Draft:Not for distribution
version: 0.90
processed: August 31, 1994
This is a working draft and contents are not fully debugged.
If any error is found, please inform it to the author.

FULL for TOPAZ analysis

INTERNAL USER ONLY

Akiya Miyamoto*
KEK, National Laboratory for High Energy Physics
1-1 Oho, Tsukuba, Ibaraki 305, Japan

ABSTRACT

As an unified TOPAZ data analysis system, FULL program package has been developed based on PEP4 FULL system. In this paper, input data for JCL version TOPAZ FULL are described, together with the FULL concept.

*e-mail address: miyamoto@kekux1.kek.jp

Contents

1 FULL System **3**

1.1 What is FULL 3

1.2 Example of *DESCRIBE* 4

1.3 Example of FLANLI 7

1.4 Input command interpreter 7

2 Input commands for `STEERING`. **9**

2.1 INPUT 9

2.2 OUTFILE 10

2.3 DELETE 11

2.4 OUTPUT 11

2.5 REMOVE 12

2.6 PARAMETER 12

2.7 EVENT 13

2.8 DATA 14

2.9 STEERING 14

3 DESCRIBES for TOPAZ data analysis. **14**

3.1 Names of DESCRIBE 14

3.2 Parameters 14

3.3 Input data for simulator module 17

3.3.1 SEED 17

3.3.2 FLAG 18

3.3.3 PARAMETER 18

3.3.4 EVENT 19

3.3.5 BCL 19

3.3.6 TPC 20

1 FULL System

1.1 What is FULL

FULL system was originally developed at PEP4 TPC group. R. Itoh and A. Shirahashi implemented it on FACOM MSP for use by TOPAZ group (Interactive version). Since this version puts emphasis on the interactive use, the author slightly slightly modified it and prepared the JCL version FULL program. Official TOPAZ DST production is based on the JCL version FULL program. Both Interactive and JCL versions are now widely used in the TOPAZ group. FULL is also implemented on SUN and HP Unix system. This paper mainly discuss the JCL version FULL program.

High energy data analysis usually consists of several independent analysis step, such as track fitting of tracking detector and energy cluster reconstruction of calorimetric detector, however they require initialization such as calibration constants and histograming commonly at the beginning of the job or run. In order to standardize the data analysis, keeping the independency of the analysis step, FULL coordinate the analysis by *module*. Subroutines are assigned to each module corresponding to the analysis step.

By using FULL, it is easy to include or remove data analysis module. It also uses has TBS system for data communication to keep independency between modules.

Typical flow of the program is shown in Fig. 1.

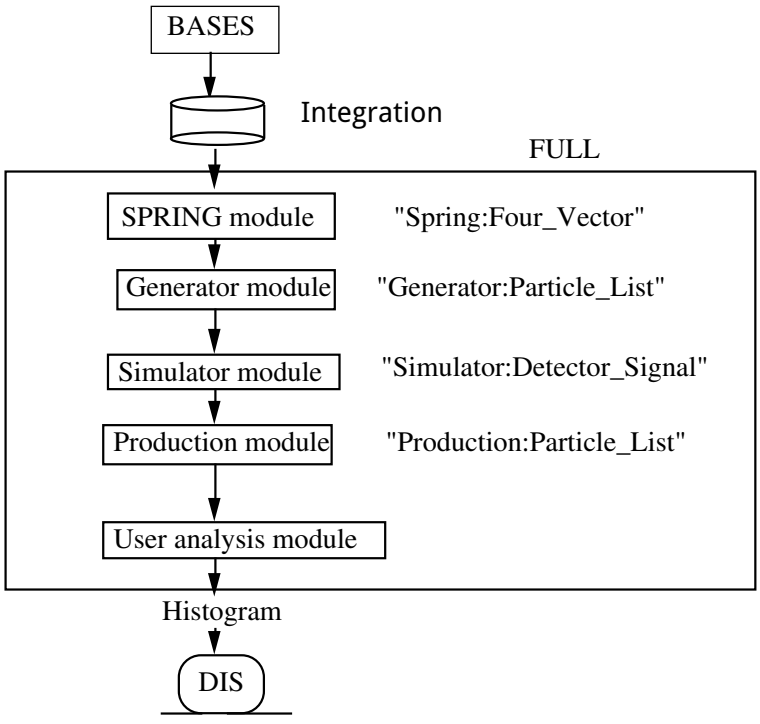

Figure 1: Typical example of FULL system

At the initialization, the subroutine `FLANLI` is called to define analysis module. User

should prepare *DESCRIBES* to define their analysis. In the *DESCRIBE* you have to set following variables.

namcom(i)	;	module name
entdia(i)	;	subroutine address to read parameter.
entini(i)	;	for job initialization.
entbru(i)	;	for run initialization.
entevt(i)	;	for event analysis.
enteru(i)	;	for run termination.
entter(i)	;	for job termination.
listsp(i)	;	module execute sequence.

As a naming convention, the first two characters of the DESCRIBE subroutine should be `DS`. These routines are called in the subroutine `FLANLI`. And analysis program is executed according to the calling order of the DESCRIBE subroutine in the subroutine `FLANLI`.

After the DESCRIBEs are defined in `FLANLI`, the program reads data from the FORTRAN unit number 5. If the obtained character string is same as the module name, the main program calls the subroutine defined by `ENTDIA`. At the point, the control of the program is passed to the user program. The user program should read data from the FORTRAN unit number 5 and does whatever they want. It should return to the caller when it reads the characters "EXIT". When the main program read the character "GO" while expecting the module name, it terminate the parameter reading and start the execution of the analysis.

When the analysis is started, it calls the job initialization routine defined by `ENTINI` as an job initialization. Then it enters the event loop. In each event, the program reads data from a file by calling TBREAD or initialize TBS buffer for event generation by itself. In the event loop, the begin run module, event module and end run module are called according to the type of the event obtained. Finally, the end job modules are called at the end of the job. The flow of the job is shown in Fig. 2.

1.2 Example of *DESCRIBE*

An example of *DESCRIBE* is shown below. In the example, `PRUCOM, PRUJIN, PRUJTM, PRURIN, PRURTM,` and `PRUSER` is the subroutine name of parameter reading, job initialization, job termination, run initialization, run termination and event analysis respectively. `PRGNAM` is the name of the *DESCRIBE*. `NENTR` is the sequence number of the *DESCRIBE* module and should be incremented in the *DESCRIBE* subroutine. `LISTSP(0)` keeps the total number of the full module and `LISTSP(i)` is used to define the calling order of FULL module. The default values of the execution level `LEVEL` and the debug level `DEBUG` stored in `NJBLVL(20,1)` and `NJBDBG(20,1)` are set to the array `EXELVL` and `DBGLVL`.

CALL FLANLI ; define user program by ***DESCRIBEs***

```
call DSSPRG
call DSCGEN
call DSCUSR
.........
```

What is defined in DESCRIBE ?

```
namcom(i)  ; module name
entdia(i)  ; subrtn address to read parameter.
entini(i)  ;              for job initialization.
entbru(i)  ;              for run initialization.
entevt(i)  ;              for event analysis.
enteru(i)  ;              for run termination.
entter(i)  ;              for job termination.
listsp(i)  ; module execute sequence.
```

data analysis flow.

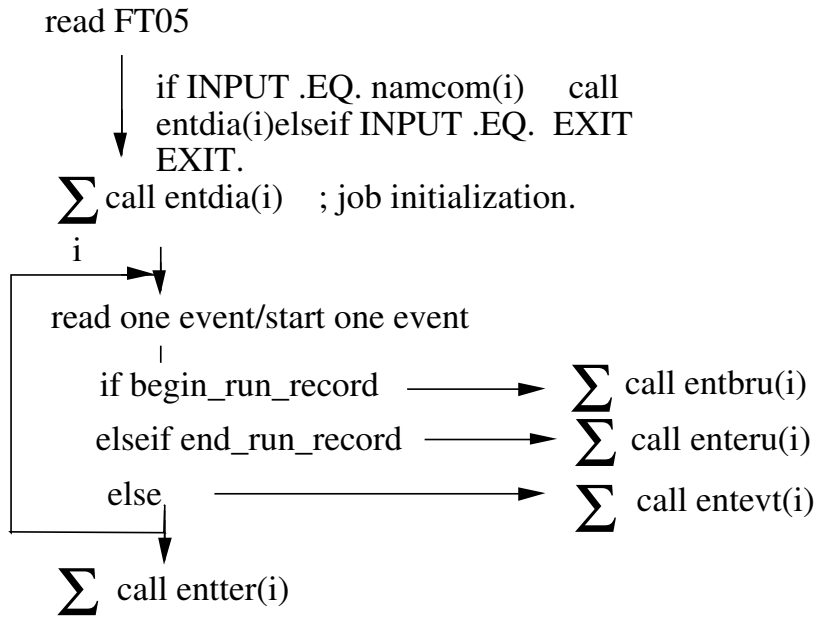

Figure 2: Job flow of FULL

```
C*********************************************************************
C*
C* -----------------
C* Subroutine DSCUSR
C* -----------------
C*
C*(Function)
C*   Describe User routine.
C*
C*(Input & Output)
C*   None.
C*
C*(Relation)
C*   CALLed from FLANLI
C*
C*********************************************************************
C
      SUBROUTINE DSCUSR
C
=EXPAND 'T#PR.FULL.FORT/FLDSCR.inc'
=EXPAND 'T#PR.FULL.FORT/FLLEVL.inc'
=EXPAND 'T#PR.P#STER.FORT/PRJFLG.inc'
C
C FUNCTIONS
C
      EXTERNAL      PRUCOM, PRUJIN, PRUJTM, PRURIN, PRURTM, PRUSER
      CHARACTER*12  PRGNAM/'USER:      '/
C
C ======< Entry Point >=============================================
C
C ---------------------------------------------------------------------
C (1) Define program name.
C ---------------------------------------------------------------------
C
      NPROG = NPROG + 1
       NAMCOM(NPROG) = PRGNAM

      ENTSTA(NPROG) = 0
       CALL EXTSUB (ENTDIA(NPROG),PRUCOM)
      CALL EXTSUB (ENTINI(NPROG),PRUJIN)
       CALL EXTSUB (ENTTER(NPROG),PRUJTM)
       CALL EXTSUB (ENTBRU(NPROG),PRURIN)
       CALL EXTSUB (ENTERU(NPROG),PRURTM)
C
C---------------------------------------------------------------------
C 2. Event Analysis routine
C---------------------------------------------------------------------
C
      NENTR = NENTR + 1
       ENTNAM(NENTR) = PRGNAM
       CALL EXTSUB (ENTEVT(NENTR),PRUSER)
      INPBLK(NENTR) = 0
      OUTBLK(NENTR) = 0
C
C---------------------------------------------------------------------
C 3. Define default calling sequence.
C---------------------------------------------------------------------
C
     LISTSP(0)     = NENTR
      LISTSP(NENTR) = NENTR
C
C---------------------------------------------------------------------
C 4. Define Level and Debug operand.
C---------------------------------------------------------------------
C
       EXELVL(NENTR) = NJBLVL(20,1)
       DBGLVL(NENTR) = NJBDBG(20,1)
C
      RETURN
       END
```

Figure 3: Example of a *DESCRIBE* program.

1.3 Example of FLANLI

An example of the subroutine FLANLI is shown below. In the example below, only the *DESCRIBE* for a user analysis described in the previous subsection is called. The subroutine `PRPDSC` is called just to assign the name of `FLANLI`, version number and date of `FLANLI`.

```
CC*************************************************************CC
C*                                                             *C
C*=================                                            *C
C* Subroutine FLANLI                                           *C
C*=================                                            *C
C*(Purpose)                                                    *C
C*    Define analysis program for standard production job.     *C
CC*************************************************************CC
C
      SUBROUTINE FLANLI
C
=EXPAND 'T#PR.FULL.FORT/FLDSCR.inc'
C
C======< ENTRY POINT >=============================================
C
C ---------------------------------------------------------------------
C (1) Declare version number of Full module.
C ---------------------------------------------------------------------
C
       CALL PRPDSC( 'ANALYSIS', 101, 891109)
C
C ------------------- User analysis
      CALL DSCUSR

      RETURN
      END
```

Figure 4: Example of a `FLANLI` program.

1.4 Input command interpreter

This section describes the input parameters for so called "JCL-Full" program.

Various parameters for the job control can be supplied as an input data from FORTRAN logical unit number 5. The general format of input data is shown in Fig. 5. The data consists of the sequence of the input data for each module, terminated by the word `GO`. `GO` signals the start of the execution.

Input data for each module should started by the name of the module and terminated by `EXIT` as shown below (Fig. 6). Between them, input data should be prepared depending on the module. Input for each module can be omitted, if the module can be executed by the default parameter set. Note that the parameter reading routine set in `ENTDIA` by the

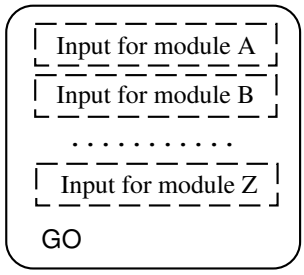

Figure 5: General structure of input commands.

D̂ESCRIBE is never called if the corresponding name does not appear in the input data from `FT05`.

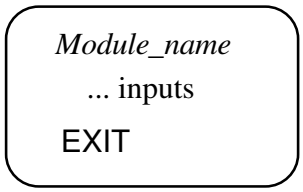

Figure 6: Input data format of each module.

Input data is first interpreted by CLI/COM routines which was originally interpreted by PEP4-TPC group. CLI/COM routines consideres the line started by `@!` as a commented line. The characters after `!` in the line are also comment. Non-commented lines are stored in the 4th element of the bank `FULL` in the begin_run record. The line started by a character `@` is a macro command and command strings are read from the file. The string following `@` up to next blank character is the file name where the command is obtained. The rest of the string other than comments are considered as arguments. The character "`% n`" (`n` is a positive integer) in the command file are replaced with the n-th argument specified as the command arguments. The example is shown in Fig. 7.

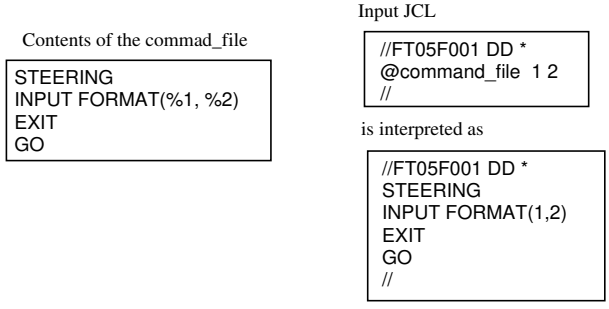

Figure 7: Example of a usage of macro command arguments.

When the *module_name* is `STEERING`, the input data is for FULL steering. Valid parameters for them are described in the following section. For these command, special command interpreter is prepared, which considers (1) the line started by `*` as an comment,

and (2) the line whose preceded line ends with tt + as an continuation line. The input data has following structure.

command_name option_name(*option_value*)

The *command_name* should be upper case and placed within first 16 columns of the input line. *option_name* and *option_value* are case insensitive except for the *option_value* for `FILE`, `VOLUME`, `DEVICE` *option* for *INPUT* and `OUTFILE` command. The latter part of the command and the arguments name can be omitted when it can be identified uniquely.

2 Input commands for `STEERING`.

The `STEERING` should be used to control the FULL program execution. The command syntax is described in the previous section. The valid commands are described in the following subsections.

2.1 INPUT

The `INPUT` command defines the input files for dynamic file allocations. The `INPUT` command can be omitted (1) to read input data from a file specified by the JCL file, or (2) to generate Monte Calro data which is designated by specifying `EVENT GEN(`*event_number*`)` command in the `STEERING` section. If more than one `INPUT` command is specified, those files are allocated sequentially according to the order of `INPUT` command.

The command syntax is,

`INPUT`	`FILE(`*file_name*`)`	*options*
	`FORMAT(`*exp_no*`,` *runno_from*`,-` *runno_to* `)`	*options*
	`DST001(`*exp_no*`,` *runno_from*`,-` *runno_to* `)`	*options*
	file_type `(`*exp_no*`,` *fileno_from*`,-` *fileno_to* `)`	*options*

`FILE(....)` : The input file *file_name* is allocated dynamically. The file can be a disk file or a `UMT` file.

`FORMAT(....)` : TOPAZ FORMAT data of the experimental number *exp_no* and run number from *runno_from* to *runno_to* is read. If *runno_to* is omitted, upto the last run of the experiment is read. If both `,-` and *runno_to* is omitted, only the run *runno_from* is read.

`DST001(....)` : Read TOPAZ DST001 tapes. The meanings of the arguments are same as those used for `FORMAT` command.

file_type`(....)` : Read TOPAZ standard strip tapes *file_type* such as `TOPAZ02` or `RANDOM` of the experimental number *exp_no* and file number *fileno_from* to *fileno_to*. If *fileno_to* is omitted, up to the last tapes of the experiment is read. If both `,-` and *fileno_to* are omitted, only the file *fileno_from* is read.

As an *option* of the file allocation, following string can be specified. The options marked by * is not implemented yet.

`BUFNO(`*n*`)`: Specify `BUFNO` (buffer number) parameter of the file allocation. *n* must be integer. Default is 20. A option only for FACOM MSP system.

`OPTCD(`*s*`)`: Specify `OPTCD` (option code) parameter of the file allocation. *s* must be a valid `OPTCD` value of one character. Default is "C". A option only for FACOM MSP system.

`WAIT(`*n*`)`: Specify `WAIT` time in unit of sec when the file allocation failed as the tape is in use by other user. Maximum retry count is 20.

`TYPE(`*type*`)`*: (D=`TBS`) Defines the data format of the input data. Can have any of the following type.

TBS: Files created on the same machine by TBS system.

VBS: Files transfered from FACOM using VBSCOPY utility without data format conversion.

SONY: Files stored on SONY DIR-1000L tapes, with a TOPAZ format.

RMTL: Files FACOM system accessed through RMTL library.

`PATH(`*path_name*`)`*: Defines the directory path name of the file. The character string of *path_name* is substituted in front of the *file_name*.

`DEVICE(`*device_name*`)`*: (C*8) Defines the tape device name.

`VOLUME(`*volume_name*`)`*: (C*64) Defines the tape volume name. Must be given when `DEVICE` is specified.

`POSITION(`*position*`)`*: (I*4) Defines the file position in the file. Must be given when `DEVICE` is specified.

`MOUNT(`*mount*`)`*: (C*8,D=`NOMOUNT`) If `MOUNT` is specified, mount tape prior to open the file.

`DISMOUNT(`*dismount*`)`*: (C*8,D=`NODISMOUNT`) If `DISMOUNT` is specified, dismount the tape when end-of-file is detected.

The example to use `INPUT` command is as follows;

To analyse the DST or FORMAT tape, use INPUT command for the dynamic allocation instead of writing the long DD statement. For example, to read FORMAT tape of Run # 100 to 200 and 300 to 350 of exp#4 data, write

```
INPUT FORMAT(4, 100-200, 300-350)
```

First number is experiment # and the rest is the run #. To read only one run,

```
INPUT FORMAT(4,100)
```

is enough. To read DST tape, write

```
INPUT DST001(4, 100, 300-500)
```

for example.

2.2 OUTFILE

`OUTFILE` command specifies the name of the output file. This command is not implemented yet. The syntax of the command is,

`OUTFILE  FILE(`*file_name*`)  `*options*

The valid *options* are

`PATH(`*path_name*`)`: Defines the directory path name of the file. The character string of *path_name* is substituted in front of the *file_name*.

`DEVICE(`*device_name*`)`: (C*8) Defines the tape device name.

`VOLUME(`*volume_name*`)`: (C*64) Defines the tape volume name. Must be given when `DEVICE` is specified.

`POSITION(`*position*`)`: (I*4) Defines the file position in the file. Must be given when `DEVICE` is specified.

`MOUNT(`*mount*`)`: (C*8,D=`NOMOUNT`) If `MOUNT` is specified, mount tape prior to open the file.

`DISMOUNT(`*dismount*`)`: (C*8,D=`NODISMOUNT`) If `DISMOUNT` is specified, dismount the tape when end-of-file is detected.

2.3 DELETE

The banks defined by `DELETE` command are deleted just before the data is written to the output device. The command syntax is,

```
DELETE  BANK( bank_name )
        RECORD( record_name )
```

bank_name : The name of the bank to be deleted. Up to 64 characters. User does not need to specify exact bank names, as those banks whose first nth characters (n = character length of *bank_name*) are equal to the input *bank_name* are deleted. For example,
`DELETE BANK(Raw_Data)`
deletes the banks both "RAW_DATA:HEADER" and "RAW_DATA:CALORIMETER".

record_name : The record name of the bank. Up to 32 characters. If not specified, `EVENT` record is assumed.

2.4 OUTPUT

This command defines the banks to be written to the output device. Those banks which are not defined by the `OUTPUT` command is not written. The use of this command is not recommended unless user knows exactly what should be written to the output file. If some required banks are not included, the problem may happen in the latter step. The syntax of this command is,

```
OUTPUT  BANK( bank_name )
        RECORD( record_name )
```

bank_name : The name of the bank to be deleted. Up to 64 characters. The same rule as `DELETE` command applies to specifying the bank names.

record_name : The record name of the bank. Up to 32 characters. If not specified, `EVENT` record is assumed.

2.5 REMOVE

The bank specified by the `REMOVE` command is deleted just after the data read from the input device. This command is useful to delete banks for the data reanalysis. The command syntax is,

```
RENAME  BANK( bank_name )
        RECORD( record_name )
```

bank_name : The name of the bank to be deleted. Up to 64 characters. The same rule as `DELETE` command applies to specifying the bank names.

record_name : The record name of the bank. Up to 32 characters. If not specified, `EVENT` record is assumed.

2.6 PARAMETER

The `PARAMETER` command sets various parameters of the job. In the followings, first arguments in ( ) describes the variables types where, I*4, R*4, C*n and F*4 means integer, real, characters of maximum n, and flags, respectively. Flag can be a string of `YES/ON` for meaning `YES` and `NO/OFF` for the meaning `NO`. The command syntax is,

```
PARAMETER   KILL_LEVEL(kill_level)
            MAX_EVENT(max_event)
            DEBUG_FREQUENCY(debug_frequency)
            INFILE(infile)
            OUTFILE(outfile)
            DEVICE(device)
            SIZE(size)
            INNIT(inunit)
            OUTUNIT(outunit)
            TIME_LEFT(time_left)
            RUN_TIME(run_time)
            PRINT_CONSTANT(print_constant)
            DELETE_BANK(delete_bank)
            WRITE_ALL_RECORDS(write_all_records)
```

kill_level : (I*4,D=-9) When the `NRET` of analysis module is less than this value, the event analysis of the following module is abandoned and data is not written to the output file.

max_event : (I*4,D=99999999) The maximum number of event to be processed.

debug_frequency : (I*4, D=99999999)If the number greater than 1 is specified, only every *debug_frequency*-th events are debugged.

infile : (I*4, D=1) Number of input files specified by DD statement of JCL cards. See examples for detail.

outfile : (I*4, D=1) Number of output files specified by DD statement of JCL cards. When the output data size exceeds the `SIZE`, write to the next file.

device : (C*8, D=`''DISK''`) The output device type, should be `CTL` or `DISK`.

size : (R*4, D=-1.0) The size of the output device. The job is stopped when the output data size exceeds *size* or write to the next file, if *outfile* is greater than 1. If *size* is negative, does not check output data size.

inunit : (I*4, D=10) Logical unit umber of input file. The default is 10.

outunit : (I*4, D=21) Logical unit number of output file. The default is 21.

time_left : (R*4, D=15.0) When the remaining CPU time of the job at the end of the event analysis is less than this value, job is terminated. The unit is sec. The default value is 15 sec.

run_time : (R*4, D=99999.0) The maximum CPU time of the job, in a unit of sec.

write_all_records : (F*4, D=`''NO''`)If `YES` is specified, the begin_run records and end_run record are not output if no event of the run does not output.

2.7 EVENT

The `EVENT` command is used to control the sequence of data analysis. This command works based on the information stored in the bank "Raw_Data:Header" bank. If more than one run is included in a file, the use of event-number-based command may cause unwanted results because same event number can appear during the execution, and users are recommended to use this command with caution. The syntax of the command is,

```
EVENT  FIRST_EXP(first_exp)
       LAST_EXP(last_exp)
       FIRST_RUN(first_run)
       LAST_RUN(last_run)
       FIRST_EVENT(first_event)
       LAST_EVENT(last_event)
       DEBUG_FIRST(debug_first)
       DEBUG_LAST(debug_last)
       OUTPUT_FIRST(output_first)
       OUTPUT_LAST(output_last)
       GEN(no_of_event)
```

first_exp : First experimental number to start data analysis. If the experimental number of the event obtained from the input file is less than *first_exp*, the event is skipped without calling any analysis module.

last_exp : The analysis is terminated when the experimental number of the event is greater than *last_exp*.

first_run : First run number to start data analysis.

last_run : The analysis is terminated when the run number of the event is greater than *last_run*.

first_event : First event number to start data analysis.
last_event : The analysis is terminated when the event number of the event is greater than *last_run*.
debug_first : First event number from which the debug flags are validate.
debug_last : The event number at which the debug flags are invalidate.
no_of_event : Use this option to generate event in FULL system itself, instead of reading data from the file.

2.8 DATA

`DATA` command is another way to specify the options available by `EVENT` command.

2.9 STEERING

`STEERING` command is exists for historical reason. It is used to provide the debug information, namely to control the printout message. The format of the command is,

`STEERING DEBUG(`*debug*`)`

The value of *debug* may be three-digit number, each digit having meanings as shown in the following table.

3 DESCRIBES for TOPAZ data analysis.

3.1 Names of DESCRIBE

For the TOPAZ data analysis, standard DESCRIBE routines are prepared. Their names and functions are summarized in the table 4.

3.2 Parameters

For TOPAZ standard DESCIBEs, `LEVEL` and `DEBUG` parameters and the version values of calibration constant can be specified in the input data. The format of the command is,

module_name `LEVEL(`*level*`)`
`DEBUG(`*debug*`)`
`CONSTANT(`*version*`)`

`CONSTANT` is used specify the version number of the calibration constant for the analysis. If not specified, the experiment and run dependant constant version number is assigned based on the data base information. If positive number is specified, it is interpreted as an version number of the constant and the corresponding version's constant data is read from the constant data base file. If negative number is specified, it's absolute

Table 1: Meaning of first digit of `DEBUG` parameter. The information shown in the first column is printed when the cell corresponding to the *debug* value is marked "Y"

Value of *debug*	0	1	2 or 3	4	5	6	7	8
Job title, Job end information	Y	Y	Y	Y	Y	Y	Y	Y
Input parameter and record statistics		Y	Y		Y	Y	Y	Y
Job error summary			Y				Y	Y
Run start message				Y	Y	Y	Y	Y
Run end message						Y	Y	Y
End run statistics								Y

Table 2: Meaning of second digit.

Value of *debug*	0	1	2	3
Event header		Y	Y	Y
CPU time for each module per event			Y	Y
Event header for unprocessed events				Y

Table 3: Meaning of third digit.

Value of *debug*	0	1
Generator information		Y

number is considered as an FORTRAN logical unit number and the constant data is read from the corresponding files, which are usually specified in the JCL statement.

The example to specify `CONSTANT` is as follows.

When you write,

```
PRTPC CONSTANT(3)
```

TPC production constants of version # 3 is used for all run and when you write

```
PRTPC CONSTANT(-35)
```

program reads the constants for TPC production step from the file allocated to the FORTRAN logical unit # 35 in JCL. "`PRTPC`" may be `RDIDC, RDTPC, RDBTOF, ... PRIDC, PRBTOF, PRBCL,..` according to the type of constants (FULL module). If you don't write these commands, the program gets the version # from RUNLIST data base.

Version # used by the analysis are saved in the bank "Full_Analysis" of Begin Run record. Their element # is 3 + 100*(ITRY-1), where ITRY is the number of times processed data. (Usually, element # = 3 for official DST pass1 data.)

The meanings of the data in this element is,

Table 4: TOPAZ standard DESCRIBE routine and their function.

module name	Subroutine name	function	
RDBCL	DSRBCC	BCL reduction	(calibration)
	DSRBCH		(hit point making)
RDECL	DSRECC	ECL reduction	(calibration)
	DSRECH		(hit point making)
RDTPC	DSRTPS	TPC reduction	(sorting)
	DSRTPC		(calibration)
	DSRTPT		(tracking)
RDBTOF	DSRBTC	BTOF reduction	(calibration)
	DSRBTH		(hit point making)
RDBMU	DSRBMC	MDC reduction	(calibration)
RDTCH	DSRTCH	TCH reduction	(hit point making)
	DSRTCC		(calibration)
RDFCL	DSRFCL	FCL reduction	(calibration)
RDVTX	DSRVXC	VTX reduction	(calibration)
RDFLT	DSRFL1	Reduction event filter - 1	
	RDFLT2	Reduction event filter - 1	
PRTPC	DSTPSP	TPC production	(space point making)
	DSTPTR		(tracking)
	DSTPLW		(link wire)
	DSTPDP		(duplicate removal)
	DSTPDE		(dE/dx)
	DSTPRF		(distortion correction)
PRIDC	DSIDC	IDC production	(tracking)
PRTCH	DSTCH	TCH production	(tracking)
PREDC	DSEDC	EDC production	(tracking)
PRBDC	DSBDC	BDC production	(tracking)
PRBTOF	DSBTOF	BTOF production	
PRBCL	DSBCL	BCL production	(clustering)
PRECL	DSECL	ECL production	(clustering)
PRFCL	DSFCL	FCL production	
PRBMU	DSBMU	MDC production	
PRVTX	DSVTX	VTX production	
DSSIML	SIMULATOR	Detector simulation	
DSSTTP	TOPAZ01	TOPAZ01 strip	
DSBCSM	BEAM_COSMIC	Beam-cosmic strip	
DSRAND	RANDOM_TRIG.	Random trigger event strip	
DSFCEV	FCL_EVENT	FCL event strip	

```
IBUF( 1) = Version # used for IDC reduction step.
    ( 2) =                     TPC
    ( 3) =                     BTOF
    ( 4) =                     BDC
    ( 5) =                     BCL
    ( 6) =                     BMU
    ( 7) =                     EDC
    ( 8) =                     ECL
    ( 9) =                     LUM
    (10) =                     Filter
    (11) = Version # used for IDC Production step.
    (12) =                     TPC
    (13) =                     BTOF
    (14) =                     BDC
    (15) =                     BCL
    (16) =                     BMU
    (17) =                     EDC
    (18) =                     ECL
    (19) =                     LUM
    (20) =                     Steering.
```

The meanings of *level* and *debug* parameter depend on the analysis module. The meanings of `LEVEL` parameter for IDC production module is shown in Table 5.

Table 5: The meanings of `LEVEL` parameter for IDC production

LEVEL	Meanings
1	Analyze using exact hit information.
2	Analyze using raw data bank, compare results with simulator.
3	Analyze using raw data bank.

3.3 Input data for simulator module

The input data format for the TOPAZ detector simulator module (`DSSIML`) is same as those used for FULL steering module. Some of the useful commands are summarized below.

3.3.1 SEED

This command defines the initial seed for the random variables used in the detector simulator. The command syntax is,

```
SEED  FATE( fate )
      MS( ms )
      SHOWER( shower )
      SHOWER_MS( shower_ms )
      DETECTOR( detector )
```

fate : (I*4,D=769999997) Initial seed of random variable for particle life time generation.

ms : (I*4,D=304859885) Initial seed of random variable for the generation of the multiple scattering angle in the low density material.

shower : (I*4,D=68569992) Initial seed of random variable for the EGS and GEISHA program except multiple scattering, which are used to simulate particle shower in the massive material region.

shower_ms : (I*4,D=666666666) Initial seed of random variable for the multiple scattering of the EGS and GEISHA program.

detector : (I*4,D=766767676) Initial seed of random variable for the detector signal generation.

3.3.2 FLAG

Use this command to printout the constants for detector simulator, turn on/off multiple scattering and energy loss and so on. The command format is,

```
FLAG   PRINT(print)
       MS(ms)
       ENERGY_LOSS(energy_loss)
       SAVETTL(savettl)
```

print : (C,D=`NONE`) Use this option to specify the printout information. `print` can have the value of `NONE, GEOMETRY, MATERIAL, PARTICLE,` or `ALL`. When any of these values are specified, the program prints nothing, detector geometry, material definition, particle properties, and all information, respectively.

ms : (F,D=`ON`) If `OFF` is specified, the effect of multiple scattering in low density material. However, there is no way to turn off the effect of the multiple scattering in the EGS and GEISHA multiple scattering.

energy_loss: (F,D=`ON`) If `OFF` is specified, the energy loss in the low density material is turn off.

`savettl`: (C,D=`SELECT`) As a default simulated track information which does not go into the sensitive region of the detector is not output. However, if `ALL` is specified for *savettl* all simulated track information is output

3.3.3 PARAMETER

Use this command to set The command format is, various parameter for the simulator. The format of the command is,

```
PARAMETER   FIELD(field)
            DX(dx)
            DY(dy)
            DZ(dz)
```

field : (R*4,D=12.0 (k-gauss)) The `field` specifies the solenoid magnet field.
dx : (R*4,D=0.01cm) The `dx` specifies the rms width of the interaction point along X axis. Usually, generators generate the event at the coordinate (0., 0., 0.). But, actual event distribute depending on the beam width. Therefore, at the very beginning of the simulator, the origin of the all track are smeared by the gaussian distribution with the width, (*dx, dy, dz*).
dy : (R*4,D=0.01cm) Same as *dx*, but for Y direction.
dz : (R*4,D=1.0cm) Same as *dx*, but for Z direction.

3.3.4 EVENT

`EVENT` command is used to control the event information of simulated data. The command format is,

```
EVENT   FIRST(first_event)
        LAST(last_event)
        DEBUG_FIRST(debug_first)
        DEBUG_LAST(debug_last)
        OUTPUT_FIRST(output_first)
        OUTPUT_LAST(output_last)
```

first : (I*4,D=1) First event number to start simulation.
last : (I*4,D=1000000) Last event number to terminate simulation.
debug_first : (I*4,D=1) Event number where to turn on debug flag.
debug_last : (I*4,D=1000000) Event number where to turn off debug flag.
output_first : (I*4,D=1) Event number where to output simulation result.
output_last : (I*4,D=1000000) Event number where to turn off simulator output.

3.3.5 BCL

Use `BCL` command to specify the type of the simulation of BCL calorimeter. The format of the command is,

```
BCL   LEVEL(level)
```

The first digit of level is a switch for the inclusion of the effect of the effect of electromagnetic and hadronic shower.

First digit of *level*	Elemag. shower	Hadron shower
2	on	off
3	off	on
4	on	on

If the second digit of *level* is 1, BOOTSTRAP method is used for shower simulation to save CPU time. If it is 0, no BOOTSTRAP method is used. If the third digit of *level* is 1, turn off signal smearing. If it is 2, turn off module correction.

3.3.6 TPC

The `LEVEL` parameter of `TPC` command is used to control output information. The format of the command is,

`TPC  LEVEL(`*level*`)`

The meanings of `level` is summarized in the following table.

level	meanings
0	Use default swimmer
1	Output only exact hit information.
2	Output only exact hit and sorted raw data information.
3	Output exact hit, sorted raw data, and raw data bank for TPC.