is especially useful to test Return Codes or Symbolic Variables or both. For example, assume that Symbolic Variable **%A** contains the value **'A'** in it. By using an **IF** statement, it is possible to check at any time if the value has been retained. The statement:

```
IF %A = 'A'
THEN DISPLAY 'VARIABLE A HAS VALUE %A' ;
```

will display the required message. If the variable **%A** contains any value other than **A**, no message will be displayed.

A simple IF statement can also be utilized to test the Return Codes of programs. For example:

```
IF PROGRAM1 > 8 THEN TYPE 'ERROR OCCURRED';
```

will perform a simple Return Code test and display a message on the Operator's console if **PROGRAM1** did not return an 8 or less on completion.

With Jol, it is also possible to use a simple **IF** statement to test both Symbolic Variables and Return Codes. For example:

```
IF %A = 10 & PROGRAM1 = 8 THEN.....
```

will initially cause the value of the Symbolic Variable **%A** to be checked. If **%A** contains a value of 10, the Jol compiler then creates an instruction to the monitor to test that the program, **PROGRAM1**, did run and that it returned an 8.

If **%A** was not found to contain a value of 10, the next instruction would not be executed, because the Jol preprocessor would determine that the instruction could never be true. Therefore, it would not even generate an instruction to the monitor to execute when the job is actually running.

The **IF** statement can be made as simple or as complex as required. Up to 40 different tests can be incorporated within one single statement, and there is no limit to the number of IF statements that can follow one another. Additionally, **DO** and **END** pairs can be used to create a group of instructions that are executed or bypassed.

A realistic example of this type of facility is shown in the example on the next page. It is the sort of operation carried out by many organizations for their payroll. It takes into account weekly paid and salaried staff, including the variation of week/month endings.

```
DCL %WEEKRUN, %MONTHRUN
      INIT ' ';                  /* Define Symbolics */

/*    Calculate if this date is the first day of the Business Week
      or the first day of the month.

      If so, set %WEEKRUN and %MONTHRUN to 'YES', because if so
      we shall run the Weekly and Monthly programs. */

IF %DAY='MONDAY'                 /* Is it MONDAY ? */
THEN %WEEKRUN='YES';             /* Yes, Weekly Processing */

IF %DAYNO=1                      /* First day in  Month ? */
   & (%DAY < 3
   & %DAY='MONDAY')              /* Take care of Weekends */
THEN %MONTHRUN='YES';

IF %WEEKRUN='YES'
THEN DO;                         /* Do Weekly Processing */
      RUN WEEKPGM;               /* Run first Program */
      IF WEEKPGM=0               /* Did it Execute correctly ? */
       THEN SUBMIT NEXTJOB;      /* Yes, so SUBMIT the next job of
                                    WEEKLY job stream */
END;

IF %MONTHRUN='YES'               /* Test for Month Run */
THEN DO;
      RUN MONTHPGM;              /* Run 1st Month Program */
      RUN MNTHUPDT;              /* and the second one */
      CATALOG X,Y;               /* CATALOG Output Data Sets */

      IF MONTHPGM=0              /* If both MONTHLY Programs
                                    & MNTHUPDT=0 THEN
                                    executed correctly, THEN
                                    SUBMIT the next two jobs
            SUBMIT JOB2,JOB3        passing the value of
                                    %MNTHRUN through to them */
          SYMS 'MONTHRUN=%MONTHRUN';

END;                             /* End of Job */
```

Figure 5-1 Sample of Payroll Job

In the above example, **WEEKPGM** and the generation of **NEXTJOB** will only occur on **MONDAY**s. **MONTHPGM**, **MNTHUPDT**, the **CATLG** instruction and the generation of **JOB2** and **JOB3** will happen only on the first day of the month, or the next Monday following.

Following the above program, one would normally have the DAILY processing programs to perform.

Restarts

To provide Users with the maximum flexibility, Jol itself does not have an inbuilt automatic restart facility. However, the **STARTAT** and **STOPAT** instructions allow the User to skip over any instructions that do not need to be executed.

Using Symbolic Variables offers you *total flexibility* in the way that restarts are organized. Although this initially means a little extra planning and coding, you can organize the Jol program so that one Symbolic Variable is set to a specific value. This value can be tested in your main line code and depending on its value you can:!a

- Reset generation numbers.
- Uncatalog or delete data sets or both.
- Rerun certain programs.
- Introduce new steps.
- Perform overrides on data set names.

For example,

```
IF %RESTART = 5
THEN PRINT MASTER(0);
ELSE PRINT MASTER(+1);
```

Simple Reruns

A data set is never **CATALOG**ed or **KEPT** *until the appropriate instruction has been given*. Therefore, it is possible to keep all the disposition instructions until the end of your job. If an error occurs, any **CATALOG, KEEP** or **DELETE** statements will not be performed, thus ensuring that the data sets will be left exactly as they were. The job can simply be rerun.

This facility can also be used with the generation of data groups.

Advanced Methods

A more advanced method, especially applicable to long jobs, is to set up the Jol program in such a manner that a series of regular reference points are set up. At these points the normal disposition processing will be performed as described above.

However, if an error occurs, the program will abort and the disposition processing will not have been performed. By using Symbolic Variables it is possible to backtrack to the previous point from which a rerun can commence.

While this method requires some extra planning, it is by far the most powerful way of restarting a job. Some of the restart programs may replace defective copies of data sets from Backup copies or other Delete Data Sets or both. All this can be planned in advance.

Generation Data Groups

MVS and many other Operating Systems provide a powerful facility with JCL that allow the writing of JCL so that when a job is executed new data sets will be automatically created and cataloged.

Every time you need to write a new data set, you specify that you require a (+ generation), and the new data set will be automatically created. For example, when you have an Accounts Receivable application, which has a masterfile incorporated within that you wish to keep each time the Accounts Receivable job is run, you may call the data set:

AR.MASTER

When the main masterfile update program writes a new masterfile it calls the new data set:

AR.MASTER(+1)

JCL will automatically change the name to, for example:

AR.MASTER.G0019V00

The '019' is updated every time a new +1 data set is created with the next data set being:

AR.MASTER.G0020V00

However there are two main problems with this particular JCL facility:

1. The only way to stop the system from **CATALOG**ing the data set is to have the program **ABEND**.

2. Restarting is extremely difficult with JCL, because every time you refer to that data you must change the name to the G----V-- format.

Jol solves these problems by:

1. Allowing the User to **CATALOG** such a data set, after testing that the program has been executed correctly. For example:

```
EXEC        UPDATE
            AR.MASTER(+1);

IF UPDATE=0
THEN CATLG AR.MASTER(+1);
```

   With this method, you can control whether the data set is to be **CATALOG**ed or not.

   However, it should be remembered that you can execute or RUN more than one program. Then all the data sets can be **CATALOG**ed at the end of the job so that if a failure occurs, the entire job can be rerun from the beginning with no change to the Jol program.

2. Automatically resetting Generation numbers in Restart situations. For example, if 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 also allows you to define the data set in only one place, and have your programs refer to the definition.

This means that if the name defined has to be changed to the G----V-- format, the change has to occur only in the one place. You do not have to make sure that you have overridden every DD card referring to that data set.

Consider:

```
EXEC UPDATE    AR.MASTER(0),
               AR.MASTER(+1);

IF UPDATE = 0 THEN
DO;
     CATLG     AR.MASTER(+1);
PRINT:
     EXEC PRINT  AR.MASTER(+1);
END;
```

Now, assume that the **PRINT** program fails to execute properly, after the **CATALOG** instruction. It is not possible to merely resubmit the job, because then the system will rerun the first update program as well as the print.

You can say:

```
STARTAT PRINT;
```

and Jol will reset the generation number (+1) to (0) so that the job will execute correctly. This action is performed because the data set **AR.MASTER(+1)** would not have been created by the **UPDATE** step as it would have been by-passed by the **STARTAT** instruction.

Another method that could be used in similar circumstances is to use a Jol override instruction to manually change the (+1) data set to a (0) data set. For example, if the following is coded *before* the Jol program, the generations will be set to the correct values.

```
DCL NEWMAST DS AR.MASTER(0);

STARTAT PRINT;
```

The generation of **AR.MASTER** is set to relative generation zero (0) with the override statement and the job will commence at the **PRINT** step.

3. Allowing you to define the relative generations in Program Definitions, and allowing you to use a *Base* Index level or Generation Name. For example:

```
DCL INDEX DS DSN=AR.MASTER;

DCL UPDATE1 PROG
          INPUT READS INDEX(0)
          OUTPUT WRITES INDEX(+1);
```

```
    DCL UPDATE2 PROG
            INPUT READS INDEX(+1)
            OUTPUT WRITES INDEX(+2);

    DCL UPDATE3 PROG
            INPUT READS INDEX(+2)
            OUTPUT WRITES INDEX(3);

  ONE:  RUN UPDATE1;
            IF ONE=0
            THEN DO;
                CATLG INDEX(+1);
 TWO:   RUN UPDATE2;
                IF TWO=0
                THEN DO;
                    CATLG INDEX(+2);
 THREE:             RUN UPDATE3;
                        IF THREE=0
                        THEN CATLG INDEX(+3);
                END;
            END;
```

Now, if the job breaks down at level **THREE**, what has to be done is:

```
    DECLARE INDEX DS DSN=AR.MASTER(-2);

    STARTAT THREE;
```

and place the original Jol program immediately after. Jol will then calculate that (+2) is really (0) and that (+3) is really (+1).

SUMMARY OF Jol's INSTRUCTIONS

Jol provides Users with a wide range of short form instructions. Many of these instructions and commands have been defined through the use of Macro Command statements.

The specific number of Commands available depend on the Jol Release. In addition, each installation usually adds its own Commands. Hence, the following list of Commands is representative only.

ALLOCATE The **ALLOCATE** instruction dynamically allocates New or Old Data Sets for Input or Output in the Preprocessor Phase of Jol.

Allocated files can then be accessed with the **OPENFILE, GETFILE, READ, PUTFILE** and **WRITE** instructions. **ALLOCATE** can be used instead of **TSO ALLOCATE** or **JCL DD**cards to allocate data sets.

ASSIGN The **ASSIGN** instruction is used to alter the relative values of Symbolic Variables. By using this command it is possible to add, subtract, divide or multiply the original value of Symbolic Variables.

ASM The **ASM** command compiles Assembler Source programs. The Object code is then made into an executable program with the **LINK** Command, and then can be executed with a **RUN** or **EXEC** statement. For example:

```
ASM SOURCE(PAYROL01) ;
```

BUILDGDG This command creates an index in the system catalog for generation data sets. Examples:

```
BUILDGDG PAYROLL.MASTER,ENTRIES=5 ;
```

or

```
BUILDGDG PAYROLL.MASTER,NEW.MASTER,
                        ENTRIES=3 ;
```

BUILDJOB The **BUILDJOB** command allows the User to create Jol programs directly from the console. The User simply types in the name of the programs, the data sets they require, and the actions the programs perform on the data sets. The **BUILDJOB** command then creates Jol statements required for the job.

CALL The **CALL** instruction loads and executes programs *immediately* in the Preprocessor Phase of Jol.

CATALOG The **CATALOG** instruction enters either individual or groups of data sets into the system catalog.

CLOSE and CLOSFILE The **CLOSFILE** instruction closes a file previously opened with the **OPENFILE** instruction.

COBOL The **COBOL** command compiles Cobol Source programs. The Object code is then made into an executable program with the **LINK** Command, and then can be executed with a **RUN** or **EXEC** statement. For example:

```
COBOL PAYROL01 ;
```

COMPARE This command compares either two Sequential Data Sets or two Partitioned Data Sets. Examples:

```
COMPARE NEW.PROCLIB
        TO OLD.PROCLIB PDS ;
```

or

```
COMPARE MASTER.FILE(0)
        TO MASTER.FILE(-1) ;
```

COMPILE The **COMPILE** Command compiles Source programs into Object code. The Object code is then made into an executable program with the **LINK** Command, and then can be executed with a **RUN** statement. Examples:

```
COMPILE PAYROL01 ;
```

or

```
COMPILE AND LINK PAYROL02 ;
```

COMPRESS This command re-organizes (removes unused space) from a library or Partitioned Data Set. Examples:

```
COMPRESS SYS1.PROCLIB ;
```

or

```
COMPRESS SYS1.PROCLIB, SYS1.MACLIB,
         SYS1.CMDLIB ;
```

COPY The **COPY** Command will copy a Sequential, Indexed, VSAM or Partitioned Data Set to another. If required, only selected members can be copied from a data set. Examples:

```
COPY SEQ1 TO SEQ2 ;
```

or

```
COPY PDS1 TO PDS2
     SELECTING (MEM1,M3M2)
     OLD ;
```

or

```
COPY VSAM1 TO VSAM2 ;
```

DECLARE/ DEFINE

The **DECLARE** or **DEFINE** statements define variables to be used by Jol instructions. The following items can be *declared or defined:*

- Symbolic Variables
- Programs
- Data Sets
- Card Image Files
- Printer Files
- Punched Output Files

Other Jol instructions then use the declared items in the same manner as other high-level languages.

DELETE

The **DELETE** instruction combines the functions of the **SCRATCH** and **UNCATALOG** instructions. The **DELETE** instruction removes the name of the data set from the system catalog and frees the space assigned to the data set.

```
DELETE OUTPUT;
```

DISPLAY

Prints a message on the job's system log. The message is not displayed on the operator's console. For example:

```
IF %START= 'SORT10' THEN
   DISPLAY 'RESTARTING AT %START' ;
```

DO

The **DO** instruction is used in conjunction with an **IF** statement to execute parts of a job, only if predefined criteria have been fulfilled. The **DO** instruction is always terminated by an **END** statement.

DUMPVOL

The **DUMPVOL** Command will create a Tape copy of a direct access volume. The **RESTORE** Command will restore the volume.

EDIT

The **EDIT** instruction allows the editing of any Symbolic Variable according to any FORMAT list specified, allowing additional short form formatting to be carried out.

EXEC

The **EXEC** command executes registered programs. When a program is registered, Jol knows the filenames or ddnames used by the program, and whether they are used for input or output. To **EXEC** such a program, simply specify the names of the data sets to be used. For example:

```
EXEC  UPDATE          /* Program Name */
         PAY.MASTER(0)
         PAY.MASTER(+1)
         ;
```

EXIT

The **EXIT** instruction allows an abnormal exit from Jol and does not generate any instructions.

EXTEND

The **EXTEND** command adds data from one file to the end of another. For example:

```
EXTEND ACCOUNTS.TRANS.ACTION
         WITH NEW.TRANS.ACTIONS;
```

FORT

The **FORT** command compiles Fortran Source programs. The Object code is then made into an executable program with the **LINK** Command, and then can be executed with a **RUN** or **EXEC** statement. For example:

```
FORT SOURCE(PAYROL01) ;
```

FREE

The **FREE** instruction frees data sets or files so they can be re-used.

FS

FS turns the interactive Jol Panel instruction on and off. When ON, Jol will provide interactive assistance to the User; when OFF, Jol operates in Command Mode.

GET

The **GET** instruction provides the facility for the examination of a data set in the Preprocessor Phase of Jol. You can **GET** an entire file, or have Jol search a file (or files) and only return those parts corresponding to specified KEYS.

GETFILE

The **GETFILE** instruction reads a record from a file previously opened with the **OPENFILE** instruction.

IF

The **IF** statement is used to test return codes from programs, test if errors occured and other conditions. The **IF** instruction is frequently used in conjunction with a **DO** statement to execute parts of a job, only if predefined criteria have been fulfilled.
Examples of the **IF:**

```
IF VALIDATE<8
THEN RUN UPDATE;
```

or

```
IF ERROR   /* Abend ? */
THEN DO;
        TYPE 'CRITICAL ERROR IN ACCOUNTS';
        SUBMIT RESTORE;
END;
```

INCLUDE

The **INCLUDE** instruction allows you to include text into your program from a library or Partitioned Data Set.

INVOKE

The **INVOKE** instruction causes User written programs to be loaded and executed. The results produced by the Invoked Routine(s) can be Jol Source Statements. Jol will then interpret these as if they had been part of the original input stream. The invoked program can open its own files and perform any normal program functions.

JOBCAT

The **JOBCAT** instruction defines a private catalog to be used for the job, instead of the default system catalog. A **STEPCAT** can be used to temporarily override the **JOBCAT** or default catalog. For example:

```
JOBCAT PRIVATE.CATALOG1
       || PRIVATE.CATALOG2;
```

JOBLIB

The **JOBLIB** instruction defines a private catalog to be used for the job, instead of the default system catalog. A **STEPLIB** can be used to temporarily override the **JOBLIB** or default catalog. For example:

```
JOBLIB PRIVATE.LINKLIB1
       || PRIVATE.LINKLIB2;
```

JOBPARM The **JOBPARM** instruction informs JES3 processing systems about particular requirements for your job. For example:

```
JOBPARM ELAP 2, COPIES 2;
```

JOLOPT The **JOLOPT** instruction re-specifies Jol Compiler processing options.

KEEP The **KEEP** instruction ensures that a new data set is retained after the termination of the job in which it was created.

```
KEEP NEW.MASTER;
```

LINK The **LINK** Command will create an executable load module or program from the object code produced by the **COMPILE** Command or produced from a previously linked load module or from both. The **RUN** Instruction can then be used to execute the program.

```
LINK UPDATE LOAD('TEST.LOAD');
```

LIST The **LIST** command is used to print the contents of a data set in **HEX** or **CHAR**acter format. If required, only parts of the data set can be printed.

```
LIST TEST.INPUT.FILE;
```

LISTCAT The **LISTCAT** Command is used to list the System Catalog in its entirety or list only specified NODEs.

```
LISTCAT LEVEL(PAYROLL.TEST.FILES);
```

MACRO The **MACRO** statement defines the start of code to be used as a new instruction or macro. Parameters to be used by the instruction, and defaults if any, are also defined in the **MACRO** statement.

MAIN The **MAIN** instruction informs JES processing systems about particular requirements for your job. For example:

```
MAIN LINES 20 CANCEL SYSTEM ANY;
```

MERGE The **MERGE** Command merges two or more sequenced data sets into one composite data set.

```
MERGE 'INPUT.FILE1'
      || 'INPUT.FILE2'
      TO OUTPUT
      FIELD=(10,10,CH,A);
```

PLI The **PLI** command compiles PL/1 Source programs. The Object code is then made into an executable program with the **LINK** Command, and then can be executed with a **RUN** or **EXEC** statement. For example:

```
PLI SOURCE(PAYROL01) ;
```

PRINT The **PRINT** command will print the contents of a data set in its entirety.

```
PRINT 'REJECT.TRANS.ACTIONS';
```

ON ERROR **ON ERROR** defines the type of system storage dump to be printed if a program Abends.

OPCNTL The **OPCNTL** instruction allows you to output a non-JCL control card for HASP, ASP and JES Systems.

OPENFILE **OPENFILE** opens a previous **ALLOCATE**d data set for Input or Output processing. **READ** and **WRITE** instructions then transfer the data from or to the files.

PANEL The **PANEL** instruction provides a powerful method to either:

Display screens of data on a Screen.

or

Input full screens of formatted data to Jol. Text is displayed on the screen with any defaults; answers or defaults are then used as input to Jol.

PUTFILE **PUTFILE** writes data from a symbolic variable to a file.

READ The **READ** instruction reads data into a Symbolic Variable from either a terminal, or from a file previously opened with **OPENFILE.**

REDO The **REDO** instruction allows a section of Jol code to be repeated. This is very helpful when used with the **PANEL** instruction to check for valid answers.

RENAME **RENAME** renames data sets, or members of partitioned data sets.

```
RENAME 'CORRECTED.MASTER.FILE'
       'ACCOUNTS.MASTER.G0115V00';
```

REGISTER The **REGISTER** instruction registers a program to Jol. When the program is used in a Jol job, only the name of the program needs to be entered. Jol then checks the Jol register for the relevant details about the program.

RESTORE The **RESTORE** Command is used to restore the contents of a direct access volume from a Tape created with the DUMP Command.

RETURN The **RETURN** instruction stops execution without an error message.

```
RETURN 'JOB SUCCESSFUL';
```

ROUTE The **ROUTE** instruction directs printed or punched output to a particular printer or punch. For example:

```
ROUTE PRINT TO RMT3;
```

RUN The **RUN** instruction executes a previously defined and named program.

```
RUN VALIDATE 'DAY=%DAY';
```

SAVESYMS **SAVESYMS** saves the names and values of Symbolic Variables into a member of a Partitioned Data Set. These names and values may be restored later, thus providing communication facilities between jobs or subsequent executions of the same job.

SCRATCH The **SCRATCH** instruction frees the space assigned to a data set.

```
SCRATCH 'PAYROLL.TEST.FILE';
```

SIGNAL The **SIGNAL** instruction displays an error or warning message.

```
SIGNAL 3,'ERROR FOUND, CONTINUE?';
```

SORT The **SORT** Command will re-sequence a data set according to format instructions. Examples:

```
SORT TEST.FILE1 TO TEST.FILE2
        FIELDS = (10,2,CH,A,25,2,FL,A) ;
```

or

```
SORT FILE1 TO FILE2 USING
        SYS1.SORTCARD(SORTO2) ;
```

STARTAT The **STARTAT** instruction allows you to commence execution of a job at any point other than the beginning of the Job. All instructions are effectively bypassed until the label referred to is found, and execution begins at this point. This has been designed to assist restarting failed applications, and an example of its use will be found in the section on **Restarts.**

STOP The **STOP** instruction completely aborts the job in hand.

STOP WHEN The **STOP WHEN** instruction stops a job executing when certain Return Codes or ranges of Return Codes are detected.

```
STOP WHEN ANY>16 | ANY=8;
```

SUBMIT

This command allows a submission of subsequent jobs at any point within the currently executing job, providing the ability to pass symbolic variables from job to job. Examples:

```
SUBMIT PAYROLL2,SYMS='symbolic-name-lis/' ;
```

or

```
SUBMIT USING SYS2.PAYROLL(PAYROLL2) ;
```

or

```
SUBMIT      '//...Job..."
'//JCL Code'
'//JCL Code'
'//JCL Code'  ;
```

TEXIST

TEXIST tests if a specified data set exists. This command is particularly useful for restarts or scheduling purposes.

```
TEXIST 'RESTART.FILE';
IF LASTCC=0 THEN ...
```

TYPE

The **TYPE** instruction places a message on the Operator's console.

```
TYPE 'ACCOUNTS RECEIVABLE PASSED RESTART POINT
     10';
```

UNCATALOG

The **UNCATALOG** instruction removes the name of the data set from the system catalog, but does not **SCRATCH** it.

```
UNCATALOG 'ACCOUNTS.TEST.FILE(-4)';
```

WRITE

WRITE copies data from Symbolic Variables to files opened by **OPENFILE** or to the terminal.As stated earlier, other Commands can be added at the discretion of the Installation through the use of the Macro Language.

USER EXITS

Jol offers exit routine control at specific points allowing you to stop violation of Installation Standards, protect sensitive data sets, and generally validate the JCL or J-Code before it is produced. You can, for example:

- Allow only specific account codes to access certain data sets, thus providing data set protection.
- Validate the job card information, for example, to check that the accounting information is valid.
- See how many tape drives, or disk drives, are being used in any program.
- Check that data set naming conventions are being followed.

These exits all access *pre-formatted* tables. All Symbolic Variables or parameters *are replaced* when these routines are given control. Simple Assembler Macros are provided for use at these points. For example, the JOLERR Macro will write an error message to the Jol error log, and the JOLSAVE and JOLRETN Macros will perform saving and returning for the User from a Dynamic Save Area.

The following exit routines are available:

UJA01JOB, UJA02PGM and **UJA03DS**

These are given control when an unknown symbol is found when processing Job, Program and Data Set definitions. One of the main uses for the job exit could be to allow the Programmer to code particular information without using the relevant keyword, while having the exit routine recognize that individual range of codes. For example, if the exit routine recognized all account codes starting with FC01, the ACCOUNT keyword need not be coded. Similiarly, if all Disk Volumes in your installation commence with 'DISK', then the exit **UJA03DS** could recognize DISK01 as a disk, and even supply the device type.

The following validation exits are also available:

UJU01JOB	:	validate Job Card information
UJU02PGM	:	validate Program information
UJU03DS	:	validate Data Set information
UJU05CAT	:	validate Catalog information
UJU06DEL	:	validate Delete information
UJU07KEE	:	validate Keep information
UJU08UNC	:	validate Uncatalog information
UJU09SCR	:	validate Scratch information

Suppose that you did not wish anyone other than a specific person to be permitted to **DELETE SYS1.LINKLIB.** It is a simple matter to examine the delete instruction at exit **UJU06DEL** and to check that either or both, the Programmer's name and account code are correct.

If the instruction is held to be invalid you can:

- Cancel the entire job.

- Delete the one instruction.
- Write a note to the system log with WTL.
- Write to the Jol error log.
- Any combination of the above.

Two other exits are available from the Compiler - **UJA98SET** and **UJA99SET**. These are given control before and after the Job Card has been generated, and are mainly used to assist in generating any **SETUP** cards that may be required.

Jol Concepts and Facilities Manual.

Readers Comment Form

This manual is part of the Jol library that serves as a reference source for Managers, Systems Analysts, Programmers and Operators. This form may be used to communicate your views about this publication.

OSCAR Pty Ltd may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use any information you supply.

Possible topics for comments are:

Clarity Accuracy Completeness Organization Legibility

If comments apply to a Selectable Unit, please supply the name of the Selectable Unit _____.

If you wish a reply, give your name and address

_____

_____

_____

_____

_____

Number of latest Newsletter associated with this publication: _____

Please send your comments to:

OSCAR Pty. Ltd.,
38 Kings Park Road,
West Perth,
AUSTRALIA, 6005.