2 System elements.

2.1 Job control language.

The environment of 360/370 SIMULA is the 360/370 Operatin System. Actions to be performed under this system are specified by a job control language (JCL), the statements of which are called job control statements. The job control language is considered to be more complicated and more difficult to use than other similar control languages, but it is also very flexible and useful once it is mastered.

The general format of a JCL statement is:

```
//name  operation   operands
```

The two slashes are in columns 1 and 2 of a line and identify it as a JCL statement.

name is an identifier of not more than 8 consecutive alphameric characters, the first of which is a letter. The name starts in position 3.

operation is a word separated from the name and from the operands by at least one blank. The words considered here are JOB, EXEC and DD, identifying a job statement, an execute statement and a data definition statement, respectively.

operand is a list of operands separated by commas, with no intervening blanks. The operands are either positional (the meaning of the operand is determined by its relative position in the list) or keyword (the meaning is determined by a keyword followed by an equal sign preceding the operand).

An operand list can contain only positional operands, only keyword operands, or both, but all positional operands must precede all keyword operands in the operand list.

The operand list can be continued on the next line if it is interrupted after a comma. The following line is then marked with two slashes in position 1 and 2, and the operand list is continued starting anywhere between the 4th and 16th position. No part of an operand list may extend past position 71.

The use of each particular JCL statement is described in the succeeding sections. For reasons of readability and simplicity several operands are omitted and the rules are sometimes given in a stricter form than is necessary when this does not decrease usability. The actual rules are given in (6).

2.1.1 Job statement.

The largest unit of work recognized by the system is a job, defined by a job statement and the following JCL statements up to the next job statement in the input stream.

The name field of the job statement is the job name, used to identify the printed output from the job. The operands of a job statement supply account inginformation and programmer name by positional operands. Keyword operands can be used to specify whether the JCL statements of the job will appear on the message listing, and to specify a minimum size of core in which the job can be run.

Example:

```
//A  JOB  0,SMITH,MSGLEVEL=1,REGION=90K
```

A	is the job name
JOB	identifies the this as a job statement
0	is the accouting information. In general you must replace the 0 by your account number and possibly some other information depending on the installation.
SMITH	is the programmer name. Special rules apply if the name contains blanks or other nonalphameric characters (see (6)).
MSGLEVEL=1	requests the system to let the JCL statements of the job appear on the message listing. If omitted, no JCL statement will appear.
REGION=90K	is a request for a problem program partition of not less than 90 K bytes. The parameter is an unsigned integer followed by a K indication multiplication by 1024. In a system with LCS support (release 17 or later) storage of both hierarchies can be requested (see (6)).

2.1.2 Execute statement.

A job consist of one or more job steps. A job step is defined by an execute statement and the data definition statements (DD-statements) following it. The job steps of a job will be executed sequentially in the order in which they appear within the job. The execute statement defines a program which will operate on data defined by DD-statements of the job step. The name field of the EXEC statement is the job step name. The operands of the execute statement define the program to be executed and, optionally, a parameter string to the program and a test for bypassing the step.

Example:

```
//S1  EXEC  PGM=progname,PARM='parm string',COND=conditions
```

S1 — This is the step name (name of the execute statement) of the job step to be executed. This is used for referencing the step for instance in JCL messages.

PGM=progname — The PGM operand defines the program to be executed. Progname os the name of a load module (2.2.2). in the system link library, the job library or the step library.

The system link library (SYSL.LINKLIB) is always accessible. A job library can be defined by a DD-statement with the ddname JOBLIB, immediately following the job statement. A step library is defined by a DD-statement with the ddname STEPLIB, placed among the DD-statements of the step.

PARM='parm string'

A parameter in the form of a character string is passed to the program by means of the PARM operand. The interpretation of the parameter string is dependent on the program. The parameters allowed for the SIMULA compiler and the SIMULA object program are discussed in sections 2.2.1 and 2.2.3, respectively.

COND=conditions — conditions is a condition or a list of conditions separated by commas and enclosed in parentheses. If any one of the conditions is satisfied, the step will not be executed. A condition has the form (unsigned integer, relop, stepname)

relop is any of EQ, GE, GT, LE, LT or NE,
meaning=, >=, >, <=, < or ,
respectively.
stepname is the name of a preceding job step in the job.

When a program terminatesnormally, a return code is passed to the operating system. It the stepname of the conditions is replaced by the return code of that step you get an arithmetic relation, and the condition is satisfied if this relation is true. The return codes passed by application programs are discussed in subsections of 2.2.

A condition can also be any of the words EVEN and ONLY.

EVEN is never satisfied, which implies that the step will be executed even if an earlier step terminated abnormally.

ONLY is satisfied only if no earlier step has terminated abnormally.

If neither EVEN or ONLY occurs among the conditions, the job step will be bypassed if an earlier job step terminated abnormally.

Special rules apply if EVEN of ONLY is mixed with relational conditions (see (6)).

2.1.3 Data definition statement.

A data definition statement (DD-statement) defines either data on which the program of a job step will operate, or a program library from which the program is to be loaded. The name field of the DD-statement is the ddname, which connects the data to a logical file of the program. A detailed description on the use of DD-statement is found in section 5.

2.1.4 Catalogued procedures.

In order to reduce the number of control statements for frequently used job step combinations, the system provides a cataloguing facility for JCL statements. A set of catalogued JCL statements, constituting one or more job steps (procedure steps) is called a catalogued procedure. The operands of the JCL statements can be modified when the procedure is used by means of symbolic parameters. The operands that can be changed in this way are defined when the procedure is catalogued. Operands not given symbolic parameters can be overridden when the procedure is used. The use of the catalogued procedures supplied as part of the 360/370 SIMULA is described in section 2.4.

Invoking a catalogued procedure.

A catalogued procedure is invoked with an EXEC statement in which the keyword PGM operand is replaced by a positional operand which is the name of the procedure. The keyword operands of this statement are the symbolic parameters of the procedure.

Modifying execute statements of a procedure.

Operands of the execute statements of the procedure can be redefined in the EXEC statement invoking the procedure by means of operands of the form

```
keyword.procstepname=newoperand
```

where keyword is the keyword of the operand which is to be added or modified, procstepname is the name of the execute statement in the procedure which is to be modified, and newoperand is the new operand value.

If operands of more than one procedure step are redefined, the operands must occur in the same order as the procedure steps to which they apply.

Modifying and adding DD-statements in a procedure step.

Operands of a DD-statement of a procedure step can be modified by means of an overriding DD-statement:

```
//procstepname.ddname DD newoperands
```

DD-statements can be added to a procedure step with an ordinary DD-statement where the ddname is preceded by the procedure step name and a dot.

Overriding and added DD-statements must be sorted so that DD-statements belonging to a later step follow those belonging to an earlier step, and within each step overriding DD-statements must occur in the same sequence as the overriding DD-statements in the procedure.

Overriding DD-statements of a procedure step must precede added DD-statements of that step.

Some examples on modifying and added DD-statements are found in 2.4.

2.2 SIMULA System Elements.

The software elements needed to use 360/370 SIMULA are the SIMULA compiler, named SIMULA, the IBM-supplied Linkage Editor, named IEWL, and the run-time system load module library, named SIMLIB.

Furthermore, the JCL procedures of Appendix J should be catalogued, i.e. put in the catalogued procedure library SYS1.PROCLIB.

2.2.1 Compiler.

The SIMULA compiler will read a SIMULA source program (Appendix A), and it will produce one or more of the following items:

i) a source program listing,
ii) a cross-reference listing,
iii) an object module,
iv) a copy of the object module and
v) a diagnostic message listing.

The parameter string passed to the compiler from the EXEC statement determines which items are produced (2.2.1.1).

The object module is a machine-language, non-executable version of the source program (2.2.2).

The source program and cross-reference listings are useful for debugging and documentation.

When an error or a possible error in the source program is detected, an error or a warning message (Appendix B) is written. Production of the object module is suppressed if an error with severity code greater than 3 has been detected by the compiler.

The files used by the compiler are given in Table 2.1.

The return code issued by the compiler is the highest diagnostic severity code encountered (Appendix B), rounded upwards to a multiple of 4.

ddname	mode	use
SYSIN	input	contains the source program
SYSGO	output	will contain the object module produced by the compiler.
SYSLIB	input, partitioned	contains external class definitions used in the source program.
SYSPRINT	output	will contain the source listing, the cross-reference table and diagnostic messages.
SYSPUNCH	output	will contain the object deck (identical to the object module).
SYSUT1 SYSUT2 SYSUT3 SYSUT4	input, ouput	temporary data sets used to hold compiler created tables and partially translated code if these do not fit into the available memory.

Table 2.1: Compiler files

2.2.1.1 Compiler parameters.

Compiler control is achieved by means of the parameters coded in the PARM field of the compiler EXEC statement. The compiler parameters are a number of words, separated by commas and with no intervening blanks.

The parameters are separated into two groups, those that are only given by a keyword (positional keyword), and those that are given by a keyword and a value (value keyword).

Positional keywords are used to invoke various processing options of the compiler, and any of these can be preceded by NO which has a suppressing effect.

Word:	Meaning:
LIST	an assembly-like version of the compiled program is listed on SYSPRINT.
LOAD	an object module is produced on SYSGO.
DECK	a copy of the object module is written to SYSPUNCH.
WARN	warning messages will appear on the diagnostic message listing.
SUBCHK	instructions to check array indexing are compiled.
EXTERN	the source program on SYSIN is an external procedure definition.
XREF	a cross-reference listing is produced on SYSPRINT.
SOURCE	the source program is listed on SYSPRINT.
RESWD	reserved words will be underlined on the source listing. This is accompanied by by double printing and on most systems two lines will be charged for each line which contains a reserved word.
TERM	output abbreviated error messages on SYSTERM.
LONGREAL	All arithmetic computations involving real quantities should be performed in long real precision.

The following parameters are specified with a value following the parameters and an equalsign.

LINECNT=n	n is the number of lines per page written on SYSPRINT.
MAXERROR=n	n is the maximum number of diagnostic messages (errors and warnings) to be printed. Further diagnostics will be counted, however.
MAXLINES=n	n is the maximum number of lines to be listed in the source listing.
MAXPAGES=n	n is the maximum number of pages in the source listing. If the value is exceeded, the processing is terminated. n=0 is interpreted as "no limit".
INDENT=n	n is the number of positions that each block level is to be indented (i.e. shifted right) in the source listing to show the block structure of the program.
TIME=n	specifies that the compilation should be stopped after n/1oo cpu seconds. n=0 is interpreted as "no limit".
SYMBDUMP=n	specifies that provision of compiler generated data structures for runtime debugging is to be made, see section 6.3.

SIZE=(s1h1,s2h2,s3h3,s4h4,s5,s6)

The SIZE parameter controls the internal use of core by the compiler. The default size will be appropriate for most purposes, otherwise see section 2.2.1.2.

RESWD=n — specifies special marking of reserved words in the source listing. The value n has the following meaningful values:

0	no marking (identical to NORESWD)
1	underline reserved words (identical to RESWD)
2	reserved words in triple printing
3	reserved words in lower case, identifiers in uppercase and standard names in lower case with capital letter.
4	reserved words in upper case, identifiers in lower case and standard names in lower case with capital letter.

The underlining of the reserved words in the compilation listing is only possible if the applied printer allows printing of more than one image at one physical line and the printer chain contains the underscore (_) character.

EXTERN=v — Specifies the kind of compilation:
O: main program,
C: external class,
P: external procedure.
'EXTERN' is equivalent to EXTERN=P, and
'NOEXTERN' is equivalent to EXTERN=O.

The default parameters and values are:

'NOLIST,NODECK,LOAD,WARN,SUBCHK,NOEXTERN,NOXREF,SOURCE,
NORESWD,LINECNT=60,MAXERROR=50,INDENT=0'.

SIZE parameter

The SIZE parameter is composed of 6 subparameters, written in the general format:

SIZE=(s1h1,s2h2,s3h3,s4h4,s5,s6)

Each si (i=1,2,...,6), stands for a value of the form ddd,,d (a decimal integer) or ddd...dK, where K means multiplication by 1024, or it may be omitted, leaving the default value unchanged.

Each hi (i=1,2,3,4) is either "H1" or omitted.

"H1" means that the corresponding area or areas should be placed in LCS (large core storage, hierarchy 1). Otherwise the area(s) are placed in HSS (high speed storage, hierarchy 0).

Trailing commas need not be written.

The subparameters have the following significance (refer to "DEFAULT"):

s1 - DSYMBUFL — length of buffers for SYSUT1 and SYSUT2 (intermediate language symbols). Preferably a multiple of 16.

s2 - DIDBUFLE — length of buffers for SYSUT3 (idenfifier list). Should be a multiple of 16.

s3 - SYSFREE — work area size for system use. Must accommodate one or two buffers for each of SYSGO and SYSPUNCH (if used), in addition to some space for transient system modules.

s4 - COREMIN — this is the minimum work storage needed by the compiler. If DIDBUFLE and COREMIN have the same storage hierarchy, and COREMIN <= LIMBLKSZ, then the actual min. work area size is (2*DIDBUFLE+COREMIN) except in passes 1,2 and 9, because SYSUT3 buffers are released when not in use in that case.

s5 - COREMAX — maximal amount of core requested in any hierarchy. Must be larger than the sum of component areas in the respective hierarchies. The effective components are: 1*SYSFREE, 1*COREMIN, 2*DIDBUFLE, 4*DSYMBUFL, 8*MAXERROR.

s6 - LIMBLKSZ — largest block size allowed to be written on SYSUT1-3. A larger value of DIDBUFLE or DSYMBUFL forces the corresponding file to be kept entirely in-core.

Default values of S1,...,S6 are installation dependent, and, for the sake of efficiency, it is important that any change in buffer size etc. should be reflected in the corresponding DD statement.

2.2.2 Linkage Editor.

The Linkage Editor is an IBM-supplied program which reads the object module and produces an executable object program.

An executable program is always a member of a program library on direct-access secondary storage. In IBM terminology it is called a load module, since it has a different organization from an object module and it can be loaded into core by the control program.

The object module (primary input) is read from a file with ddname SYSLIN. Additional input is taken from a loaded module library identified by the ddname SYSLIB to resolve external references (automatic library call). The load module produced is put in the library identitied by ddname SYSLMOD, under a name given in the DD-statement.

Information and diagnostic messages, as well as an optional map and cross- reference listing, useful for debugging on the machine-code level, are printed on a data set with ddname SYSPRINT.

The Linkage Editor will optionally perform a large number of functions requested by control statements in the primary input. One of these functions, overlay editing, is described in Appendix E, while the remaining functions are described in (10).

2.2.3 Object program.

The object program output from the Linkage Editor is an executable program corresponding to the source program. When this program is executed by means of an execute statement, the machine equivalents of the SIMULA statements will be executed. The PARM operand of the execute statement can be used to control the storage setup and the amount of debugging information produced (2.2.3.1).

The return code to the operating system can be used in COND operands of following job steps (2.2.3.2).

If a run-time error occurs during the execution of the object program it is terminated at once and a diagnostic message is printed together with programmer controlled debugging information on a file identified by the ddname SYSOUT (Section 6).

The run-time system is assigned the three letter prefix ZYQ, occurring in all diagnostic messages and control section names.

2.2.3.1 Object program parameters.

The parameters to the object program are given by the PARM operand of the execute statement in the form of a character string enclosed in quotes. The character string consists of keyword operands separated by commas with no intervening blanks. If a permitted keyword is omitted, the default value of that operand is used. The permitted keywords and operand formats are shown below.

DUMP=digit — The DUMP parameter controls the post-mortem dump (see section 6.2). The digit is 0,1,2,3,4,5,6 or 7. Default value is 1, giving a diagnostic message and a register dump.

HIARCHY=0 or 1 — In a system with LCS support this operand determines the hierarchy of the SIMULA working storage. The default value is 0 (fast core storage).

LINECNT=unsigned integer

The unsigned integer is the initial value of the variable LINES PER PAGE of a printfile object.

60 is the default value.

MAXPAGES=unsigned integer

The amount of user program output on SYSOUT is controlled together with all other printfiles. If the value is exceeded, the processing is terminated. A value of zero means "no limit".

SIZE=(q1,q2,q3) This operand controls the partitioning of available storage into SIMULA working storage and system free storage. The size of the SIMULA working storage is fixed during execution, and is used for declared quantities, texts, arrays and control blocks for program sequencing and the sequencing set, see section 2.2.4 for further details.

q1, q2 and q3 are unsigned integers, optionally followed by the letter K indicating multiplication by 1024.

Only those of the sizes which are required may be specified, however, only trailing commas may be omitted.

All sizes are measured in bytes.

q1 is the maximum expected sum of the sizes of all blocks, arrays, texts and temporary results in the SIMULA program. If this size is exceeded, the object program is terminated.

If q1 is omitted it is set equal to q3.

q2 is the minimum size of the system free storage. It should be large enough to accommodate access routines, I/O buffers, and all GETMAIN, LINK and LOAD requests from non-SIMULA external procedures.

q2 is set to 10K if omitted.

q3 is the desired size of the SIMULA working pool. If q3 is omitted all storage except that defined by q2 is allocated.

If q2 and q3, are both specified, q2 takes precedence if the sum of q2 and q3 is greater than the available storage. If the sum is less, q3 takes precedence.

TRACE=unsigned integer

This operand determines the size of the tracing buffer for control and dataflow tracing (See section 6.3).

TEST
NOTEST print header lines prior to the first line of the program output indicating the compilation date, release identification and the options in force and traling lines indicating the total exectuion time, return code and the time spent in garbage collection (if any).

TIME=unsigned integer

set cpu-time limit for execution in hundredths of seconds - the default value = no limit. Appropriate diagnostics and optional dump are obtained in the case of limit overflow rather than abnormal termination, thus facilitating the location of unintended infinite loops.

SYMBDUMP=digit specifies the level of information to be given in a post-mortem symbolic dump (see section 6.4). The digit is 0, 1, 2, 3, 4, 5 or 6. Default value is 3, giving the heading of blocks on the operating chain and a symbolic dump of the local quantities in the involved blocks.

The default values indicated are common standard. They can, however, be altered to any other default values at system installation using the SIMULA system macro SIMRDF which is a standard part of every system delivery. The same macro also allows alteration of the standard ddnames of the RTS files.

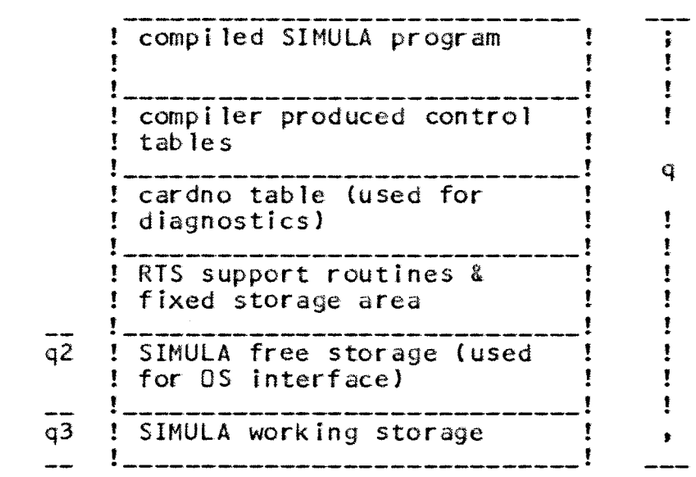
2.2.3.2 Return code.

The object program passes a return code to the operating system indicating the way the program was completed.

Code:	Meaning:
0	Successful completion.
4	One or more edit overflows occurred.
8	The program was terminated because of a run-time error.
12	One or more edit overflows occurred and the program was terminated because of a run-time error.

2.2.4 Storage management under RTS.

The following is a simplified scheme of the storage allocation during a SIMULA program execution:



This figure illustrates use of storage in the case where a SIMULA program is executed in a separate JCL step (it has to be modified accordingly for situations where the execution is invoked using the OS loader or NCC's utility SIMCNT). In this case the value of q is equal to the size of the region allocated to this task by the operating system.

The size of the SIMULA free storage is 10K by default. If more core than q2 bytes is needed for OS interface routines (e.g. when performing I/O using extraordinarily large buffers), the execution terminates abnormally (usually with the System Completion Code 804).

The SIMULA working storage is by default the largest contiguous area of core left within the region after all other core requests are fulfilled, and its size is printed in the RTS header.

The internal organisation of the SIMULA working storage is as follows:

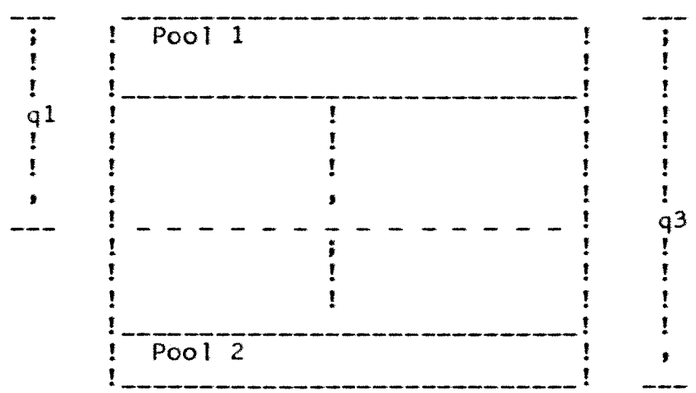


fig. 2.

Pool 1 is used for block instances, array and text objects and temporary results; pool 2 is used for control blocks used by the RTS working storage management system. The governing idea of this system is that pool 1 and pool 2 are extended in the course of program execution in the respective directions indicated by arrows in fig. 2 until they use up all the working storage area. The garbage collection is then automatically performed which deletes all unreferenceable data structures so that further extensions of the pools are possible. If garbage collection does not result in a sufficient gain of core, the RT error ZYQ0017 (STORAGE EXHAUSTED) is forced.

Further, it is the user's option to set up a limit for pool 1, namely the value of q1 in fig. 2. If this is done and pool 1 reaches this limit in the course of the program execution without garbage collection being able to compress pool 1 beyond this limit, the RT error ZYQ0018 (DATA LIMIT) results.

The value of q1 is zero by default, which the RTS interprets as "no pool 1 limit" and allows pool 1 to expand until it reaches pool 2.

A user may exercise control over the RTS storage management via the parameter SIZE, where the respective values of q1, q2 and q3 may be specified in the form of subparameters, see section 2.2.3.1.

A user should bear in mind that:

- the range of the values of the respective subparameters is dependent on the program size, the size of the requested RTS support and the region size.

- if both q2 and q3 are specified, q2 takes precedence if the sum of q2+q3 is greater than the available storage, otherwise, q3 takes precedence.

- the main storage request issued by the RTS via GETMAIN on behalf of the SIZE subparameter q3, is conditional and no extra action is taken if the requested amount of the main storage is not available.M

2.3 External procedures and classes.

External procedures and classes make it possible to compile procedure and class declarations separately.

An external procedure can be saved as an object module (2,4. ex. 7) or as a load module (2,4 ex. 8).

An external class is saved in an image library.

External procedures can be replaced in a program without recompilation of the main program, but external classes cannot.

External procedures can be of three different types: SIMULA, FORTRAN or ASSEMBLY. How to write an external procedure of type FORTRAN or ASSEMBLY is described in Appendix G.

2.4 Use of catalogued procedures.

The following sample jobs illustrate the use of the catalogued procedures listed in Appencix I.

The catalogued procedures are:

SIMC — compile source program, create object module. The procedure consists of one procedure step named SIM, which is an execution of the SIMULA compiler. The procedure has no symbolic parameters.

SIMCL — Compile source and linkedit to executable PROGRAM. Procedure step names are SIM and LKED. Symbolic parameters:

PROG=progname
progname will be the name of the created program.

LIB='libname'
libname is the name of the catalogued library in which the object program is placed.

EXLIB='libname1'
libname1 is the name of the catalogued library from which external procedures are taken.

LDISP=OLD or MOD
OLD: the object program is to replace an existing, identically named program in the library.
MOD: The object program is to be added to the library.

SIMCLG — Compile, linkedit and execute object program. Procedure step names are SIM, LKED and GO. Symbolic parameter:

EXLIB='libname'
Name of library from which external procedures are to be loaded.

SIMG Execute a SIMULA object program from a load module library. The only procedure step of SIMG has the name GO. Symbolic parameters:

PROG=progname
Name of program to be executed.

LIB='libname'
libname is the name of the load module library.

SIMCG Compile cource and execute the LOADER program (which combines the steps LKED and GO in SIMCLG). Step names are SIM and GO. Symbolic parameter same as in SIMCLG.

SIM A one-step JCL-procedure which makes use of a special program called SIMCNT. This program is a standard part of every SIMULA system, starting with release 3.0. For details on this procedure, see section 2.4.1.

Note: The mandatory job statement has been omitted in all the following examples.

```
!                                                  !
!  //A  EXEC   SIMC,PARM.SIM='NOLOAD,XREF'     1   !
!  //SIM.SYSIN   DD  *                         2   !
!     <source program deck>                        !
!  /*                                        3   !
!__________________________________________________!
```

Ex. 1: Compile program to obtain program listing, cross-reference listing and diagnostic messages.

Explanation (digits refer to line numbers of the example).

1: This line invokes the catalogued procedure SIMC. The PARM operand of its only step, SIM, is replaced by 'NOLOAD,XREF', which will suppress object module generation (NOLOAD) and cause the cross-reference listing to be printed (XREF).

2: This dd-statement defines the source program data set. The asterisk indicates that the data set follows this line in the job stream.

3: The delimiter statement signals the end of the source program.

```
!-------------------------------------------------!
!                                                 !
! //A  EXEC   SIMC,PARM.SIM='DECK,NOLOAD'      1  !
! //SIM.SYSPUNCH DD DSN=MYSAVE,UNIT=3330-1,---  2  !
! //SIM.SYSIN DD   *                           3  !
!      <source program                            !
! /*                                          4  !
!-------------------------------------------------!
```

Ex. 2: Compile and save program.

When the program has been found to be free from errors, the object module can be saved. This will mean that when the program is to be executed it is not necessary to recompile it each time.

1: Since we want to produce an object module but not load it, the parameter to the compiler is 'DECK,NOLOAD'.

2: This DD-statement has been added to the procedure step. SYSPUNCH is the ddname on which the module is stored.

3 and 4:
Same as 2 and 3 in the preceding example.

```
!-------------------------------------------------!
!                                                 !
! //A  EXEC  SIMCLG          1                    !
! //SIM.SYSIN DD *           2                    !
!   <source program>                              !
! /*                         3                    !
! //GO.SYSIN  DD *           4                    !
!   <test data>              5                    !
! /*                         6                    !
!-------------------------------------------------!
```

Ex. 3: Compile, linkedit and execute.

This is the typical test run in a debugging cycle. The object program is not saved.

1: This statement invokes the SIMCLG catalogued procedure.

2 and 3:
Same as in the previous examples.

5: Test data to be read by sysin.

6: Signals end og file for sysin.

When you want to use the LOADER program replace SIMCLG by SIMCG in line 1.

```
//A  EXEC  PGM=IEFBR14                          1
//LIB  DD DSN=A.LLIB,DISP=(NEW,CATLG),          C2
//  UNIT=2314,VOL=SER=XXXXXX,                   C3
//  SPACE=(TRK,(10,5,20))                        4
//B EXEC  SIMCL,PROG=PROGA,LIB='A.LLIB'         5
//SIM.SYSIN  DD *                               6
    <source program>                            7
/*                                              8
```

Ex. 4: Create a program library in which an object program is saved.

The library is created in a dummy step preceding the compilation.

1: This statement requests an execution of the dummy program IEFBR14.

2: The first line of the DD-statement requests the system to catalogue a new data set with data set name A.LLIB. A must be the name of an index in the catalogue.

3: The first continuation line defines the disk pack on which the data set will be put: 2314 disk pack with serial number XXXXXX.

4: The second continuation line defines the space allocated to the data set (section 5.3.2.2). The number 20 indicates that the data set will be a library with 20 directory blocks, which will permit approximately 5 * 20 = 100 different programs in the library.

5: This statement invokes the SIMCL catalogued procedure. The symbolic parameters request the linkage editor to add the object program to the library A.LLIB under the name PROGA.

```
//A  EXEC    SIMCL,PROG=PROGA,LIB='A.LLIB',LDISP=OLD    1
//SIM.SYSIN    DD *                                     2
   <source program>
/*                                                      3
```

Ex. 5: Compile and replace a program in an existing program library.

The first statement of this job requests the linkage editor to replace the program PROGA in A.LLIB with the object program resulting from this compilation.

```
//A  EXEC   SIMG,PROG=PROGA,LIB='A.LLIB'            1
//GO.SYSIN   DD *                                   2
    <execution data>
/*
```

Ex. 6: Execute an object program from an old program library.

When an object program has previously been saved, the above example shows how it will be executed.

```
//A  EXEC  SIMC,PARM.SIM=EXTERN                     1
//SIM.SYSIN  DD *                                   2
   <external procedure source>                      3
/*                                                  4
//B  EXEC  SIMCLG                                   5
//SIM.SYSIN  DD *                                   6
   <main program source>                            7
/*                                                  8
//GO.SYSIN   DD *                                   9
   <test data>                                     10
/*                                                 11
```

Ex. 7: Compile and run external procedure and main program.

Statement 1 - 4 can be repeated if there is more than one external procedure.

```
//A  EXEC SIMCL,PROG=PROC1,LIB='A.LLIB',                1
//  PARM.SIM=EXTERN,PARM.LKED=NCAL                      2
//SIM.SYSIN  DD *                                       3
  <external procedure source>                           4
/*                                                      5
```

Ex. 8: Save an external procedure in a program library.

1: Add the operand LDISP=OLD if an old procedure is to be replaced.

2: The linkage-editor parameter NCAL prevents addition of run-time system elements. This will save space in the library. The operands must occur in this order since the SIM procstep precedes LKED in the catalogued procedure.

```
//A  EXEC SIMCLG,EXLIB='A.LLIB'      1
//SIM.SYSIN  DD *                    2
  <source program>
/*
//GO.SYSIN   DD *                    3
  <test data>
/*
```

Ex. 9: Compile and execute program using external procedures in a load module library.

The external procedures used must have been put in A.LLIB using the method of Example 8.

The name specified in the PROG operand of Example 8 must coincide with the identifier of the procedure and with the <external identifier> of the external procedure declaration in the main program.

2.4.1 Using the SIMULA system with the JCL procedure SIM

SIM is a one-step JCL procedure which makes use of a special program called SIMCNT. The basic action of SIMCNT is to retrieve input programs and process them, one after another.

There are four modes in which SIMCNT can operate:

```
source mode = input program is in source code
object mode = input program is an object module
load mode   = input program is a load module
update mode = input program is a source code which is to be updated
              prior to further processing.
```

The operating mode may be freely changed for any program.

Program processing in the source mode consists of compilation (using the SIMULA compiler), loading (using the OS LOADER) and execution. Once compiled and loaded, a program can be executed an arbitrary number of times. The above mentioned pattern of program processing is cut short if either compiler or loader actions were not successfull or if only compilation was requested; processing continues with the next program in such a case. In object mode, program processing consists solely of loading and subsequent execution (possibly multiple executions). In load mode, the program is loaded from secondary storage into core (using the LOAD macro) and then executed the specified number of times. In update mode, the source program is updated (using the IEBUPDTE utility) and the new master, which is a temporary, sequentially organised data set, is then treated as an ordinary input program in source mode unless updating was not successfull.

The function of SIMCNT and mode switching are controlled by a trivial command language, sentences of which are passed to SIMCNT via the SIM symbolic parameter P, which is positioned at the beginning of the EXEC PARM field. A sentence of the SIMCNT language is a string of arbitrary length consisting of the letters C, O, L, U, E and unsigned integers. An empty string is also legal and it is interpreted as digit 1. In addition, program names (terminated by commas, if necessary) may also be part of the control string.

The meanings of the respective symbols are as follows:

C ... switch to source mode:
retrieve and compile a source code program.

O ... switch to object mode:
retrieve, load and execute an object program.

L ... switch to load mode:
retrieve the program whose name follows the L (and is delimited by a comma if not terminating the string), and execute it.

U ... switch to update mode:
retrieve and update a source program, then compile the updated version.

E ... if in source or update mode:
(load if necessary and) execute the program which was compiled last, otherwise execute the last loaded program.

unsigned integer

... if followed by any of the above control letters, indicates repetition of the associated action (e.g. 2C means CC, 3E means EEE, etc.); if terminating the control string: perform CE the requested number of times (i.e. compile/load/execute the specified number of programs, e.g. digit 3 at the end of the control string stands for CECECE).

The physical location of prograams to be processed by SIMCNT depends on their processing mode: the program to be processed in load mode must be a member of a load library specified as or concatenated with STEPLIB; the program to be processed in update mode must be a (member of a partitioned) data set specified as (or concatenated with) SYSLIB; any other program must be a sequential data set with ddname SYSINn, where n is empty for the very first program to be processed or i-1 for the i-th program (regardless of mode witching). In update mode, SYSINn is used as a ddname for the IEBUPDTE control statement data set. Execution data (if any) must be submitted on the following ddnames:

DATA for the first execution of the first program (i.e. on SYSIN)

DATAj for the (j+1)-th execution of the first program (i.e. on SYSIN)

DATAij for the J-th execution of the i-th program (i.e. on SYSINi)

Notes:

- Execution data set ddname conflicts must be resolved by appropriate sequencing of the programs (e.g. DATA11 may be data for the 12th execution of the first program as well as for the first execution of the second).

- Syntax checking of the control string is minimal and the effect of illegal strings is unpredictable.

- The global condition code returned by SIMCNT is:
(max update/compilation cc)*100 + (max loading/execution cc) or 1111 in the case of the command language syntax error.

- Control of the processors involved (compiler, loader, RTS) may be exercised in the usual way, i.e. by specifying requested options in the EXEC PARM field. But remember that this affects all processed programs equally. (The EXEC PARM field to be used with SIMCNT may either be empty or of the following format:

  PARM='<SIMCNT control>/<SIMULA comp. parm.>/<loader
  parms.>/<RTS parms.>/<user parms.>'

  Any angle bracketed part may be empty, but only trailing slashes may be dropped.)

The complete set of control statements for procedure SIM as suggested in the SIMJCL data set of the release tape are given in appendix I.

Note that the basic set of DD statements used by SIMCNT is identical to that of a standard SIMULA compiler. Providing that SYSPRINT and SYSGO DD statements are not altered, SIMCNT operates equally well with any compiler version regardless of its default ddnames.

Additional data sets may also be used.

Setting of the RTS parameter TIME is a recommended safety precaution which will inhibit waste of CPU time if an infinite loop occurs in some executed program (the time unit used is 1/100 second).

Examples of SIMCNT activation using SIM:

```
//A      EXEC SIM
//SYSIN  DD   *
  <SIMULA program>
//DATA   DD   *
  <execution data>
/*
```

Ex. 10: Compile, load and execute one SIMULA program

```
//B      EXEC SIM,P=2,CP=XREF
//SYSIN  DD   *
  <1st program>
//DATA   DD   *
  <data for 1st program>
//SYSIN1 DD   *
  <2nd program>
//DATA11 DD   *
  <data for 2nd program>
/*
```

Ex. 11: Compile two source programs, taking cross-reference listing and execute them.

```
//C      EXEC SIM,P=COE
//SYSIN  DD   *
  <1st program (source)>
//SYSIN1 DD   *
  <object module of the 2nd program>
//DATA11 DD   *
  <data for the 1st exec of 1nd prg>
//DATA12 DD   *
  <data for the 2nd exec of 2nd prg>
/*
```

Ex. 12: Compile one program, then load and execute twice another program which is supplied in object module form

```
//D      EXEC SIM,P='LPROGRAMX,2EC'
//DATA   DD   *
  <data for 1st exec of PROGRAMX>
//DATA1  DD   *
  <data for 2nd exec of PROGRAMX>
//DATA2  DD   *
  <data for 3rd exec of PROGRAMX>
//SYSIN1 DD   *
  <source program>
/*
```

Ex. 13: Execute three times PROGRAMX which has previously been compiled and linked-edited (into SIMLIB), then compile a source program

```
//E      EXEC SIM,P=U
//SYSLIB DD DSN=OLDTEST,DISP=SHR
//SYSIN  DD *
./  CHANGE NAME=TEST,NEW=PS
 -- update data statements --
./  ENDUP
/*
```

Ex. 14: Update and compile a source program which is held under member name TEST in pds OLDTEST.