JULIA User's Guide
and
ALEPH Programmers hints

J. Knobloch, E. Lançon

September 29, 1994

Abstract

An introduction to the usage of the ALEPH reconstruction program JULIA (Job to Understand Lep Interactions in Aleph) is given. The data cards for running JULIA are described.

Rules and hints for programming JULIA in particular and ALEPH off-line in general are given.

An up-to-date version of this document is available via WWW in HTML format [1] and in a Postscript file[2] as well.

This note replaces the previous version [1] of the JULIA user's guide (ALEPH note 90-115/SOFTWR 90-011).

[1] http://alephinfo.cern.ch/aleph$general/jul/doc/julia.html
[2] http://alephinfo.cern.ch/aleph$general/jul/doc/julia.ps

Contents

1 Introduction

JULIA is the ALEPH reconstruction program. All files referring to this program are kept in the JUL directory on VAX, JUL disk on IBM and in /al/reference/jul and /aleph/jul on ALEPH UNIX systems (Aloha, CSF, Shift9) [3]: Fortran, historian, object, executable and command files.

Input to JULIA are raw data produced by the ALEPH detector, or by the ALEPH simulation program GALEPH[7]. JULIA needs as extra input the ALEPH database.

The current version of JULIA is 2.74. In all files referenced the sequence '274' is used. For any other version, replace '274' by the desired version number.

A number of interactive tools have been written to access informations referring to the program (JTREE, SBANK, SCOM). Type HELP 'tool' to get more details about them.

Listings of the program, variables in commons, BOS banks, tree outputs can be found in the software documentation office (2-R-032).

2 Running JULIA

Basically all you need to run JULIA in the standard mode, is a card file containing a FILI and a FILO card [4]. Assuming the file JULIA.CARDS contains the following machine independent lines :

```
FILI 'julia_input.EPIO'
FILO 'julia_output.EPIO'
ENDQ
```

To run the program one needs :

On VAX :

```
$ ASSIGN JULIA.CARDS FOR099
$ RUN JUL:JUL274
```

On IBM :

```
'FILEDEF 99 DISK JULIA CARDS A'
'EXEC GIME PUBXU 210 *'      /* the JUL disk */
'EXEC CERNLIB JUL274 ALEPHLIB BOS77 GENLIB'
'LOAD CORR274 (CLEAR NOMAP NOAUTO RESET AAMAIN START'
```

On UNIX :

```
> setenv JULIACARDS JULIA.CARDS
> /aleph/jul274
```

[3]For mode details, refer to the ALNEWS 203 from F. Ranjard in the UNIX_OFFLINE folder.

[4]If you dont know what a FILI and a FILO card are, refer to the ALPHA manual [8] or go to 3.1.

If you only need to run JULIA *in the standard mode, you can stop* ***here*** *.*

To run JULIA with new or modified FORTRAN routines, a command file `JUL:JULRUN.COM` on VAX (by F.Ranjard) and an EXEC file `JULRUN EXEC` on IBM (by R.Edgecock) may be used, to set up a JULIA job in an interactive dialog, no tool exists on UNIX systems for the moment.

3 Data Cards

The data cards used by JULIA can be generated automatically, by the JULRUN EXEC on the IBM or by JULRUN on VAX. One can also start from the `JULIA.CARDS` file (`JULIA CARDS I` on IBM) and edit it.

The minimal set of data cards for reading a data set, performing complete event reconstruction, and writing a standard output consists just of the two cards `FILI` and `FILO`.

The rules for the syntax of the data cards (comments, continuation cards) are given in the BOS-manual [2].

In the section 'Overview' a one-line comment for every card is given. Apart from cards described in this section, cards replacing a bank from the data-base can be used. The bank number on the data card should be -1 in order to take precedence.

3.1 Overview

File definitions

- `FILI` defines the event input file (see 3.2)
- `FILO` defines the standard output file POT (Production Output Tape) (see 3.2)

- `MONI` writes `MONI` file for event reconstruction (see 4.3) monitoring (Vax only)
- `SUMR` writes `SUMR` file (see 4.3) for run reconstruction summary (Vax only)
- `FDBA` defines an alternate version of the data-base to be used (see 3.2)
- `HIST` defines a PAW display file (see 3.3.1)

Dealing with BOS banks

- `ADDD` adds banks from the output stream (see 3.2)

- `REMD` removes banks from the output stream (see 3.2)
- `REMI` will remove banks from the input file before processing
- `UDAF` uses constants banks from tape rather than data-base
- `COMP` selects data compression. JULIA default is to compress integers only.
- `NGAR` do NOT perform BOS garbage collection between program modules
- `JCAR` saves an image of data cards used for processing in `JCAR` run header bank (see 4.3)

Run/event selection

- `SEVT` Selects run and event numbers to be processed (see 3.4)
- `SRUN` Selects runs to be processed (see 3.4)
- `IRUN` Defines runs to be ignored (see 3.4)
- `NEVT` Defines number (or range) of events to be processed (see 3.4)

Printing

- `HPRI` Prints HBOOK histograms at end of job (see 3.3.1)
- `ERRP` defines the number of times any single error is printed (see 3.3.1)
- `DEBU` selects events to be debugged (see 3.3.1)

Steering

- `PROC` defines which JULIA module (process) to be run (see 3.5)
- `REPR` Selects POT to POT re-processing mode (see 4.2)
- `PASO` Selects PASS 0 running mode (see 4.1)

Special reconstruction options

- `FIEL` defines central value of ALEPH magnetic field (override value from tape)
- `TOGL` defines TPC Tzero (normally taken from database)

- PASC defines PASS0 running conditions (normally taken from database) (see 4.1)
- TCLK defines TPC clock period (normally taken from tape)
- TDRP drops TPC sectors from reconstruction
- TOPT selects special options for TPC reconstruction
- INOZ disables z-reconstruction in ITC
- IMAT enables additional ITC-TPC track matching after pattern
- SCAU calibrates SATR x vs t
- JTOF disables jet finding in energy-flow
- MCUT override default cuts for muon reconstruction
- NESC do NOT use platines for ECAL module calibration
- NEWT do NOT eliminate ECAL towers if wire signal of module below threshold
- NGAM do NOT perform GAMPE(C,Q,X) photons reconstruction
- NSCA do NOT apply calibration constants to SiCAL data
- BPOS writes beam position bank (ALPB) on separate file (see 4.3)
- VREC performs VDET tracking even if too few tracks in event
- etc...

3.2 Input Output

On the cards FILI, FILO, FDBA, and HIST file definitions are given. The files containing events can be of either of the formats supported by BOS: EPIO (machine independent, transport format), or NATIVE (machine dependent, faster on VAX). The card format is always the same. Only the first two cards are needed for standard reconstruction.

For a complete description of the FILI, FILO, FDBA, and HIST cards files refer to chapter 4 of the ALPHA manual [8].

The cards ADDD and REMD allow to add or remove banks from the output stream.

The card REMI will remove banks from the input file before processing.

- FILI

  On the FILI cards the input file is described (EDIR are accepted). If several files are to be read in sequence, several FILI cards can be given.

- FILO

  Card to specify where the POT (Production Output Tape) is to be written. A list of the banks written to the POT-file is printed at the start of job. Banks currently written to POT file :

```
EVEH ASEV BCNT BHIT BOMB BOME BOMP BOMR BPTR D4CD DHEA ECRQ ECTE EFOL EGPR
EIDT EJET EROR ERRF ETDI ETKC EWHE FKIN FPOI FPOL FRID FVER FZFR HCCV HCTE
HINF HLTU HMAD HPDI HROA HTUB IXTR IZDB JBER JEST KEVH LIDT LOLE LUPA MCAD
MHIT MTHR MUDG MUEX MUID OSTS PASL PCHY PCOB PCOI PCPA PCRL PECO PEHY PEMH
PEPT PEST PEWI PFER PFHR PFRF PFRT PGID PGPC PHCO PHHY PHST PIDI PITM PLPD
PLSD PPDS PPOB PPRL PRPW PSCO PSPO PT2X PTBC PTEX PTNC PTST PYER PYFR REVH
RTLO RTLS SCLS SFTR SILH SILU SIXA SPDA VCOM VCPL VDCO VDHT VDTD VDXY VDZT
VFHL VFLG VFPH X1AD X1ER X1RG X1TI X2DF X3EC X3EW X3HC X3IT X3L2 X3LU X3TM
X3TO X3TP X3X3 XTBP XTCN XTEB XTRB YVOV
```

  Note : The KINE, VERT, KHIS and KVOL banks of a Monte-Carlo raw data file, are converted to FKIN and FVER tabular banks by JULIA if not already done by GALEPH. In addition, KPOL (resp. KZFR) is converted to FPOL (resp. FZFR) to get a coherent scheme[3].

- FDBA

  Normally JULIA uses the latest version of the ALEPH database. To use an old version instead, or a test version, the FDBA card may be used.

  To use the test version of the data base:

  On IBM: FDBA 'ADBSTEST DAF * | GIME DBASE'

  On VAX: FDBA 'DBASE:ADBSTEST.DAF'

  On UNIX: FDBA '/aleph/dbase/adbstest.daf'

- ADDD

  Additional banks may be put onto the POT-file using the ADDD 2/ card. Example :

```
ADDD 2/ 'PHANTASTIQUE' ! will add the banks PHAN, TAST and IQUE to the POT
```

- REMD

  With the REMD card banks can be removed from output:

```
REMD 2/ 'FVERFKIN' ! Do not write FKIN and FVER to POT.
```

3.3 Printing and Histogramming

3.3.1 General data cards

- ERRP

With an `ERRP` card the number of times any single error is printed out (reported via `RERROR`) can be controlled. Anyhow the total number each error has occurred will be printed in the job summary.

Example:

```
ERRP 3  ! print any error not more than 3 times
```

- `HPRI`

  If the `HPRI` card is present histograms will be printed at the end of job.

- `HIST`

  Card to define a PAW readable file.

  Example:

```
HIST 'SCWEEK:MYJOB.HIS' ! Histograms written to file SCWEEK:MYJOB.HIS
```

- `DEBU`

  The two numbers following this card will specify for which events you want to get printout. This numbers correspond to the event sequence number.

  Example:

```
DEBU 1 50   ! Debug events from number 1 to 50.
```

3.3.2 Detector specific data cards

Each process is controlled by a card. The name of these cards must correspond to the accepted list of 4 character names (corresponding to each JULIA modules):

`VDET, ITC, TPC, ECAL, LCAL, SATR, HCAL, MUON, CALO, YREC,`
`YTOP, EFLO, BCAL, FALC, BOM, SICA and SKEL.`

These cards are used to select histogram, and printout level that you want for each detector or module, specifying the keywords '`HIST`' or '`PRIN`' followed by the level number. If a non-existing keyword is specified an error message will be issued and the keyword ignored. If you put a card for a detector which is not selected in the `PROC` card, it will be ignored. The printout levels are defined as follows:

- **Level 0 :** no printout
- **Level 1 :** 1 line per event and per task (track finding, cluster finding,)
- **Level 2 :** 1 line per object (track, cluster, vertex,...)
- **Level 3 :** 1 line per cell (storey, TPC coordinate,...)

- **Level 4 :** very detailed.

Other levels will depend on every detector. To the list of detectors we have added the SKELeton, so you can select printout and histogram level for it. The levels for the SKELeton are : 1 to get printout for every run, 2 to get it for every event. The 'SKEL' keyword in the PROC card is not mandatory.

The end of job summary printout for each detector is driven by the 'EPRI' keyword. Example:

```
TPC  'HIST' 4 'PRIN' 2 'EPRI' 1
```

This will produce level 4 histogramming ,level 2 printing for the TPC and level 1 end of job summary. To print histograms at the end of the job a card HPRI is needed. Histograms will be written onto a PAW readable file using the HIST card.

3.4 Run event selection

For the selection of events one can refer to two kinds of numbers.

1. The sequence number of the event in the input stream
2. The event trigger number given once by online or GALEPH.

The first one is used on the cards NEVT and DEBU, the second one on SEVT.

- SEVT

  Event selection card. The first number corresponds to the run number. Then, you can specify any number of events. If you want to specify a range, the upper limit must have a negative sign. This card has priority over SRUN and IRUN. You can have as many SEVT cards as you want.

  Example :

  ```
  SEVT  10 45 56 -70 101
  ```

  Events 45, 56 to 70 and 101 in run 10 will be reconstructed.

- SRUN

  Run selection card. Here you can specify the run numbers you want to process. If you want to run in a specific range, the upper limit must have a negative sign. This card has priority over IRUN but will be ignored if you specify SEVT.

  Example :

  ```
  SRUN 1 -60 120
  ```

  Runs from 1 to 60 and run 120 will be reconstructed

- IRUN

  Runs to ignore. Here you can specify the run numbers you don't want to process. If you want a specific a range, the upper limit must have a negative sign. This card will be ignored if you specify SEVT or SRUN.

  Example :

  ```
  IRUN 1 -60 120
  ```

  Runs from 1 to 60 and run 120 won't be reconstructed

- NEVT

  Select a range of events according to the sequential number

  Example :

  ```
  NEVT  1 -50     ! Reconstruct the first 50 events
  ```

3.5 Job steering

Job steering card may be used to modify the standard running conditions of JULIA.

The PROCess card (PROC) contains the list of processes to be run. Process names can be in any order, but must correspond to the accepted list of 4 character names :

```
VDET, ITC, TPC, ECAL, LCAL, SATR, HCAL, MUON, CALO, YREC,
YTOP, EFLO, BCAL, FALC, BOM, SICA and SKEL.
```

If there is no PROC card, all processes will be run. The 'SKEL' keyword doesn't need to be in the PROC card. Example:

```
*------ Detectors to be processed -----------------------------------
PROC 'SKEL', 'VDET', 'ITC ', 'TPC ', 'ECAL', 'CALO', 'HCAL', 'MUON'
     'YREC', 'YTOP', 'LCAL', 'SATR', 'BCAL', 'BOM ', 'SICA'
```

With this card everything will be reconstructed.

4 Special running modes of JULIA

JULIA can also be run in a pre-processing mode, called PASS 0 or in the POT to POT re-processing mode.

4.1 PASS 0 mode

This mode is usually used on FALCON and serves as pre-processing. In this mode, the TPC drift velocity and the VDET hot channels list can be determined.

4.1.1 Introduction

The steering is equivalent to :

```
*------ Detectors to be processed ---------------------------
PROC 'SKEL', 'VDET', 'TPC '
```

The PROC card is not needed, however only one of the two PASS 0 actions can be performed by selecting the corresponding detector my means of a PROC data card. For the TPC, the dE/dx reconstruction is not performed and the Kalman filter is not run. For the VDET, only the clustering is performed.

This mode requires two input files and produces one output file. The input files are :

- the raw data file, described in the FILI card
- the PASS 0 reference file which is a BOS card file, described in the READ card in the following example.

The output file, is a BOS card file, described in the PASO card. Below, is an example of JULIA data cards used on FALCON for PASS 0 running mode.

```
FILI 'RAW.NATIVE | NATIVE'
PASO 'OUTPASS0.CARDS'
READ 'REFPASS0.CARDS'
* PASC allow to change  PASS0 cuts , PASC is on the DAF
*
* first parameter : minimum number of 'good for pass0' events needed
*                   to update the drift velocity
* second          : maximum number of 'good for pass0' events used
*                   to update the drift velocity
* third           : maximum allowed elapsed time since last laser
*                   measurement to use vz_laser is there is too few events
* fourth          : maximum allowed absolute value for delta-t0 (mus)
*                   in order to update the drift velocity
*
*PASC -1 / 4 1 50  250 -1. 0.200
ENDQ
```

4.1.2 How does it works

The purpose is to determine the TPC drift velocity along the z axis. Events are reconstructed with a *guess* of what this drift velocity should be, the *guess* should not be too far away from the real value otherwise the reconstruction would fail.

Then by selecting *good* tracks, and by measuring for those tracks the mean z impact parameter on each half of the TPC, it is possible to determine, knowing the input drift velocity to JULIA (the *guess*) the *real* drift velocity value : the new one.

Good tracks are selected, according to the following criteria (those lines are taken from a standard JULIA log file):

```
TPC TO MONITOR initialized on first call
-----------
TPC-T0 : Default cuts used **********
1) min. number of fitted tracks on one side....          2
2) min./max. lambda for a track (degrees).......      0.000 /     50.000
3) max. allowed d0 (cm)........................     2.000
4) max. allowed z0 (cm)........................    10.000
5) max. invers radius..........................   0.00200
6) number of required hits on a track..........        10
```

The relation between, the old and the new drift velocity is given by the formula :

$V_{new} = (T0_{new} - T0_{old} + \Delta t) * V_{old}^2/220. + V_{old}$

where 220. is the TPC half length, $T0$ stands for the TPC $T0$ and V for the TPC drift velocity along the z axis. Δt being half the difference along the z axis, between the vertices found in the two TPC halves.

One should note that it is impossible with this method to distinguish between a change in the drift velocity and a change in the TPC $T0$. Usually, the $T0$ value is obtained by other means and one simply states that $T0_{new} = T0_{old}$. However, any brutal change in the TPC drift velocity as a function of time would indicate a change of $T0$ value.

The output of JULIA is a card file (`OUTPASS0.CARDS` in the previous data card example). This card file contains,

- a short JULIA summary,
- the PASS 0 summary bank (`JPAS`), which will go onto the POT run header after the JULIA processing,
- the new TPC drift velocity bank (`TPDV`),
- the new VDET hot channel bank (`VHOT`).

The running of JULIA in PASS 0 mode, is driven by the `PASC` data base bank. A minimum number of events is required to allow for a new determination of the drift velocity etc.. If the conditions are not met, the drift velocity is taken from the previous run for which the conditions were met (usually it is the input card file `REFPASS0.CARDS` in the previous example). If the conditions are met, on FALCON, the `OUTPASS0.CARDS` is renamed in `REFPASS0.CARDS` and used as input for the next run.

Example of an output of PASS 0, produced on FALCON :

```
PON
*PASS0 --- Summary for Run  27639 -  823 Z read
*PASS0 Date  940725 Time    1221
*PASS0 Julia 274.02 Alephlib  20.10 Daf   191 from  940624
*PASS0 Min. nb of evts :    50 - TLAS max. elapsed time :   -1.00 h.
*PASS0 TLAS too OLD :   280. hours
*PASS0 T0 difference, 0.5*(TPC end-A - TPC end-B) for   377 events
```

```
*PASS0 delta-T=   0.00284 +- 0.00179 mus; rms= 0.00121 mus
*PASS0 Number of VDET HOT channels :    70 Previous run :    65
*PASS0 TPC Vz ref. from Run  27638 VHOT from Run  27638
*PASS0 Vold 5.24144 Vnew 5.24179+- 0.00022 Dv/v -0.00007
*PASS0 ----> Reference Drift Velocity updated <----
 * START OF TEXT-RECORD
 JPAS $FMT 2I,(3I,9F,4I)
 JPAS 27639/ 17 1 823 377 50 0.00284068 0.00178954 0.00120732 5.24491
 5.24144 5.24179 223.47E-6 -1. 279.666 940725 1221 999999 65 70
 TDPV $FMT 2I,(4F)
 TDPV 27639/ 4 2 -900.E-6 0.0035 5.24179 223.47E-6 900.E-6 -0.0035
 5.24179 223.47E-6
 VHOT $FMT 2I,(I)
 VHOT 27639/ 1 70 7080433 11799026 10324000 23169057 29198370 29460515
    .......
 ENDQ
```

4.2 POT - POT re-processing mode

JULIA input can also be a POT file instead of a raw data file, this allows for partial reprocessing of the events (new calibration constants, new detector alignement,...). The dE/dx cannot be redone!. To select this mode, the following data card has to be added to the standard JULIA data cards :

```
REPR
```

Although, JULIA could have determined by itself if the input file is a POT or a raw data file, some initializations are being done before reading the first event, this should be corrected in the future.

4.3 Special cards

- BPOS

  this card is used to write the ALPB bank (Aleph Beam position), into a special file, in data card format. It is the result of Dave Brown and George Redlinger procedure [4]. Format :

  ```
  BPOS 'ALPB_for_this_run.cards | CARD'
  ```

- JCAR

  This card is used to save onto the run header (into the JCAR bank), an image of the data cards used for the processing.

  ```
  JCAR 0 / $TEXT
  *... Every thing here will be
  *... saved on the run header in
  *... the JCAR bank
  END$
  ```

- SUMR

  This card described where the julia run summary banks (luminosity, beam position, statistics, etc..), have to be written. Format :

  ```
  SUMR 'SUMR.NATIVE | NATIVE'
  ```

  This SUMR.NATIVE file will contain the following banks :

  ```
  RUNR ALPB LEHI LUMI LBAK LVHI JSUM
  SLUM RLEP JEDS JPAS JTRE JHDX JCON
  ```

  On FALCON, after JULIA processing, the SUMR file is read and its content is inserted onto the POT before the first event to produce the final POT.

- MONI

  This data card describe where to write a special monitoring file used on FALCON only. Format :

  ```
  MONI 'MONI.NATIVE | NATIVE'
  ```

4.4 Running on FALCON

On FALCON, JULIA is run twice on a given raw data file, one in PASS 0 mode and one in standard mode. In PASS 0 mode, the TPC drift velocity is being determined for the run (the drift velocity from the previous good for PASS 0 run is used as input, the TDPV bank), the VDET hot channels (VHOT bank) are also flagged. The output of PASS 0 (OUTPASS0.CARDS) is then used as input for the standard JULIA run. Below is given an example of JULIA cards used on FALCON for the second step.

```
JCAR 0 / $TEXT
FILO 'POT.NATIVE  | NATIVE'
SUMR 'SUMR.NATIVE | NATIVE'
FILI 'RAW.NATIVE  | NATIVE'
READ 'OUTPASS0.CARDS'
*---------------------- START of FALCON CARDS ------------------------------
JOBC 'TIME',60
*------ Detectors to be processed ------------------------------------------
PROC 'SKEL', 'VDET', 'ITC ', 'TPC ', 'ECAL', 'CALO', 'HCAL', 'MUON'
     'YREC', 'YTOP', 'LCAL', 'SATR', 'BCAL', 'BOM ', 'FALC', 'SICA'
     'EFLO'
*------ Histograms and print level flags ----------------------------------
SKEL 'HIST' 0 'PRIN' 2 'SPRI' 0 'EPRI' 2
VDET 'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
ITC  'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
TPC  'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
```

```
ECAL 'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
CALO 'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
HCAL 'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
MUON 'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
YREC 'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
YTOP 'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
LCAL 'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
SATR 'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
BCAL 'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
BOM  'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
SICA 'HIST' 0 'PRIN' 0 'SPRI' 0 'EPRI' 0
*-----------------------------------------------------------------------------
HPRI
UDAF            ! Use UDAF for Normal processing, no UDAF for re-processing
ERRP 1          ! print any error not more than one time
GARB            ! perform garbage collection between julia modules
COMP   'INTE'   ! compress only integer numbers
*------ missing formats ------------------------------------------------------
*IZDV $FMT 'I'   ! Example
*----------------------- END   of FALCON CARDS ------------------------------
POFF
END$
```

JULIA can also be ran without PASS 0, in that case, the TPC drift velocity is taken from the TDFV data base bank and the VDET hot channels bank (VHOT) as well.

5 ALEPH Programmers hints

People contributing to the ALEPH off-line software should follow rules which are given in the following sub-topics:

5.1 FORTRAN-rules

The program shall follow the rules given by the standard ANSI FORTRAN77. Additional rules are checked by the FLOPPY program of J. Bunn [5]

5.1.1 Deck structure:

FORTRAN rules to be followed and checked by FLOPPY: (The numbering as used by FLOPPY)

```
No.1,11 Avoid comment lines outside module
No. 39  Order of statements should be:
                1. PROGRAM, FUNCTION, SUBROUTINE
                2. PARAMETER, IMPLICIT, COMMON, REAL, INTEGER,..
                3. DATA
```

4. Statement function declarations
5. ENTRY, FORMAT, Executable statements
6. END statement

5.1.2 Common blocks

FORTRAN rules to be followed and checked by FLOPPY: (The numbering as used by FLOPPY)

```
No.  4  Variables in COMMON should be ordered according to their word-length
        (i.e. COMPLEX and DOUBLE PRECISION first)
No.  5  COMMON block definitions should not change
No. 18  Avoid multiple COMMON definitions per line
No. 19  Do not dimension COMMON variables outside COMMON
```

5.1.3 Names of variables,subroutines,functions and commons

FORTRAN rules to be followed and checked by FLOPPY: (The numbering as used by FLOPPY)

```
No.  7  Variables in COMMON should be 6 characters long
No. 43  Variables (not COMMON, not PARAMs) should be <6 characters
No.  9  Integer variables should begin with I to N
No. 10  Variable names should not equal FORTRAN keywords
No. 12  Module names should not equal intrinsic functions
No. 32  COMMON block names should not equal variable names
No. 34  Avoid FUNCTION names the same as intrinsics
No. 36  Module names should all be different
No. 41  Names used in Statement Function definitions should not be used
elsewhere
```

5.1.4 Comments

FORTRAN rules to be followed and checked by FLOPPY: (The numbering as used by FLOPPY)

```
No. 14  Module should begin with at least 3 comment lines
No. 15  Comment lines should begin with a C
No. 16  No comment lines between continuations
No. 40  Separate Statement Functions by comment lines
```

5.1.5 Coding Style

FORTRAN rules to be followed and checked by FLOPPY: (The numbering as used by FLOPPY)

```
No. 17  Avoid non-standard variable types eg INTEGER*2
No20,21 Avoid embedded blanks in variable names or syntactic entities
No. 22  Avoid the use of PRINT statements (use WRITE)
```

```
No. 23  Do not give the END statement a label
No. 24  Avoid WRITE(* construction
No. 26  Avoid the use of PAUSE statements
No. 27  Statement labels should not begin in column 1
No. 28  Always precede STOP by a descriptive WRITE
No. 37  Avoid expressions of mixed mode eg A=B/I
No. 42  Use LLT,LGT etc to compare CHARACTER vars. in IFs
```

5.1.6 SUBROUTINES and FUNCTIONS

FORTRAN rules to be followed and checked by FLOPPY: (The numbering as used by FLOPPY)

```
No. 29  Avoid the use of ENTRY in FUNCTIONS
No. 30  Avoid using I/O in FUNCTIONs
No. 31  Avoid the use of the alternate RETURN statement
No. 33  Avoid use of obsolete CERN library routines
No. 38  Length of passed CHARACTER variables should be *
No. 44  Passed arguments should be dimensioned * in module
```

5.2 Naming of SUBROUTINES, FUNCTIONS, COMDECKS and BOS-banks

SUBROUTINES, FUNCTIONS, COMDECKS and BOS-banks names shall start with the letter corresponding to the task or sub-detector given in the following table:

```
Single Description                                              4-Letter
Letter                                                          code
========================================================================
A       Aleph Simulation and Aleph Geometry ({\GALEPH})
        General purpose routines in ALEPHLIB
B       BHABHA calorimeter  (Very small angle lumin counter)    BCAL
        Bookkeeping and beam pipe geometry
C       Calorimeter (common to ECAL, HCAL and LCAL)             CALO
D       Display
E       ECal                                                    ECAL
F       Fast tracking (Galeph)
F       Find tracks in more than one detector (Julia)
GU      GUser routines (Galeph)
H       HCal                                                    HCAL
I       ITC                                                     ITC
J
K       Kinematics interface routines to event generator
L       LCal                                                    LCAL
M       MUon chamber                                            MUON
N
O       beam Orbit monitor (BOM)                                BOM
        online banks
```

P	Particle identification	
Q	QUadrupole (Galeph)	
R	Reconstruction steering routines	SKEL
RS	Reconstruction Simulation routines	
S	Small Angle Tracking device	SATR
SI	SICAL	SICA
T	TPC	TPC
U	Utility routines	
V	VDet	VDET
W		
X1	Trigger Level 1 (Galeph)	
X2	Trigger Level 2 (Galeph)	
X3	Trigger Level 3 (Galeph)	
Y	Vertex finding	YTOP
Z		

If any of the unused letters is to be assigned, please call F. Ranjard or J. Knobloch. If in case of a `FUNCTION` the letter does not coincide with the default type (e.g. `REAL FUNCTION ITCFUN(DUMMY)`) the type has to be declared explictly in the function declaration and in the calling routine. In any case should the first letter correspond to the above list.

5.3 BOS-banks

BOS is the memory manager chosen for ALEPH software. As data structure we use the entity-relationship model. The Bos manual [2] is available in the ALEPH documentation office.

There are two types of banks in the system: Named banks and work banks. All named banks shall be described in the form of DDL (Data definition language). Any named bank created shall have its FORMAT defined by a call to `BKFMT`. Any bank should have a mini-header of two integer numbers giving the number of columns and the number of rows of the table.

5.3.1 Lists of BOS-Banks

There are 5 lists available in the BOS system : C, E, R, S, T. A list is a collection of bank names, a bank name can appear in various lists. A bank name can be on a list even if the bank does not exist. A bank name can be deleted from a list without dropping the bank. We use the following conventions:

1. **'C' list**

   means Constants list. This list contains the names of all banks containing constants valid for more than one event (usually for a run). The banks which names appear on the 'C' list are written on the output before the 1st event. Then the 'C' list is reset to '0', but the banks of constants are kept.

2. **'E' list**

   means Event list. This list contains names of banks which belong to the current event (input or output). A bank can be added to the 'E' list by a process or can be deleted from the 'E' list by a process. Information going from one process to the next must go to the 'E' list. At the end of an event the 'E' list is written to output, then the 'E' list is dropped.

3. **'R' list**

   means Reading list This list contains the names of all banks read in a logical record. The 'R' list is reset to '0' before reading. After reading, the content of the list is analyzed.

   ```
   IF the EVEH bank is  found THEN
       it is  an event  record and 'E' list = 'R' list
   ELSE it is a Constant  record
       and new names  are added to the 'C' list
       'C' list = old 'C' list + 'R' list
   ENDIF
   ```

4. **'S' list**

   means System list.This list cannot be used by users.

5. **'T' list**

   means Temporary list. Users can use it as they want.

5.3.2 HAC (HAnd Coding)

In order to unify the way in which tables are handled we present here guidelines for the coding using mostly statement functions. Statement functions are expanded by the compiler in-line and can be treated subsequently by the optimizer.

5.3.3 Mapping of tables onto BOS-banks

A 'mini header' at the beginning of every BOS-bank gives the row length and the number of rows of the ADAMO table that is mapped onto the BOS-bank. Relations to rows of tables will be represented by integer numbers giving the row number (id) of the target. The long name of an entity set appearing on data-flow-diagrams will be mapped upon a four-character long BOS-bank name. The correspondence of long and short name should be given already in the data-dictionary [6].

Two types of banks are possible:

1. **Static tables** the id is implicitly given by the position (row number) in the table.

2. **Dynamic tables** the id is explicitly given as an integer number at the beginning of every row.

The latter form is preferable in the case where rows of the table are suppressed frequently and relationships should be easily maintained. In the TAP dynamic tables are mapped onto single BOS-banks. An alternate way is to use the BOS bank numbers.

5.3.4 Name Index

Maintaining the 'name-index' : Each named bank has a name-index set by BOS the first time a name is encountered. This name-index does not change during the processing even after garbage collection. The user has access to the name-index through the function `NAMIND`.

```
NAXYZT = NAMIND ('XYZT')
```

`NAXYZT` is the name-index of the bank `XYZT`. The index of the bank `XYZT` is stored in `JQ(NAXYZT)`.

```
KXYZT = JQ(NAXYZT)
```

`JQ(NAXYZT)` is updated after garbage collection. The following scheme for named banks is adopted:

Each process sets a common block called `/xxNAMC/` (xx being the two letter code of the process) with the name-indices of all the banks created and filled by the process. This common block will be filled at initialization time. The format of the banks will be defined at the same time.

The name of the name-indices is of the form : `NAxyzt`, `xyzt` being the bank name starting with the 1st letter code of the process.

```
   i.e. - ITC processor
          SUBROUTINE ITIRUN
 C! initialization of ITC module
 *CA JQCOM
 *CA ITNAMC
 C
 C - Set name-indices
       NAITHT = NAMIND ('ITHT')
       NAITDI = NAMIND ('ITDI')
       NAITDT = NAMIND ('ITDT')
       NAITTD = NAMIND ('ITTD')
 C
 C - Define bank formats
       CALL BKFMT ('ITHT','2I,(3I,6F)')
       CALL BKFMT ('ITDI','B16')
       CALL BKFMT ('ITDT','I')
       CALL BKFMT ('ITTD','I')
C
       RETURN
       END
```

Then the access to a name bank will be done through the name-index for the 1st bank :

```
*CA xxNAMC
*CA JQCOM
*CA BMACRO
         KXYZT = JQ(NAXYZT)
C - if necessary the index of the next bank is :
         KXYZ2 = JQ(KXYZT-1)
```

5.3.5 Creating a named-bank

BOS banks should not be created directly through calls to BOS routines. Instead, the routine AUBOS should be used. This checks whether there is enough space. If not, a garbage collection is done. If this fails as well, the processing of the current event is terminated. Assume a process called PROCES which creates and fills the bank PBNK. The process is entered only once per event.

To CREATE the bank PBNK the following has to be done:

```
1st check the existence of the bank,
IF yes, RENAME it, or DROP it.
THEN create the bank.
THEN put the bank on a list :   'E' event list put on output file
        'T' temporary list drop after the processor
```

Example

```
       SUBROUTINE PROCES
C! PROCES module
*CA JOBCOM  contains flags and counters
*CA JQCOM   BOS array
*CA PRNAMC  name-index for PROCES
*CA BMACRO  statment functions
C
C - if PBNK exists rename it as OPBK or drop it depending on FKEPJO
       KPBNK = JQ (NAPBNK)
       IF ( KPBNK .NE. 0 ) THEN
          IF (FKEPJO) THEN
             CALL BSWAP ( JQ,'PBNK','OPBK')
          ELSE
             CALL BDROP ( JQ,'PBNK' )
          ENDIF
       ENDIF
 C
 C - THEN create the bank 'PBNK', NR=0, with NROW of NCOL words
       LEN = NROW * NCOL + LHEVT
       CALL AUBOS ('PBNK',0,LEN,KPBNK,IGARB,*998)
 C
```

```
C - THEN put the bank on the 'E' list
      CALL BLIST (JQ,'E+','PBNK')
C
      RETURN
      END
```

5.3.6 Statement functions to access tables

A number of statement functions have been defined for the Monte-Carlo program. They are stored in the COMDECK BMACRO. Statement functions are expanded by the compiler into the code and thus do not create major overhead in execution time.

```
*CD BMACRO        set of intrinsic functions to handle BOS banks
C - # of cols/row in bank with index ID
      LCOLS(ID) = JQ(ID+1)
C - # of rows in bank with index ID
      LROWS(ID) = JQ(ID+2)
C - index of next row in the bank with index ID
      KNEXT(ID) = ID + LHEVT + JQ(ID+1)*JQ(ID+2)
C - index of row # NRBOS in the bank with index ID
      KROW(ID,NRBOS) = ID + LHEVT + JQ(ID+1)*(NRBOS-1)
C - # of free words in the bank with index ID
      LFRWRD(ID) = ID + JQ(ID) - KNEXT(ID)
C - Lth integer element of the NRBOSth row of the bank with index ID
      ITABL(ID,NRBOS,L) = JQ(ID+LHEVT+(NRBOS-1)*JQ(ID+1)+L)
C - Lth real element of the NRBOSth row of the bank with index ID
      RTABL(ID,NRBOS,L) = QJ(ID+LHEVT+(NRBOS-1)*JQ(ID+1)+L)
```

Examples

```
        SUBROUTINE NEWHIT(HIT)
        DIMENSION HIT(*)
*CA JOBCOM,JQCOM,XXNAMC,BMACRO
C--------------------------------------------
C       get index of bank 'XXHT'
C
        JXXHT = JQ(NAXXHT)
C
C       if bank does not exist -- return
C
        IF (JXXHT.EQ.0) RETURN
C
C       get # of words/hit
C
        NWRD = LCOLS(JXXHT)
C
C       check if there is enough space to fill a row
```

```
C
      IF (LFRWRD(JXXHT).LT.NWRD) THEN
C
C       increase bank length by NHIT
C
        ND = JQ (JXXHT) + NHIT * NWRD
        CALL AUBOS ('XXHT',0,ND,JXXHT,IGARB,*998)
      ENDIF
C
C     get the index for the next row
C
      KXXHT = KNEXT(JXXHT)
      CALL UCOPY(HIT,QJ(KXXHT+1),NWRD)
C
C     increment the # of filled rows
C
      JQ(JXXHT+2) = JQ(JXXHT+2) + 1
      GOTO 999
 998  LERRBO = .TRUE.
 999  RETURN
      END
      FUNCTION INLOC ( KIND,IREL,ICOL)
C
C     locate the row # which contains the value IVAL in column # ICOL
C     of the bank known by its index KIND
C
C     if ICOL<1 or > # of columns of the table then return INLOC = -1
C     if the value IVAL is not found then return INLOC = 0
C     if the index KIND = 0 then return INLOC = -2
C
*CA JQCOM,BMACRO
C
      IF (IND.EQ.0) THEN
        INLOC = -2
         GOTO 999
      ELSE
        IF (ICOL.LT.1.OR.ICOL.GT.LCOLS(KIND) )THEN
          INLOC=-1
          GOTO 999
        ELSE
          DO 100 I = 1,LROWS(KIND)
            IF (IVAL.EQ.ITABL(KIND,I,ICOL)) THEN
              INLOC = I
              GOTO 999
            ENDIF
 100      CONTINUE
        INLOC = 0
```

```
999     RETURN
        END
```

5.3.7 Inverting Relationships

Assume a static table known by its bank index KBNK2 which contains in column ICOLR the relation to table 1.

The routine UINDXS will build 2 banks:

```
IWREL(1) is a table with 3 columns. It has as many rows as table 1.
IWREL(1)[col1]          will contain a list sorted in increasing order of
                        relations with bank1
IWREL(1)[col2]          will contain the # of elements of table2 associated
                        with the element of bank1
IWREL(1)[col3]          pointer into the bank IWREL(2) to the first element
                        of table 2 associated to the current row of table 1.
IWREL(2)                is a list of pointers to rows of table 2.
```

5.3.8 Work-banks

Work-banks are used within the BOS system [2] for the purpose of storing banks, that are used by the BOS system itself (banks to hold the FORMAT information, banks of indices of named banks produced by call to BWIND,...). In the ALEPH off-line software work-banks can be used to temporarily store information within a module. If improperly used, work-banks can be a source of serious problems, that are very difficult to trace. We therefore require, that the following rules be followed:

RULES

1. A work bank index has to be in a common block. (otherwise the variable may be put into a register by the compiler and in a garbage collection something else than the pointer will be updated)

2. A work bank index cannot be a subroutine argument (similar reason as 1.).

3. A work-bank shall be dropped at the end of a module.

4. A work bank index shall be set to 0 (zero) once before the first creation of the bank (It should not be assumed, that variables are initially set to 0 on every computer).

5. There should be no path in the program, whereby a work-bank index can be modified (e.g. set to zero) for an existing work-bank. (If this rule is violated, there will be two banks, the real one and one 'dead' bank in the BOS structure. During a garbage collection the pointer will be updated twice, first to point to the good bank and immediately later to point to the 'dead' bank which will contain data from a previous pass through the program-module - sometimes several events back. This will at least give bad results).

6. If a bank is not only used locally (i.e. in more than one FUNCTION or SUBROUTINE), it should be given a name in the following way: IW(KWINDX-3)=INTCHA('WSRT') , where KWINDX is the bank index and WSRT the name. This will facilitate debugging in the work-bank area.

A typical way of using a work-bank is as follows:

```
     SUBROUTINE EXAMPLE(IERR)
     ...
     COMMON/EWINDX/KWINDX              | Rule 1
     LOGICAL FIRST                     | Note 1
     DATA FIRST/.TRUE./                |
     ...
     IF(FIRST) THEN                    | Note 2
        KWINDX=0
        FIRST=.FALSE.
     ENDIF
     ...
     CALL WBANK(IW,KWINDX,LEN,*995)    | Note 3
     ...
995  WRITE ....
     IERR=3
     GOTO 999                          | Note 4
996  IERR=4
     GOTO 999                          | Note 4
     ...
999  CALL WDROP(IW,KWINDX)
     RETURN
     END
```

NOTES:

1. Please note that "LOGICAL FIRST/.TRUE./" is not allowed by FORTRAN77.
2. Make sure, that this construct does not appear in more than one routine for the same index. Otherwise rule 5 may be violated.
3. This structure and the rules apply also, if you call instead of WBANK the routines BKTOW,BWIND,UINDXS which also create work-banks with a user-given index.
4. Every error exit of a subroutine should go through the part of the module where work-banks are dropped.

5.4 Bank documentation

All banks used in aleph offline software are documented in the form of DDL. The files are kept in the directory BANKDOC on VXCRNA. Printed information is available in the ALEPH off-line documentation office

The bank structure and data description can be interrogated interactively by the command SBANK.

Banks on data files and on the database can be inspected by the command LOOK.

5.5 Program documentation

In every subroutine there must be a one line comment giving a short description This line has to start with 'C!'. In the subroutine header the author name and the date of writing and of modifications shall be given. The author name shall be given in full and not only the initials. All arguments used in the calling sequence shall be described. A longer comment (minispec) explaining the function of the subroutine shall be given. Minispecs are indicated by 'C?' at the start of line. Routines called and calling or BOS-banks booked need not be given in the SUBROUTINE header. This information will be extracted automatically from the code.

Example

```
*DK EXAMPLE
      SUBROUTINE EXAMPL(INVAR,RIN,RARR,OUT,FLG)
C
C-----------------------------------------------------------------------
C!   - This is an example for a subroutine header
C!
C!    Author      :- J. Knobloch 86/9/3
C!    Modified    :- J. Knobloch 86/9/12
C!
C!   Input:
C!        - INVAR /I     : The first one
C!        - RIN   /R     : This is a real input variable
C!        - RARR  /R     : Some input array dimensioned by calling routine
C!
C!   Output:
C!        - OUT   /R     : Even a dummy routine creates output
C!        - IFLG  /I     : Flag to indicate errors
C!                    = 0   if all O.K.
C!                    = 1   if INVAR out of range
C!
C!
C!   Libraries required: none
C!
C!    Description
C!    ===========
C!    This is a routine serving as an example how to write a
C!    subroutine header. The type indications for the arguments are
C!    not necessary if standard is used.
C?
```

```
C?     read a hypothetical bank from the raw data tape
C?     if the bank HYPO does not exist
C?               print a warning message
C?               set an error flag
C?               return from subroutine
C?     else
C?               create another bank XBNK
C?               apply global calibration constant
C?               transform ....
C?               .....
C!====================================================================
*IF .NOT.DOC
```

5.6 COMMON documentation

In every Historian COMDECK there shall be only one COMMON statement. The COMMON name and the COMDECK name shall be the same. An exception is allowed if variables of different type (character and other) belong logically together. Dimensions of common-variables shall be given as parameters. If these are not used elsewhere, the parameter statement may be put into the same COMDECK. COMMON variables shall be dimensioned in the COMMON declaration and not in an extra DIMENSION or type declaration statement.

A general comment for the comdeck shall be given in the first three lines of the documentation part of the COMDECK which starts with '*IF DOC'. The documentation lines start with 'C', 'C!' or just two blanks. Every COMMON-variable or PARAMETER is described in at least one line giving the variable name, a separator (which can be '=',':' or blank) followed by the comment. For the automatic documentation tools only the first line of comment is used for every variable. Thus this line should be self contained and not refer to a previous line:

```
Counter example:    XCOORD = X value of some detector
===============     YCOORD = Y   "    "  "      "
```

Example

```
*CD ITEXAM
      PARAMETER (JLENGT=8)
      COMMON/ITEXAM/NHITIT,IADDIT(JLENGT),IAMBIT(JLENGT),ILAYIT(JLENGT),
     +              PHITIT(JLENGT),SRHOIT,PHIOIT,DROOIT,CHIQIT
*IF DOC
C
C! This is a global comment of an example COMDECK
C
C
C JLENGT  : Number of layers in ITC
C NHITIT  : Number of hits on track
C IADDITi : i=1,NHITIT: ITC coordinate no. in ITCO bank
```

```
C IAMBITi : i=1,NHITIT: ambiguity used,  = 1 if first;= 2 if second
C ILAYITi : i=1,NHITIT: Layer no. of hit i
C PHITITi : i=1,NHITIT: phi of hit i
C SRHOIT  : sig*r0 of fit
C PHIOIT  : phi0 of fit
C DROOIT  : d0   of fit
C CHIQIT  : chisq of fit
```

5.7 HISTORIAN usage

Every routine shall be put in a deck having the same name as the routine. All DECKs shall be put in alphabetical order on the HISTORIAN library (`HLB` on VAX). Correction sets shall be created using HISTG (after a deck was extracted with HISTE). In any case shall the correction contain the DECK-name it refers to:

Correct:

```
*D RJ021287.388,RJ021287.389        TKSGDP
       KTGTL=IW(NAMIND('TGTL'))
```

Wrong:

```
*I,RJ220888.10
     &,R5ESTI(IMAX),R6ESTI(IMAX),R7ESTI(IMAX)
```

References

[1] JULIA Users and Programmers Guide. (Version 2.41) by J. Knobloch (ALEPH Note 90-115, SOFTWR 90-011)

[2] "The BOS System" by Volker Blobel. DESY Internal Report DESY-F14-86-02

[3] KINGAL User's Guide by B. Bloch-Devaux ALEPH 91-82.

[4] ALEPH Note 92-008.

[5] ALEPH Note 86-94.

[6] ALEPH Note 142, rev. 1.

[7] GALEPH User Guide by F. Ranjard, ALEPH Note 119.

[8] ALPHA User Guide (Version 117/118) by J. Boucrot et al. ALEPH Note 94-092.