SLAC-PUB 523 1
April 1990
(E/I)

JAZELLE

An Enhanced Data Management System for High Energy Physics†

A.S. Johnson

Boston University, Dept. of Physics, 590 Commonwealth Ave., Boston, MA 02215

M.I. Breidenbach, H. Hissen, P.F. Kunz and D.J. Sherden

Stanford Linear Accelerator Center, Stanford University, Stanford, CA 94309

T. Burnett

University of Washington, Dept. of Physics, FM 15, Seattle, WA 98195

Abstract

The- data management system JAZELLE has been created as a successor to earlier HEP data managers such as YBOS and ZEBRA. While it has many similarities with these systems it also has many enhancements such as self-documenting data descriptions, mnemonic access to all data, relational data structures, powerful machine independent IO facilities including network IO, and many mechanisms for presenting data to the physicist in an intuitive manner. The emphasis has been on producing a powerful, user friendly, data management system which can be accessed from many languages as a natural extension of those languages.

Within the SLD collaboration JAZELLE has been used to manage the experimental data all the way from the SLD online VAX to the end of the reconstruction chain and physics analysis. Beyond this, JAZELLE is also used to store calibration constants and correction factors, detector geometry, physics parameters (such as particles masses and branching ratios) and program control parameters. JAZELLE has also been interfaced to the interactive data analysis program IDA. The uniform usage of JAZELLE throughout SLD coupled with its interface to IDA has produced an environment in which rapid interactive access to data is greatly simplified.

JAZELLE and IDA together provide an analysis environment greatly superior to any standard programming language.

Presented at the 8th Computing in High Energy Physics conference,
Santa Fe, New Mexico, April 9-13, 1990.

† This work was supported in part by the Department of Energy contracts DE-AC02-89ER40509 and DE-AC03-76SF00515

Introduction

From a software perspective the most important aspect of any high-energy physics experiment is the data. The data include both raw and reconstructed event data, calibration data such as detector gains, drift velocities etc., and data which describe the experiment, such as detector geometry descriptions. During analysis and reconstruction the data must typically be processed through several programs, often moved between several different types of computers, and at all stages the experimenters must have easy and efficient access to the data. The various different types of data may vary dramatically in size, from several 100 kbtyes of raw data per event, to several bytes per event in the final stages of analysis. Therefore an essential part of any high energy physics experiment's software should be a data management system capable of handling the varied needs outlined above.

When beginning the SLD experiment we looked for a system which would be able to handle all of these requirements. We looked at many different languages, but while many existing languages have very powerful mechanisms for defining and manipulating data structures within a program they do not in general have good facilities for moving data between programs or for providing easy interactive access to the data. Similarly there are many commercial database type programs available, but in general they are not well suited to handling the range of different data types and sizes needed and do not provide sufficiently efficient and easy to use program interfaces.

Finally we looked at other-data management systems developed specifically for HEP applications, specifically ZBOOK, ZEBRA[1] and YBOS. While these systems are well suited to handling the data processing needs of HEP experiments they generally fall short in the area of user friendliness, forcing users to memorize numeric offsets within data structures thus making programs rather hard to read and understand.

For these reasons it was decided to develop an entirely new data management system for the SLD experiment, called JAZELLE. JAZELLE has been in use by SLD now for over three years and currently runs on both VM/XA and VMS operating systems. In addition work is underway to transfer it to several UNIX based workstations.

JAZELLE has many similarities with the earlier HEP data managers described above and has adopted many of the best features of these systems. Data are stored in contiguous structures called ***bunks.*** Banks may be dynamically created, modified and destroyed during program execution, and may be read in or written out from programs. Some of the more novel features of JAZELLE include:

- The structure of each JAZELLE bank is defined in a ***template*** file. In the template file each element is typed (REAL, INTEGER etc.) and named. The template file allows a description to be attached to each element and therefore fulfills an important documentation function as well. Elements of different types can be freely mixed within banks.
- JAZELLE elements are always referenced by name. A preprocessor is used which automatically calculates the offset of each element within a bank (using the information in the template file) and generates a suitable target-language expression corresponding to each element reference.
- JAZELLE contains utilities for presenting data to the user in many different formats. The name of each element and the description from the template can optionally be included along with the data itself.

- JAZELLE is fully integrated into SLAC's interactive data analysis program, IDA. A complete set of interactive commands are available for manipulating and viewing JAZELLE structures, and data stored in JAZELLE banks can be freely accessed from IDA's interactive analysis language, IDAL.
- Debugging facilities are provided by making the entire set of interactive commands also available from within the VMS debugger.
- JAZELLE contains facilities for describing data structures using a relational technique, in addition to the more common hierarchical data descriptions.
- JAZELLE obviates the need for the large pre-assigned common block found in other systems by using the virtual memory services of the host operating system. Data can be partitioned into different virtual memory zones (called ***contexts*** in JAZELLE parlance).
- JAZELLE includes IO facilities to handle input and output of data to both sequential and indexed files. Data written on either VM or VMS can be read back on either system.
- The VAX version of JAZELLE includes features of use with online systems such as asynchronous inter-process IO and data sharing via global sections.

Many of these points will be covered in more detail in the following sections.

Defining data structures - TEMPLATES

The basic element from which JAZELLE data structures are built is called a **bank.** A bank is a contiguous piece of memory consisting of two parts, a **16-byte** header section containing JAZELLE system data, and a user area where the data associated with the bank are stored. The structure of the user area is defined in a user supplied ***template*** file. The syntax of the template file allows each element within the bank to be named and typed, and also allows for initial values and descriptions to be attached to each element. Well written templates contain all the information needed to document the data structure. Figure 1 shows an example of a template **file,** and Table 1 lists all of the variable types allowed within a template.

Table 1: Data types supported in JAZELLE templates

Types[1]
INTEGER'4 REAL*4 HEX*4 LOGICAL"1 POINTER KEY STRING"4 STRING20
INTEGERS REAL*8 HEX*2 LOGICAL*8 COMPLEX TIME STRING*8 PARTID

[1] Jazelle also supports user defined data types

Jazelle banks are grouped into ***families,*** each of which has a unique ***family name.*** All banks in a family share a single template, and hence a single structure. To distinguish banks within a family each bank is assigned a unique ID in the range O-65535.

Banks **may** contain arbitrary combinations of data types in any order, including both scaler and vector elements. Banks can also contain constants (declared using a ***parameter*** statement). In addition elements within a bank may be grouped ***into blocks.*** Blocks themselves may be dimensioned to create an arbitrary number of repetitions of the elements within the block. Blocks may be nested up to ten deep.

Figure 1: Example of a template file

```
!-----------------------------------------------------------------------
!
! Template NONSENSE:  This is an example template to illustrate features
! of the template syntax.
!
! Note that any text following an ! is treated as a comment
!
!-----------------------------------------------------------------------

BANK  NONSENSE  CONTEXT=JUNK  NOMAXID "This is the title of bank nonsense"

     INTEGER  A          "Dndimensioned  uninitialized  variable        "
   REAL       C          "Another  variable                              "
     INTEGER   NELEMS     "Variable  used  later  as  a  variable  dimension  "
     REAL*8    D(10)      "Dimensioned  array                            "
     LOGICAL"1 X(0:7)     "First/last  dimensioning                       "
     INTEGER   Y/0/       "Initial  value  assigned                       "

     PARAMETER  SIZE=8     ! Definition  of  a  constant  parameter       "
     INTEGER   N(SIZE)/SIZE*7/ "Example  of  use  of  parameter           "

     BLOCK     VBLK(NELEMS)    "Local  block  (with  variable  dimension)  "

       STRING  F(80)      "80  character  string                          "
       INTEGER I
       BLOCK  INNER(2)    "Nested  block                                  "
         REAL  x
         REAL  Y
       ENDBLOCK

     ENDBLOCK

ENDBANK
```

In addition to the fixed dimensions already described, the last element or block in each bank may be given a variable dimension. Figure 1 contains an example of a variably dimensioned block, VBLK. Variable dimensions are created by specifying the dimension of a block or element as an integer element declared previously in the bank (NELEMS in the example). The amount of memory allocated for variably dimensioned elements can be increased or decreased dynamically

Manipulating Banks and Accessing Data

Jazelle provides many routines to allow banks to be created, deleted, copied, expanded or contracted etc. Some of the most commonly used routines are summarized in Table 2.

Table 2: Frequently used JAZELLE routines

Routine	Usage
JZBADD	Create a bank
JZBDEL	Delete a bank
JZBEXP	Expand or contract a bank
JZBFND	Obtain a pointer to an existing bank
JZBLOC	Obtain a pointer to a family of banks
JZBCPY	Create a copy of a bank
JZBDMP	Dump a bank or family of banks
JZBTBL	Tabulate a bank or family of banks
JZTDEF	Add a column to a table or create a new table
JZIOPN	Open a file for JAZELLE IO
JZIOCL	Close a file
JZIOWR	Write a record using a list of banks
JZIOWC	Write a record using a context
JZIORD	Read a record
JZINDX	Produce an index of all existing banks
JZSTAT	Produce a summary of JAZELLE virtual memory usage
JZTSCN	Scan a relational table
JZXWIP	Delete an entire context

One of the main aims of designing JAZELLE was to make it fit elegantly into the programming language(s) used to write code for the experiment, in effect to give the user the impression that access to JAZELLE data is a natural extension of the language. SLD chose to use an existing preprocessor, MORTRAN, and to extend it to provide access to JAZELLE[2] . However the same features can be made available in almost any language, and JAZELLE has been interfaced to Fortran-77 (using the JFORT[3] preprocessor), and C[4] , in addition to MORTRAN. In the case of C no pre-processor is required since the JAZELLE structures can be mapped directly onto C-structs and the native C syntax used to access data from JAZELLE banks. In this paper we choose to illustrate the use of JAZELLE in Fortran.

Since JAZELLE banks are dynamic in nature, in order to access data from a bank it is first necessary to obtain a pointer to the bank. The JFORT preprocessor extends native Fortran-77 by introducing a new variable-type, ***POINTER,*** and allowing variables of this type to be declared using the POINTER declaration in a manner analogous to any other Fortran declaration. The pointer statement also requires the pointer to be cast to a particular family: e.g.

```
POINTER   KAON-->MCPART
POINTER  DECAY_PRODUCTS(6)-->MCPART
POINTER  BPTR-->NONSENSE
```

Variables declared as pointers may be used anywhere a normal variable could be used, for example as a local variable, as an element of a common block, or as an argument of a function or subroutine. Arrays of pointers can also be declared (for example DECAY-PRODUCTS in the above example).

Once a pointer declared in this way has been assigned a value, for example by using it as an argument to one of the many JAZELLE routines which return pointers such as JZBADD, then it can be used to access elements of a bank using an expression of the form:

```
BPTR%(A)
BPTR%(D(5))
BPTR%(X(K-1))
BPTR%(VBLK(3),I)
BPTR%(VBLK(3),INNER(1),X)
```

The percent sign (%) is used to indicate pointer dereferencing, and to distinguish JAZELLE data references from normal Fortran array references. Of course any Fortran expression can be used as the index of a pointer array, or as the index of any JAZELLE vector or block index. Expressions of this form can be used anywhere that a common block element can be used in standard Fortran, for example on either side of an assignment statement, as an argument to a function or subroutine etc. Elements in the bank header and parameters declared in banks can also be accessed using a similar syntax.

Note that the preprocessor converts all references to JAZELLE data elements into references to elements of a dummy common block. All offset are calculated inline so that no function or subroutine calls are necessary to access JAZELLE data. Care has been taken to generate expressions which do not inhibit the normal ability of code optimizers to optimize code processed through the JAZELLE preprocessor. Real applications making extensive use of JAZELLE show that the typical CPU time overhead incurred by using JAZELLE as opposed to common blocks is only about 5%.

One disadvantage of using a preprocessor is that when using symbolic debuggers, the code seen by the user is not the original code written. To lessen the impact of this the JAZELLE preprocessor always generates comments in the output code showing where and how it has made code substitutions. In addition, in the case of the VMS debugger, it has been possible to extend the debugger to accept interactive commands to enable the direct examination and modification of data in JAZELLE banks.

Using JAZELLE for Interactive Data Analysis

One program that has influenced the way data analysis is performed at SLAC more than any other is IDA[5] , originally written for the MARKIII collaboration. IDA, which stands for Interactive Data Analysis, is a program which allows physicists to very rapidly experiment with cuts and histograms by providing them with a powerful, semi-interpretive, programming language tailored specifically to the requirements of data analysis. In its MARKIII incantation IDA required a special data file to be created containing data to be analysed. This file had to be created by each user to contain the data that he was interested in, and each time some previously unneeded data was required, or the sample of events of interest changed this file had to be rewritten.

SLD's use of JAZELLE and the complete integration of JAZELLE and IDA has removed the need for this special data file. IDA now provides complete interactive access to all of the functionality of JAZELLE. Inside IDA a user has the ability to look at any data contained in JAZELLE banks; create and display tables of any quantities; create, modify or delete banks; or read in and write out data at will. Since all of the calibration and geometry constants are also stored in JAZELLE banks users can also examine and/or change these interactively.

Within IDA's programming language, IDAL, users can also access data from, or store data into, JAZELLE banks using a syntax very similar to that used in the FORTRAN examples above. Thus any data in JAZELLE banks can be immediately used

in calculations or to form the basis of cuts, and can be histogrammed. New banks can also be created, and new data calculated and stored in them. Special constructs have been incorporated into **IDAL** to allow easier access to JAZELLE data, such as the **BANKLOOP** construct for looping over banks in a family, and the TABLELOOP construct for accessing data from relational tables (described in the next section). Whilst the emphasis in designing the **Fortran** interface to JAZELLE was to provide a robust programming language, requiring for example the declaration and strong typing of all pointers, in IDA the emphasis has been placed on rapid and informal program development, thus pointers need not be declared gaming their casting automatically from the context in which they are used.

Since JAZELLE and IDA together provide such a powerful tool, SLD has adopted IDA as the shell for all of their offline jobs, from initial filtering of the data, through reconstruction, to final DST analysis. In this way the power of IDA to examine and histogram data is available in a uniform way at all stages of analysis.

All of the main sections of SLD Monte-Carlo and reconstruction code are written as ***processors*** callable from **IDAL.** Thus the main event loop is always written as an **IDAL** routine (see for example Figure 2). All of the control parameters for each of these processors are stored in JAZELLE banks and can therefore be viewed and **modified** from IDA. Although MC and reconstruction jobs are normally far too slow to be termed "interactive", using a uniform framework for all programs is still very worthwhile.

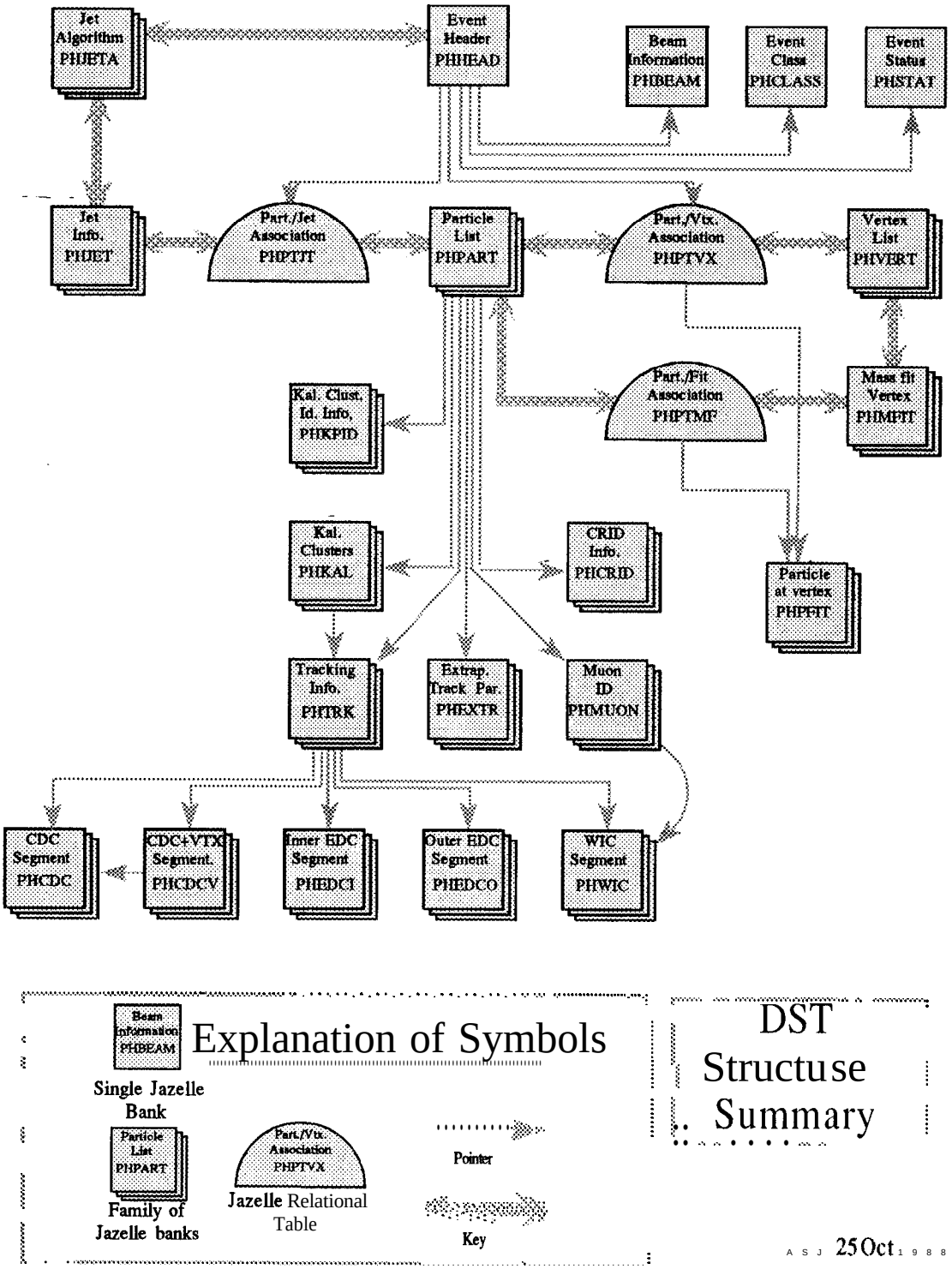
Figure 2: Example of an Event Loop Written using IDAL

```
Def  Evanal    ! Event loop

   Call  LUND    ! Generate a LUND event
   Call  GSIM    ! Simulate the experiment using GEANT
   Call  RECSLD  ! REconstruct the entire event

   Hist  BankCnt(_Phtrk) From 0 to 50 Title "Number of charged tracks found"

EndDef        ! End event loop

Poke  Mclundp.sin2thw=.220  ! Set a control parameter for the LUND generator
Peek   Mclundp              ! Examine all of the parameters of the LUND generator
Go  100                     ! Run the event loop for 100 events
```

The combination of IDA and JAZELLE provides the user with a extremely powerful tool for performing physics analysis. It is possible to histogram events and decide on a set of cuts to select an interesting subset of events; create a new bank or set of banks representing a nano-DST containing those quantities of interest for a particular analysis; write an **IDAL** program to apply the cuts and then create and fill the nano-dst banks and write them out; and then continue further analysis of the nano-dst without once having to leave the IDA program. Within the IDA environment the computer really becomes a toll for data analysis.

Within SLD the power of IDA as a data analysis system has been further enhanced by integrating the SLD event display into IDA. Work is also underway to provide a complete graphical X-window interface to IDA and JAZELLE to further increase the user-friendliness of the program.

Figure 3: Example of a data-structure built using JAZELLE



Using JAZELLE to construct complex data structures

So far we have discussed how JAZELLE allows data to be represented in banks. In practice there are many reasons for individual banks to be kept small and to compose complex data structures by linking together many small banks. JAZELLE provides mechanisms for linking banks together to make arbitrarily complex data structures.

The simplest of these mechanisms is to imbed pointers to banks inside other banks. A typical use of pointers is to create hierarchies of banks with a top level bank pointing to other banks which contain more detailed information. For example in the SLD DST data structure (Figure 3) a particle summary bank (PHPART) contains pointers to lower level banks-which contain detailed information from each of the detector elements in which the particle was reconstructed. Some of these in turn point to still lower level banks containing even greater detail. Arranging the data in this way, rather than in one huge bank, allows efficient representation of the fact that not all particles are seen in all detector elements; unnecessary banks are simply not created and the pointer which would otherwise point to them is assigned a null value. In addition such a structure allows the user to drop some or all of the more detailed information if it is not required for a particular analysis.

A second, and more innovative, way of representing relationships between JAZELLE- banks is by the use of relational tables. A relational table is a family of JAZELLE banks whose structure contains one or more elements of type KEY. Within a relational table each bank represents a relationship, with the KEY(s) pointing to the bank(s) being related, and with the bank itself able to store additional information concerning each relationship. For example in the SLD DST structure it was desired to represent the relationship between vertices and particles. Given that vertices were not reconstructed unambiguously each vertex could be associated with any number of particles, and each particle could be attached to an arbitrary number of vertices. Thus it was required to represent a completely arbitrary NxM relationship. The bank used to create the relational table to represent the particle-vertex relationship is called PHPTVX, shown in Figure 4. The example illustrates how properties of the relationship between a measured particle trajectory and a vertex, such as the distance of closest approach, can be stored in the bank along with the keys which point to the banks being related.

Keys are implemented in such a way as to make scanning a relational table very efficient. Thus tools are available to answer the question, "Which particles are attached to this vertex?" as well as the symmetric question, "Which vertices is this particle connected to?" very efficiently.

Conclusion

It has often been assumed in the past that memory management tools are only necessary in HEP to compensate for the lack of adequate language constructs in the de-facto HEP standard language Fortran. During the development of JAZELLE it has become apparent that it is possible to develop tools which have considerable advantages over those provided by any programming language alone. Examples of these advantages are the ability to transport complex data-structures easily and flexibly between programs and between different machines, tools to allow easy examination of the data, and perhaps most importantly the ability to access and manipulate the data interactively from programs such as IDA. Although JAZELLE was developed originally for use with Fortran, almost all of the code written for JAZELLE would be needed for a similar system

Figure 4: Example of a template for a relational table

```
! Each PHPTVX bank represents one particle attached to one geometric
! vertex. The bank summarizes the quality of the fit and contains a
! pointer to a PHPFIT bank which contains particle parameters either
! calculated at or constrained to the vertex.

Hank PHPTVX Context=DST NoMaxid "Particle-Geometric Vertex table"

  Real    DOCA       "Distance of closest approach of unfit track to vertex"
  Real    DDOCA      "Error on distance of closest approach"

  Real    CH12       "Chi2 contribution to fit from this particle"
  Real    NDF        "Number of degrees of freedom for this particle"

  Key     PHPART-->PHPART  "Pointer to associated particle"
  Key     PHVERT-->PHVERT  "Pointer to associated vertex"
  Pointer PHPFIT-->PHPFIT "Pointer to particle parameters at vertex"

Endbank
```

intended for use with a language already supporting more advanced language constructs such as ADA or C. In this paper if has only been possible to outline very briefly some of the most important features of JAZELLE. For a complete description see the JAZELLE Users Guide[6] .

Acknowledgments

We would like to acknowledge the members of the SLD collaboration who have helped with many original ideas during the development of JAZELLE. We would also like to acknowledge the diligent work performed by visiting students Brian Anderson, Will Ballantyne and Sean Sterner.

References

1. R.Brun, M.Goosens, J.Zoll, CERN DD/EE/85-6; V.Blobel, DESY-Rl-88-01

2. A.S. Johnson, Comput. Phys. Commun. 45 (1987) 275-281.

3. The JFORT preprocessor was inspired by the PREPFORT preprocessor developed by D.Aston, M.Gravina and P.Kunz.

4. The experimental interface of JAZELLE to the C language was performed by G.Word.

5. T.Burnett, Comput. Phys. Commun. 45 (1987) 195-199.

6. A.S.Johnson and D.J.Sherden, SLAC-PUB-263.