Advanced Job Control Language

What is J.C.L.?

Every batch job must be accompanied by instructions for the Operating System to ensure that it is processed as required. These instructions are written in the JOB CONTROL LANGUAGE or JCL.

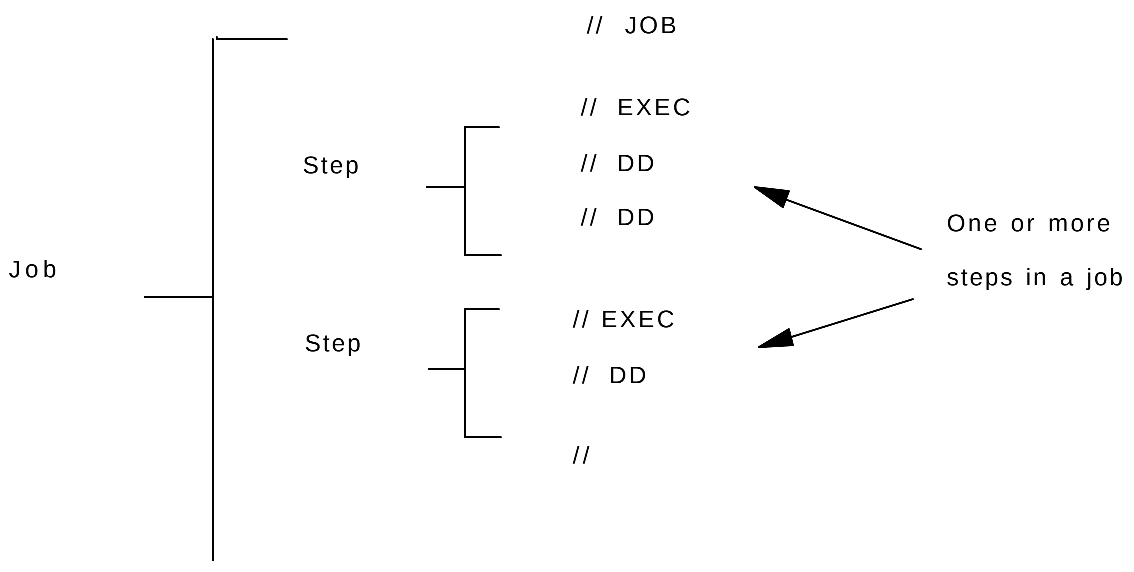
The JCL informs the Operating System which programs are to be executed within the job and which datasets are required for each program.

The JCL also uniquely identifies each job to the system through the JOB statement (see page 7).

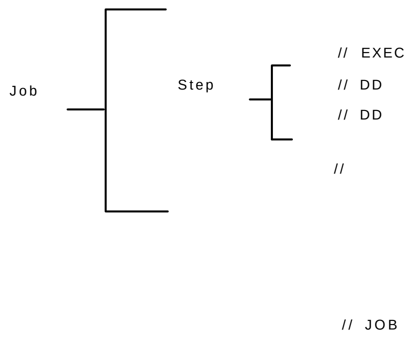
Running several programs simultaneously is a more efficient use of hardware than running one at a time but creates problems of resource allocation. The Operating System handles the allocation problem by determining the resource needs of each job from the JCL.

The Job Stream

A Job stream is the smallest unit of work that may be submitted in a batch environment. It is made up of one or more job steps but only one job identifier.



Separate jobs are completely independent:



However, execution of a **step** may be dependant on the results of a **preceding** step in the **same job.**

JCL Statements

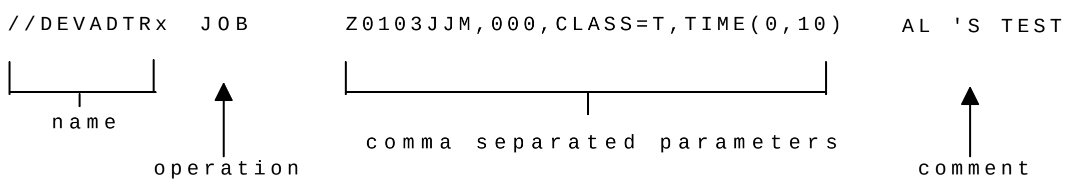
A job stream is made up of a series of statements. Each statement has the following format:

identifier   name   operation   parameters   comments

The identifier, name, operation, parameter and comment fields, if present, must be separated from each other by one or more spaces. These fields are coded in columns 1 through 71. Generally column 72 is not used. It can be used to indicate the continuation of a comment onto the next line. Columns 73-80 may only be used for comments, though comments can begin before column 72.

The **identifier** field occupies columns one and two and identifies the line as a JCL statement. It always contains //, except on a delimiter statement. Coding //* at the beginning of a line has the effect of treating the whole line as a comment.

Example:



```
//DEVADTRx  JOB    Z0103JJM,000,CLASS=T,TIME(0,10)    AL 'S TEST
```

The **name** field begins in column 3. A name can contain one to eight alphanumeric or national (#, @, $) characters. The first position **must** be alphabetic or national.

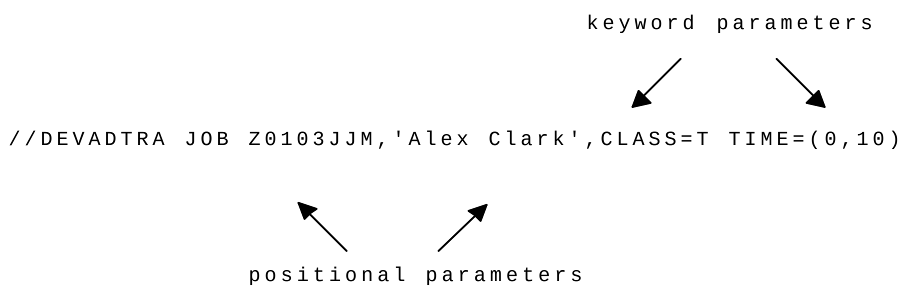
The operation identifies the type of JCL control statement. The following options will be described in more detail later in these notes: JOB, EXEC, DD, PROC, PEND and commands.

The **parameter** field consists of **positional** and **keyword** parameters separated by commas. No blanks are permitted unless they are part of a value enclosed in apostrophes. This field must start following the operation field or before column 16. It must be preceded and followed by at least one blank and cannot extend beyond column 71.

Positional parameters always come before keyword parameters and must always be in the same sequence. The absence of a positional parameter must be denoted by a comma if any other positional parameters follow.

Keyword parameters can appear anywhere within the parameter field and in any order, so long as they **follow** the positional parameters. Keyword parameters are characterised by a keyword followed by an equals sign (=) and variable information.

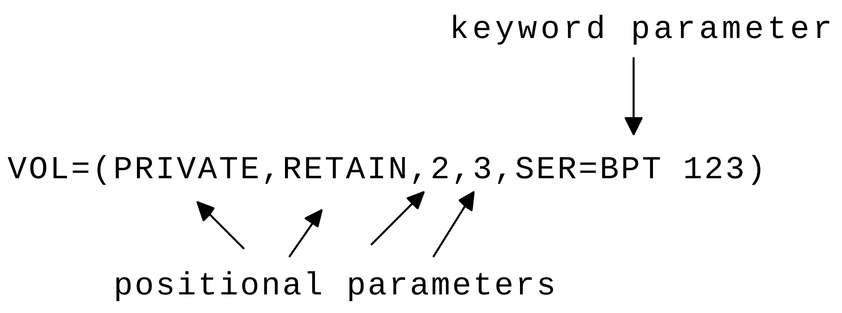
Example:



```
//DEVADTRA JOB Z0103JJM,'Alex Clark',CLASS=T TIME=(0,10)
```

Often, a parameter will contain multiple **sub-parameters**. These sub-parameters are separated by commas. They can also be positional or keyword and follow the same sequencing rules. Multiple sub-parameters must be enclosed in parentheses, unless there is only one sub-parameter and it is not preceded by any commas.

Example from the DD statement -



```
VOL=(PRIVATE,RETAIN,2,3,SER=BPT 123)
```

A statement may be continued onto the next and subsequent lines by following these rules:

1) Split the parameters <u>after</u> a comma.

2) Code // at the beginning of the next line.

3) Continue the parameters starting between **columns 4 and 16**. If the coding begins after column 16 the system will treat it as a comment.

Example:

```
//INITEM DD DSN=DEVADTR.ITEMFILE,DISP=(NEW,CATLG),
//     UNIT=SYSDU,SPACE=(TRKS,(1<1),RLSE),
//     DCB=(RECFM=FB,LRECL=80,BLKSIZE=23440)


                                                      note comma
continue between columns 4 and 16
```

The **comment** field may be coded anywhere following the parameter field. It must be separated from it by at least one blank. The comment field cannot be continued to another JCL statement.

JCL Operations

The following operations will be described in more detail:

JOB	assigns a name to the job and indicates its beginning.
EXEC	indicates the beginning of a job step, assigns a name to that step and identifies the program or procedure to be executed.
DD	assigns a dataset (file) and describes its attributes.
OUTPUT	specifies options for printing output datasets. It may be coded after the JOB or EXEC statements.
PROC	assigns default values to symbolic parameters in a catalogued procedure. For in-stream procedures this statement indicates the beginning of the procedure.
PEND	indicates the end of an in-stream procedure.

The following are used in conjunction with JCL operations:

delimiters	denote the end of an instream dataset. The default delimiter statement is /*. A delimiter is required when using DD DATA but is optional with DD *.
comments	can appear anywhere within a job following the JOB statement but before the null or terminating statement. The are denoted by //*.
null	A null statement (//) is an optional statement used as the last line in a job. It marks the end of the job.
commands	may be included in the job stream to control certain aspects of your job.

The JOB Statement

Format :

//jobname JOB account code,programmer name[,keyword parameters]...

jobname	This identifies the job stream to the operating system. For jobs submitted by programmers, this will be the programmer's logonid plus 1 alphanumeric or national character (e.g. DEVADTRP).
account code	The account code is used for charging purposes. A job will take a certain amount of time to execute; the amount of time taken is proportional to the cost charged. Boots is divided into a number of charge areas; ask your manager for your account code.
programmer name	The programmer's name is used for identifying jobs. It must be no more than 20 characters in length. If any spaces are to be included, the string must be enclosed in apostrophes.

The account code and programmer name are positional parameters. Following the programmer name one or more keyword parameters may be coded, in any order. Some of these are obligatory whilst others are optional.

CLASS=jobclass	This parameter must be coded. There are a large number of possible job classes which control what priority the executing job has and what devices are available to it; they are regularly amended.
TIME=(mins,secs)	Minutes must be in the range 0-1440, Seconds must be in the range 0-59. This parameter specifies the total CPU time allowed for the whole JOB. If this limit is exceeded the JOB will abend with a system completion code of 322. For daytime testing on the Boots mainframe TIME=(0,30) is usually sufficient but during training (0,10) is plenty of time.

MSGLEVEL=(statements,messages)

statements is either 0, 1 or 2, **messages** is either 0 or 1.

statements: 0 - only the job statement is printed.
1 - ALL JCL including catalogued procedure statements is printed.
2 - only the input JCL is printed. (default)

messages: 0 - dataset disposition/allocation messages are not printed unless the job abends.
1 - all dataset disposition/allocation messages are printed (default).

TYPRUN=SCAN Requests that the JOB be scanned for JCL syntax errors but not run.

TYPRUN=HOLD Used to put a job on to the job queue but to prevent execution until the operator releases the job, e.g. while a tape needed by the program is delivered to the computer room.

MSGCLASS=sysoutclass

Routes job output to a specified print class. Normally 6 for hold, Q for not hold; other classes exist for other purposes.

NOTIFY=identifier This keyword allows you to request that the system notifies you when your job has completed executing. You may either request to be notified in a TSO session (in which case identifier should correspond to your logonid) or in a ROSCOE session (identifier should be ROSnppp where n is the ROSCOE system number and ppp your ROSCOE prefix).

The JOBPARM Statement

This statement is a JES (Job Entry Subsystem) command. It allows you to specify printed output characteristics. It should be coded immediately after the JOB statement.

Format:

```
/*JOBPARM [keyword parameters,]...
```

B/BURST=option	Historically, was used with fanfold paper and dot-matrix printers. Now an optional parameter. Enter B=Y if you want your output on separate sheets or B=N for fanfold output.
R/ROOM=room	This will be the room name that your output will be sent to. It will appear in JES job separator pages to aid distribution of printouts .
T/TIME=nnnn	An estimation of the Jobs' execution time in minutes of real time, (as opposed to CPU time). Optional.
L/LINES=nnnn	Estimated number of lines of output to be produced by the job, in thousands. Optional.
K/LINECT=nnn	Number of lines of output to appear on each page.
N/COPIES=nnn	Number of copies of the output to be produced.

Example:

```
/*JOBPARM B=Y,R=WRLD,N=3
```

This will produce 3 copies of the job output. It will be burst and directed to room WRLD.

EXERCISE 1

Write the JCL necessary to produce a jobcard (collective expression for the JOB and associated JOBPARM statements) with the following attributes:

i) The jobname should be your logonid plus a character of your choice.

ii) Include your accounting code, your name and a telephone number where you can be contacted.

iii) Select a job class appropriate for a job classified as DB2D demand batch

iv) Force the job to abend if it takes longer than 45 seconds of CPU time to complete.

v) Select a MSGLEVEL that will provide as much information as possible about the job.

vi) Route the job output to a held output class.

vii) Ensure that the job output is directed to room TRNG and is burst.

The EXEC Statement

The EXEC statement specifies that an entity is to be executed in the course of the job stream.

Format:

```
                          { PGM=program }
//stepname EXEC           {  procedure  }   [,keyword parameters]...
```

stepname	Optional, but recommended particularly when there are quite a number of steps within a job. The stepname should be unique within the job, meaningful and no more than 8 characters. The first character of the name must be alphabetic.
PGM=program	specifies the program to be executed. If the program being executed does not reside in SYS1.LINKLIB/2/3 then a STEPLIB DD statement must be coded to inform the system which library to search for the program (see 4.3.1 Special DDnames).
procedure	specifies the name of a catalogued or instream procedure to be used (see page 31).

Any keyword parameters that are being used on the EXEC statement must be coded after the procedure name or PGM=program.

COND=(code,operator,stepname)	For information regarding COND, see page 34.
PARM=`parm...'	For PARM see page 37 .

Examples:

```
//STEP10   EXEC PGM=MSE460A
```

MSE460A is the program name

```
//STEPCOB  EXEC COB2
```

COB2 is the name of a catalogued procedure

Dummy EXEC steps

It is sometimes useful to include in a job stream a step that executes a program that does nothing. For example, suppose you wanted to submit a job that defined a dataset but did nothing else. Because every job must include at least one step, you need to execute a program that does nothing, as the dataset is defined using the DD statement. The program **IEFBR14** is available as a "null" program, and is often used in this way:

```
//DEFSTEP  EXEC PGM=IEFBR14
//DEFDSN   DD DSN=dsname ...
```

IEFBR14 will always return a condition code of zero.

EXERCISE 2

For this exercise, it will be necessary to consult the green **Programming Reference Manual** and to use the ROSCOE RPF CATPROC.

i) What is the name of the program executed by the COPYSEQ catalogued procedure?

ii) List three dataset comparison utility programs available at Boots.

iii) What is the name of the program which invokes the linkage editor (hint: examine the LET catalogued procedure)?

iv) Name two catalogued procedures for compiling VS COBOL II programs.

v) What is the function of the TEST catalogued procedure?

Special DDnames

These have particular meanings for the Operating System and should not be used for any other purpose.

```
        //STEPLIB   DD DSN=steplibrary,DISP=SHR
```

The system will search the **steplibrary** for the program referenced in the associated EXEC statement. If the program is not there it will search SYS1.LINKLIB/2/3. These three libraries are known as the linklist and contain programs which are most commonly used, such as the utility programs ICEGENER and IEBCOPY.

Only one STEPLIB is permitted per step but libraries may be concatenated (see page 27); this statement, if present, usually comes immediately after the EXEC statement but this is not essential.

If a program abends, it is possible that it may **dump**. This is a process whereby diagnostic information relating to the abend is written by the operating system to an output dataset. There are three special DDnames available that each produce different dump formats:

//SYSABEND	produces a formatted dump; this will include the system nucleus and the executing program's region. This type of dump should not usually be requested .
//SYSMDUMP	This produces a dump containing similar information to SYSABEND but in a machine-readable format. Again, this should not usually be requested .
//SYSUDUMP	This is a formatted dump containing selected system control blocks and the executing program's region. This is the usual type of dump to request • .

The DD Statement

The DD statement forms the link between the logical file referred to by the program and the physical dataset on disk or tape. It has the general format:

```
//ddname  DD [positional parameter][,keyword parameter]...
```

ddname	This **must** be present (except for the second and subsequent data set in a concatenation). The name must be unique within the step. A maximum of 255 ddnames per step is allowed.

The ddname in the JCL is linked to the files referred to in a program. The way in which this happens varies according to the programming language. For example, in COBOL programs, association is provided by using the SELECT clause of the FILE-CONTROL paragraph:

```
SELECT filename ASSIGN UT-S-ddname
```

The filename in the SELECT statement is defined via the FD statement of the FILE SECTION, within the DATA DIVISION.

In the EASYTRIEVE PLUS language, the JCL ddname is associated with program files using the FILE statement. Thus:

```
FILE ddname
```

Usually, a dataset will reside in one of three locations:

. inline, that is, it is included as part of the job stream;

. on disk

. on tape.

In addition, it is possible to specify a null or dummy dataset. This is often useful for testing purposes.

Specifying an Inline dataset

An inline dataset is treated by a program as if it were a normal dataset with records 80 bytes long (datasets with a record length of 80 bytes are sometimes referred to as **card image** datasets; this is because punched cards used to be 80 characters wide). There are two different methods of specifying an inline dataset:

```
//ddname  DD *
data ...
data ...
```

The end of the dataset is indicated by a delimiter statement (/*) or **any other JCL statement**. As a result, the inline dataset cannot include JCL. This problem can be avoided by coding:

```
//ddname  DD DATA
```

The end of the dataset is indicated only by /*. This allows JCL to be read as data. If you wish to treat a /* as data then a two-character delimiter must be specified in place of /* thus:

```
//ddname  DD DATA,DLM=ZZ
```

The end of the data would then be indicated by a line containing ZZ in the first two columns.

Specifying a dataset on Disk

JCL allows you to either refer to an existing dataset or create a new one. In addition, it is possible to deal with either permanent or temporary datasets. A temporary dataset is one that is created during the course of a job stream and automatically deleted at the end of it. To refer to an existing dataset, the general format of the JCL statement is:

```
//ddname  DD DSN=dsname,DISP=disposition
```

The **ddname** links a physical dataset to the entity referenced in a program. The two **keyword parameters, DSN and DISP**, specify what the dataset is called and how it is to be treated by the job stream. DSN stands for **D**ata **S**et **N**ame. Every dataset used by a program has a name. This name is assigned according to certain rules laid out in the Boots **Programming Standards and Guidelines**, Section 2 (see also the Basic JCL notes). There are various different types of dataset; the JCL to be coded for each is described below. The DISP parameter affects how the dataset is going to be treated by the job stream. When dealing with existing datasets, it will usually have one of three values:

OLD:	The dataset existed before this step and is to be used exclusively by this job (n.b. only one job can have exclusive access to a dataset at any one time).
SHR:	The dataset existed before this step and will be used for INPUT ONLY; therefore other jobs can access it at the same time.
MOD:	For OUTPUT only. If the dataset already exists then output from this step will be added to the end of the existing dataset.

Some examples.

To reference a specific member of a partitioned dataset for a read operation:

```
//ddname  DD DSN=dsname(member),DISP=SHR
```

To reference a dataset exclusively during a job:

```
//ddname  DD DSN=dsname,DISP=OLD
```

To write to the end of a temporary dataset:

```
//ddname  DD DSN=&&dsname,DISP=MOD
```

☞ a temporary dataset is prefixed by two ampersands.

If the dataset which is referenced by the DD statement does not already exist, it will be created. If this is to be done successfully, the operating system needs a number of items of information: where is the dataset to be located? how big is the dataset to be? what attributes should the dataset possess? what is to happen to the dataset at the end of the jobstream?. The JCL required to answer those questions is:

```
//ddname  DD DSN=dsname,DISP=(status,if-OK,if-abend),
//        UNIT=disk,
//        SPACE=(unit,(pr,sec[,dir])[,RLSE,CONTIG,ROUND]),
//        DCB=(LRECL=lngth,RECFM=recform,BLKSIZE=blk)
```

If a dataset is being created, the DISP status keyword parameter should be **NEW**. However, there are two additional sub-parameters that may be coded with DISP to control what happens to the dataset when the job stream completes, as status refers to the current status of the dataset at the start of the job step. **If-OK** specifies what is to happen to the dataset if the job step completes normally. There are 5 options:

PASS:	The dataset is to be passed on to a following step in the same job (the system will keep track of where the dataset is for the duration of the job).
KEEP:	The dataset is to be kept at the end of this step.
CATLG:	Same as KEEP, but an entry is also created in the catalogue for ease of access at some later date.
UNCATLG:	Same as KEEP. Only applies to already catalogued datasets; the catalogue entry that was created for this dataset is to be removed. This option should NOT usually be used at Boots as it is difficult to keep track of uncatalogued datasets.
DELETE:	The dataset is to be deleted at the end of this step. If the dataset was located via the catalogue, the catalogue entry will also be removed.

If **if-OK** is not specified, or only PASS is specified:

KEEP is assumed for a dataset that existed before this job;
DELETE is assumed for a dataset that did not exist before the job.

If-abend specifies what is to be done with the dataset if the job step terminates abnormally. There are four options: KEEP, CATLG, UNCATLG and DELETE. The meanings of these options are the same as for normal disposition. If a step terminates abnormally and no conditional disposition is specified then the normal disposition is used, except where that normal disposition is PASS. In that case, if the dataset existed before the job then it is KEPT. If the dataset did not exist before the job then it is DELETED .

Some examples:

```
//        DISP=(,PASS)
```

This is the usual way to reference a new, temporary dataset. The status parameter is allowed to default to NEW; the dataset is PASSed to the next job step. DELETE is not explicitly coded anywhere as temporary datasets are deleted by the operating system at the completion of a job.

```
//        DISP=(,CATLG,DELETE)
```

This is the usual disposition to employ when creating a permanent dataset. ALL permanent datasets **must** be catalogued; the DELETE parameter ensures that if the job step abends, the dataset will be deleted; this allows the job stream to be re-run with the minimum of preparation.

The **UNIT** keyword parameter tells the operating system which disk to put the dataset on. At Boots, disks are grouped into **pools**. The pool which development staff should use is **SYSDU**. It is possible to specify which particular disk, rather than which disk pool, should be used by replacing with UNIT=unit type and VOL=SER=diskname (where unit type might be 3380, 3390 etc.); this option should not usually be used. Thus, you should code:

```
//        UNIT=SYSDU
```

The **SPACE** parameter defines how much space is to be reserved for the dataset on the disk. It has a number of sub-parameters:

```
//        SPACE=(unit,(pr,sec[,dir])[,RLSE,CONTIG,ROUND]),
```

Unit defines what the unit of allocation will be. It can be expressed in bytes - the average blocksize of the dataset; tracks - TRK; or cylinders - CYL. The number of bytes in a track depends on the model of disk unit that is being used; a typical value would be 47,476. There are 15 tracks in a cylinder.

The **pr** sub-parameter specifies the number of units to be allocated initially, whereas **sec** defines the number of units to be allocated when the primary allocation runs out. The secondary amount will be allocated up to 15 times if necessary.

If a partitioned dataset is being created, space will need to be allocated to hold a directory listing of the members in the dataset. This space is specified with the **dir** sub-parameter.

The remaining sub-parameters are optional: **RLSE** specifies that when the dataset is closed or released by the job stream, any unused space should be released. **CONTIG** specifies that the primary allocation must be contiguous space on the disk. This option will only be required for certain special types of dataset. **ROUND** - when allocating in blocks round up to the nearest cylinder.

Some examples. The space required for a partitioned dataset:

```
//        SPACE=(TRK,(10,1,10))
```

The dataset will be created with 10 tracks of space but this can potentially increase to 25 tracks (10+(1*15)). Ten blocks are reserved for the dataset directory.

The space required for a non-partitioned sequential dataset:

```
//        SPACE=(CYL,(50),RLSE)
```

The largest dataset that a developer is allowed to allocate is one that extends over 50 cylinders.

The final parameter necessary to define a new dataset is the DCB keyword parameter. It has the general format:

```
//        DCB=(LRECL=lngth,RECFM=recform,BLKSIZE=blk)
```

The **LRECL** sub-parameter specifies the record length of the dataset; the **RECFM** parameter gives the format of the dataset (e.g. whether records have fixed or variable length); and **BLKSIZE** defines the blocksize of the dataset. This should be a multiple of the record length; the optimum to aim for is known as **half-track** blocking - that is, there should be two blocks per track. If a track consists of 47,476 bytes, the optimum blocksize to aim for is 23738. The ROSCOE RPF **BLOCK** will calculate the optimum blocksize for a given record length.

Some examples.

A dataset with 80 byte, fixed length, blocked records:

```
//        DCB=(LRECL=80,RECFM=FB,BLKSIZE=23440)
```

A dataset with 124 byte, variable length records:

```
//        DCB=(LRECL=124,RECFM=VB,BLKSIZE=23440)
```

the blocksize includes the 4 byte record-descriptor incorporated into variable length records.

To summarise: the DD statement required to create a dataset with 80 byte, fixed length records on disk pool SYSDU -

```
//ddname  DD DSN=dsname,DISP=(NEW,CATLG),
//        UNIT=SYSDU,
//        SPACE=(TRK,(10,1),RLSE),
//        DCB=(LRECL=80,RECFM=FB,BLKSIZE=23440)
```

The dataset will be allocated up to 25 tracks of space. Any unused tracks will be freed.

A temporary dataset with 40 byte, variable length records requiring 25 cylinders of space:

```
//ddname  DD DSN=&&dsname,DISP=(,PASS),
//        UNIT=SYSDU,
//        SPACE=(CYL,(25)),
//        DCB=(LRECL=40,RECFM=VB,BLKSIZE=23444)
```

EXERCISE 3

i) Code the DD statement required to assign the existing dataset SYS1.RUNTIME to the ddname STEPLIB such that other users will be able to use the dataset at the same time.

ii) Code the DD statement to create a temporary dataset called TEMP. Assign it to the ddname OUTLIB; it should have variable length records of 56 bytes in length. Make sure that the dataset resides on the SYSDU pool. The dataset will need to contain 1,240,000 bytes; express the space required to contain this in tracks, allowing up to 10% of the primary space allocation as a secondary allocation.

iii) Code the DD statement required to create a partitioned dataset. It should have 80 byte, fixed length records. Allocate 50 directory blocks; the dataset requires 15 cylinders of space. Direct to the SYSDU pool. The dataset should be assigned to the ddname LIBRARY and should have a name of userid.ROSLIB.RETAIN.

Specifying a dataset on Tape

The use of tapes by development staff is being phased out, as tape drives are regarded as obsolete. Moreover, the operating system has been configured such that development staff are actually prevented from using tapes.

Specifying Dummy datasets

It is often useful to allocate a dummy dataset to a ddname. This can be done in one of two ways:

```
          //ddname  DD DUMMY
     or
          //ddname  DD DSN=NULLFILE
```

This technique will work only for sequential datasets; no physical device allocations or I-O operations are performed. For input operations, the first READ or GET will cause an end-of-file condition to be set; for output, all WRITEs or PUTs will function normally as far as the program is concerned but no actual I-O takes place.

It will be necessary to code DCB=BLKSIZE= when using DUMMY for an output file otherwise the OPEN or WRITE may fail because the system does not have complete DCB information for the data set. An example:

```
     //FRED   DD  DUMMY,DCB=BLKSIZE=5000
```

Disposition messages

These are found in the SYSMSGS file of a batch job and describe the result of a DD operation. For example:

```
IEF285I  DEVSTTR.RDR          KEPT
IEF285I  VOL SER NOS=D33501
```

The possible dispositions are: KEPT, DELETED, PASSED, CATALOGED, UNCATALOGED,and RECATALOGED.

Occasionally, if a DD operation is unsuccessful, the message IEF283I may appear with:

```
NOT DELETED n
```

or message IEF287I with one of the following:

```
NOT CATLGD    n

NOT UNCATLGD  n

NOT RECATLGD  n
```

The commonest message that you will encounter is NOT CATLGD 2 which means that the dataset has not been catalogued because a dataset of the same name (on another disk) is already catalogued. Refer to the `VS2 SYSTEM MESSAGES' manual for further explanations.

Sources of DCB Information

Information about a dataset's attributes can come from one of three sources. They are the program referencing the dataset, the JCL DD statement and the dataset itself.

Program:	This has the highest priority; any attribute specified in the program cannot be overridden.
DD statement:	Any attributes specified here will **only** be used if the **program** has **not** specified a value.
Dataset:	For an existing dataset, any field which has not been specified by the program **or** the DD statement will be obtained from the dataset itself.

The resulting combination of attributes, after applying these rules, must be consistent.

Boots has a series of standards for DCB Information. For files acting as input to a program, the program may, optionally, specify RECFM and LRECL; BLKSIZE should not be specified. In COBOL programs, specify BLOCK CONTAINS 0 RECORDS in the FD. The DD statement should not normally specify any DCB information; most of the required information should be derived from the dataset itself.

For output datasets, the program should specify RECFM and LRECL but not BLKSIZE. In COBOL programs, code BLOCK CONTAINS 0 RECORDS. The BLKSIZE should be specified in the DD statement.

Using the Output Writer

A function often required by programs is to write output to the job spool, from where it can be printed. Output written to the spool can be browsed and manipulated online using either the ROSCOE Attach Job facility or the TSO SDSF program. The format of the statement to use the output writer is:

```
//ddname  DD SYSOUT=class[,OUTLIM=lines,DCB=parms]
```

If **class** = `*', the output class will default to the same as the MSGCLASS on the jobcard. Other values are possible but note that job output classes may vary. However, the following classes are unlikely to change: if class = `A' the output will be printed by the mainframe printer; if class = `6', the output will be held on the job spool until released via either ROSCOE or TSO.

The **OUTLIM** parameter specifies the maximum number of print records permitted. If the limit is exceeded the job abends with a 722 abend code.

If the program using the output writer does not specify a record format or record length then these will need to be specified in the JCL by using the DCB parameter.

An example:

```
//REPORT1 DD SYSOUT=*,OUTLIM=1000
```

This statement will allow a program to write up to 1000 lines of output to the ddname REPORT1.

Concatenating Datasets

Up to 255 sequential datasets may be concatenated and treated as a single dataset, provided:-

a) all of the datasets are on the same device type

b) all of the datasets have identical DCB attributes (however, different block sizes are allowed, provided datasets are concatenated with largest BLKSIZE first)

c) they are used for input only.

DUMMY or DSN=NULLFILE may only appear as the last entry in a concatenation, since it forces an immediate end-of-file condition on reading.

To concatenate datasets, omit the ddname from the second and subsequent DD statements:

```
//ddname  DD DSN=dataset1,.....
//        DD DSN=dataset2,.....
//        DD DSN=dataset3,.....
```

When the file is opened, the program will start reading at the beginning of the first dataset; end-of-file will be indicated by the operating system only when the end of the last dataset is reached.

Up to 16 partitioned datasets may be concatenated and treated as a single partitioned dataset, subject to the same restrictions.

DD Statement Summary

Certain information must be given to the Operating System when datasets are being created or accessed. The table below summarises what information must be included in the JCL when dealing with datasets on disks.

Information Required	**Parameter to use on DD statement**	**Creating a dataset**	**Retrieving a dataset**
Dataset name	DSN	✓	✓
Dataset location	UNIT (also VOL)	✓	X
Dataset size	SPACE	✓	X
Dataset attributes	DCB	✓	X
Dataset status	DISP	✓	✓

EXERCISE 4

i) A program is to be tested by diverting the output to a dummy dataset. It is a COBOL program; the record length has been specified as 94 bytes and the clause BLOCK CONTAINS 0 RECORDS has been coded in the FD paragraph. Code the appropriate JCL for the desired output dataset.

ii) A COBOL program contains the following code:

```
SELECT GRANTEE-FILE ASSIGN TO UT-S-GRANTEE.

FD  GRANTEE-FILE
LABEL RECORDS ARE STANDARD
BLOCK CONTAINS 0 RECORDS
RECORDING MODE IS F.

01  GRANTEE-RECORD.
     05  IN-GRANTEE-ID   PIC X(8).
     05  IN-GRANTEE-DESC PIC X(30).
```

The following JCL has been coded:

```
//GRANTEE  DD  DSN=DEVADTR.GRANTEE,DISP=MOD,
//         DCB=(LRECL=40,BLKSIZE=23440)
```

The dataset DEVADTR.GRANTEE has the following characteristics: it has fixed length records of 38 bytes in length and a blocksize of 3800.

What are the consolidated DCB attributes applicable in this case?

iii) Code the JCL necessary to divert the ddname RPT1 to the spool. Cause an abend to occur if more than 500 lines of output are produced.

iv) A program to be executed might reside in one of three libraries: SYS1.RUNTIME, DEV@@TR.TEST.LOADLIB, UB.TSS.DEV.LOADLIB. It is possible that more than one version of the program exists in these libraries. The most up-to-date version will be in DEV@@TR.TEST.LOADLIB and the oldest in SYS1.RUNTIME. Code the DD statement that will search each of these libraries for the required program in the correct order.

The OUTPUT command

OUTPUT is a JES2 command, like JOBPARM (see page 9). It is used for manipulating and directing output that is to be printed. It has the general format:

$$/{*}\text{OUTPUT} \left\{ \begin{array}{c} \text{code} \\ * \end{array} \right\} \text{parameter [,parameter] ...}$$

Each OUTPUT statement has a unique 1-4 character identifying code. This allows several different OUTPUT statements to be used in the same jobstream. If an OUTPUT statement continues over more than one line, it is continued by coding an asterisk in column 10:

```
/*OUTPUT     code parameters ...
/*OUTPUT     * parameters ...
```

the continued line still starts with /*OUTPUT. The parameters available are:

B/BURST=	either **Y** for individual sheets or **N** for one continuous sheet. Historically, was used with fan-fold paper and dot-matrix printers. Now an optional parameter.
X/CHARS=	the character set to use. A number of different character sets are available at Boots: **APL** enables special characters to be printed; **BT12** is the standard set
N/COPIES=	enables multiple copies of a printout to be specified
D/DEST=	destination printer. To print on the mainframe printer, specify **LOCAL**; to print on a remote printer, specify either **Rnnnn** or **RMTnnnn** where nnnn is the remote printer number
K/LINECT=	number of lines to be printed before issuing a page throw.

An OUTPUT command is invoked by referring to it on a SYSOUT statement:

```
//ddname  DD SYSOUT=(class,,code)
```

where **class** is the output class and code refers to the /*OUTPUT statement. An example:

```
/*OUTPUT OUTA DEST=R672
//REPORT1  DD SYSOUT=(*,,OUTA)
```

Output from REPORT1 will go to remote printer R672.

The PROC and PEND statements

PROC and PEND are used to build up JCL "programs" called procedures. Procedures are not true programs; they are simply collections of JCL statements. However, they do allow variable substitution to take place and hence can be made general purpose. Procedures are invoked using the EXEC command (see page 11).

Procedures can exist in one of two places: either they are **instream procedures** and included as part of a jobstream, or they are saved in a dataset, in which case they are known as **catalogued procedures** or **catprocs**. It is more usual to use catprocs rather than instream procedures.

Catprocs exist in specific libraries which the operating system is aware of. A list of the current libraries may be obtained in ROSCOE by typing **help catproc**. The libraries which developers within B.T.C. are likely to come across include:

SYS1.TESTPROC	used by development staff for their own catprocs
SYSJ.QT.PROCLIB	catprocs commonly used by development staff, for example, to compile COBOL programs
SYS1.PROCLIB	catprocs used by the operating system and other system software.

The general format of a procedure is:

```
//name  PROC [symbolic parameter, ...]
//  JCL statement ...
//  JCL statement ...
//      [PEND]
```

In catprocs, the **PEND** statement is generally omitted; it is necessary in instream procedures to separate the procedure from the rest of the JCL. **Symbolic parameters** are variables used within a catproc. They are defined at the start of the procedure as part of the PROC statement. Any parameter thus defined can then be referenced in the JCL statements which make up the procedure. They appear as their name preceded by an ampersand (&). This is replaced in the JCL that is executed by the value assigned to the variable.

It is not possible to nest PROC statements.

Consider the following catproc, TEST, which resides on SYSJ.QT.PROCLIB:

```
//TEST   PROC LIB='&&EXECLIB',MEMB=PROG,OUT=1000,SYSOUT='*'
//*********************************************************************
//*   TEST - Execute a COBOL program                                  *
//*   Parms: LIB - library program is to be executed from           *
//*          MEMB - program name                                      *
//*          OUT - maximum number of output lines; default = 1000     *
//*          SYSOUT - destination - default is assumed ('*')          *
//*   New Version, Alex Clark, Quality & Training, 19 Nov 1991        *
//*   Version B  ADC  28/11/1991 Add SYSABOUT and SYSDBOUT cards      *
//*********************************************************************
//STEPC    EXEC PGM=&MEMB
//STEPLIB  DD  DSN=&LIB,DISP=(SHR,PASS)
//         DD  DSN=SYS1.RUNTIME,DISP=(SHR,PASS)
//SYSOUT   DD  SYSOUT=&SYSOUT,OUTLIM=&OUT
//SYSPRINT DD  SYSOUT=&SYSOUT,OUTLIM=&OUT
//SYSDBOUT DD  SYSOUT=&SYSOUT
//SYSABOUT DD  SYSOUT=&SYSOUT
//SYSUDUMP DD  SYSOUT=&SYSOUT
```

The function of the procedure is to execute a COBOL program. The following code could be used to invoke the procedure:

```
//STEP1   EXEC TEST,LIB=`DEV@@QT.TEST.LOADLIB',
//        MEMB=QTTST01
```

That would result in the following JCL being executed:

```
//STEPC    EXEC PGM=QTTST01
//STEPLIB  DD  DSN=DEV@@QT.TEST.LOADLIB,DISP=(SHR,PASS)
//         DD  DSN=SYS1.RUNTIME,DISP=(SHR,PASS)
//SYSOUT   DD  SYSOUT=*,OUTLIM=1000
//SYSPRINT DD  SYSOUT=*,OUTLIM=1000
//SYSDBOUT DD  SYSOUT=*
//SYSABOUT DD  SYSOUT=*
//SYSUDUMP DD  SYSOUT=*
```

any character which is not alphanumeric (A-Z,0-9) must be enclosed by apostrophes. In the above example, the LIB symbolic parameter was enclosed in apostrophes because it included full stops and @ symbols. Notice that the variables which were denoted by their names preceded by ampersands have been replaced with the values that were assigned to them; these assignments were either from the catproc itself, or from the EXEC statement. Any parameters passed via the EXEC statement take precedence over those in the procedure itself.

This is an example of an instream procedure:

```
//TEST    PROC LIB='&&EXECLIB',MEMB=PROG,OUT=1000,SYSOUT='*'
//STEPC     EXEC PGM=&MEMB
//STEPLIB   DD  DSN=&LIB,DISP=(SHR,PASS)
//          DD  DSN=SYS1.RUNTIME,DISP=(SHR,PASS)
//SYSOUT    DD  SYSOUT=&SYSOUT,OUTLIM=&OUT
//SYSPRINT DD  SYSOUT=&SYSOUT,OUTLIM=&OUT
//SYSDBOUT DD  SYSOUT=&SYSOUT
//SYSABOUT DD  SYSOUT=&SYSOUT
//SYSUDUMP DD  SYSOUT=&SYSOUT
//       PEND
//*
//RUN1  EXEC TEST,LIB='DEV@@QT.TEST.LOADLIB',MEMB=FIRSTRUN,SYSOUT=A
//RUN2  EXEC TEST,LIB='DEV@@QT.TEST.LOADLIB',MEMB=SECRUN
```

Instream procedures are often used to cut down the amount of JCL that needs to be written to perform a task. The above example reduces by about half the amount of JCL required to execute the two programs FIRSTRUN and SECRUN.

Sometimes, you might want to use a procedure but you want to change part of it. If it is not a symbolic parameter, you can **override** part of the JCL. The general format for overriding a statement is:

```
//stepname.name  [overriding statement]
```

Suppose that you wish to invoke the TEST catproc but wish to divert SYSOUT output to a dataset. That could be accomplished by coding:

```
//RUN   EXEC TEST,LIB=`DEV@@QT.TEST.LOADLIB',
//         MEMB=FIRSTRUN
//STEPC.SYSOUT DD DSN=DEVADTR.TEST.OUTPUT,DISP=MOD
```

This will result in the existing SYSOUT card in the STEPC step being replaced. Alternatively, additional parameters can be specified by using override statements.

the ROSCOE RPF catproc allows you to import a specified procedure into your AWS.

Making JCL Execution Conditional

Before a step in a job executes, the condition code(s) of previous step(s) can be tested. If the condition in the test is satisfied then the step will **not** be executed. This test may be performed as part of the EXEC statement. The general format is:

```
//stepname  EXEC PGM=pgm,COND=((code,operator,stepname)..)
```

Code is a number in the range 0 to 4095. It is derived from the completion code returned by a program in a preceding step.
Operator is one of the following: LT,LE,NE,EQ,GE,GT.
Stepname is the name of a preceding step within the job stream.

Up to 8 condition code tests are allowed on one EXEC statement. If any of the tests are satisfied then the step is bypassed.

For example:

```
COND=((4,LT,STEPA),(4,LT,STEPB))
```

means:- if 4 is less than the condition code set by STEPA or if 4 is less than the condition code set by STEPB, then bypass this step. In other words, the step will be bypassed if **either** STEPA or STEPB complete with a condition code of greater than four.

Usually, if a program abends, the rest of the job stream is not executed. However, as an alternative to one of the 8 tests `EVEN' or `ONLY' may be coded:

EVEN means execute the step even if a previous step has abended.

ONLY means execute the step only if a preceding step has abended.

If you code

```
COND=((4,LT,STEPB),EVEN))
```

and the first test is satisfied, the second test will not be considered so the step will be bypassed, regardless of the fact that EVEN is coded.

To test the condition code of a step within a catalogued procedure, code:

```
        COND=((code,operator,stepname.procedure-stepname),..)
```

Example:

```
        //STEP2   EXEC  COPYPROC
          ...
          ...
          ...
        //STEP3   EXEC PGM=DEMOPEN,COND=((0,NE,STEP2.COPY1),..)
```

where COPY1 is a stepname within the COPYPROC procedure.

COND=(n,operator) - applies a test to all previously executed steps.

Steps which are not executed never generate a condition code so subsequent tests for the result of the step will always be false.

It is also possible to test the execution of steps on the JOB statement. The general format is:

```
        COND=((code,operator),...)
```

where code and operator have the same function as described above.

As with the EXEC statement, up to 8 tests are allowed. All 8 tests are applied to the condition code set by each step in the job. If any one of the steps fulfils any of the tests then the rest of the job is skipped.

Example:

```
        COND=((8,LT),(4,EQ))
```

means if 8 is less than the condition code in any step or if 4 is equal to the condition code in any step then skip the rest of the job.

The COND parameter used on the EXEC statement is generally more useful than on the JOB statement.

Setting the Condition Code in a program

In COBOL, code

```
MOVE value TO RETURN-CODE.
```

In EASYTRIEVE PLUS, code

```
RETURN-CODE = value
```

Boots' standards (Programming Standards and Guidelines, Section 1) recommend that in ANY program, the return code should always be explicitly set to zero unless the program is terminating abnormally.

Passing Information to Programs

It is possible to pass information to programs via JCL. It is done via the EXEC statement, e.g.

```
//stepname EXEC PGM=pgm,PARM=value
```

where value may be up to 100 characters of information. As with symbolic parameters, if the information contains special characters it must be contained within apostrophes. For the special considerations which apply to the use of apostrophes and ampersands and when the value needs to be continued on another line see the OS/VS2 MVS JCL manual (1982) pp163.

Examples:

```
//   EXEC PGM=HEXCHAR,PARM=`START=20,END=40'

//   EXEC PRINFILE,DSN=`&&FRED',PARM=L100
```

Boots' standards recommend that you should **not** write programs which accept information as a parameter in this way. However, there are still a number of production programs which rely on this technique.

In COBOL programs, the root module of the program will contain a LINKAGE SECTION with the following:

```
01  PARM-INFO.

        03  PARM-LENGTH          PIC S9(4)   COMP.
        03  PARM-DATA            PIC X(100).
```

the length of the PARM-DATA string above is the maximum possible (100 bytes).

The PROCEDURE DIVISION clause will include the USING option:

```
PROCEDURE DIVISION USING PARM-INFO.
```

PARM-DATA will contain the characters specified in the PARM field of the EXEC statement. PARM-LENGTH will contain the length of the PARM field.

In EASYTRIEVE PLUS programs, code:-

```
PARM-INFO      W         102   A
          PARM-LEN   PARM-INFO    2   B
          PARM-DATA  PARM-INFO   +2 100   A
...
...
...
JOB ...
CALL EZTPX01 USING (PARM-REGISTER PARM-INFO)
```

The information from the PARM statement will be placed in PARM-DATA; PARM-LEN will contain the length of the field.

Backward Referencing

Backward referencing is a method of referring to previous DD statements in a job stream. Its purpose is to ensure that datasets that should have the same attributes do so. Backward referencing may be used on the following parameters:

DSN=	the system obtains the dsname from the specified dd statement.
VOL=REF=	the system obtains UNIT and VOL information from the specified dd statement. This is required if retrieving a temporary dataset twice in one step.
DCB=	the system obtains DCB fields as coded on the specified dd statement.

There are three forms of backward referencing, depending on where in the job stream the reference is pointing to:

```
*.ddname
```

refers back to a previous ddname in the same step.

```
*.stepname.ddname
```

refers back to a ddname in a previous step.

```
*.stepname.procstepname.ddname
```

refers to a ddname within a procedure step within a procedure that was called by a preceding job step.

An example:

```
//STEPA  EXEC PGM=BANANA
//INPUT  DD  DSN=DEVADTR.BANANA.INPUT,DISP=MOD
...
...
//INPUT2 DD  DSN=*.INPUT,DISP=SHR
```

Generation Data Groups

A **Generation Data Group** (or GDG) is a collection of related datasets, maintained in chronological order; each one is known as a generation dataset. In JCL job streams, such datasets are usually referred to by their **relative** name:

```
DSN=dsname(generation number)
```

where **generation number** will have the following values:

Most recent generation is generation	0
Previous generations are generation	-1..-2..-3..
Next generations to be carried are	+1..+2..+3..

The relative generation numbers are only altered **at the end of a job** which creates or deletes one or more generations. Therefore, the dataset which is generation 0 at the beginning of a job should be referred to as generation 0 throughout the job, even if one or more generations are created by various different steps.

So, to read a generation dataset

DSN=CS.BCAMF(0),DISP=OLD	is the most recent generation
DSN=CS.BCAMF(-1),DISP=OLD	is the previous generation

If the generation number is omitted, then all existing generations in the group will be concatenated and treated as one single dataset, provided they all have the same DCB attributes.

When writing a new generation, it is necessary to refer to a model GDG:

```
//DICK  DD DSN=CS.BCAMF(+1),UNIT=....,
//      DISP=(NEW,CATLG,DELETE),
//   DCB=(BB.HK.MODELGDG,RECFM=...,LRECL=...,BLKSIZE=...)
```

New GDGs should be allocated by executing the ROSCOE RPF GDG.

Checking JCL

There are several ways of checking that JCL will work correctly without running it. The first of these is to specify **TYPRUN=SCAN** on the JOB statement, as described on page 8. This will cause the job to be scanned for errors but not executed.

A sophisticated on-line checking mechanism is provided by the **JCLCHECK** product. This may be invoked either from ROSCOE or from TSO. From ROSCOE, JCLCHECK is invoked by typing **JCK**. You are then prompted for various parameters. A batch job is then submitted which checks the contents of the AWS for JCL errors; the results of the check may be browsed by attaching the job in the normal way.

It is possible to run JCLCHECK online from TSO. To use this option, the JCL to checked must exist in a dataset. A comprehensive online tutorial is provided with the product to enable the new user to exploit the powerful functions available. It is not recommended that development staff spend a lot of time becoming familiar with this tool, as the ROSCOE JCK RPF provides all of the checking functions that are usually required.

Good JCL style

JCL is notoriously difficult to read if coded badly. The following tips should act as pointers in writing good JCL:

- always thoroughly comment your JCL
- give steps and symbolic parameters meaningful names
- don't use backward referencing when it can be avoided
- turn complicated, often used JCL into a catalogued procedure
- make sure that JCL statements are aligned
- test for condition codes on the EXEC rather than the JOB statement
- test complicated JCL using JCLCHECK.

EXERCISE 5

Code the JCL necessary to perform the following tasks:

Include a valid jobcard, including your own logonid. The job should run down a class appropriate for normal development usage. The msgclass should direct that the output from the job is held.

In the first job step, define a temporary dataset with fixed, 80 byte records. this dataset should be big enough to hold 100,000 bytes with potential to expand to 500,000 bytes. It should reside on the SYSDU pool.

In the second jobstep, invoke the COPYSEQ catproc to copy to the temporary dataset the contents of DEV@@@@.SAMPLE.INVENT.

Copy to the end of the temporary dataset the following data:

```
/////////////////////////////////////////////
//                                         //
//     EXTRA !!                            //
/*******************************************/
//   FREE BANANAS !!!                      //
//     AND TEAPOTS !!!!                    //
//                                         //
/////////////////////////////////////////////
```

Copy the resultant dataset to the spool. The output should go to remote printer 672 and should use the APL character set.

Each step should ONLY execute if each of the preceding steps finished with a zero condition code. Comment each step appropriately to describe what it does.