SLAC–362
UC–405
(E/I)

JAZELLE Users Manual

A. S. Johnson

Stanford Linear Accelerator Center
Stanford University
Stanford, California 94309

April 1990

Prepared for the Department of Energy
under contract number DE–AC03–76SF00515

JAZELLE Users Manual

The data management system JAZELLE was developed explicitly for the SLD experiment. While it has many similarities with earlier high-energy physics data management systems such as YBOS and ZBOOK, it has many other features not found in these systems. The basic data structure in JAZELLE is called a 'bank,' and families of JAZELLE banks are used to store and manage the experimental data all the way from the SLD online Vax to the end of the reconstruction chain and physics analysis. JAZELLE replaces both the raw data and the track-list/vertex-list used in previous SLAC experiments such as Mark II and Mark III. Beyond this, however, 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.

The structure of each JAZELLE bank is defined in a "template" file where 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. All data in JAZELLE banks is referenced by name — an extremely convenient feature. JAZELLE banks can be created and destroyed dynamically as required by a specific analysis, and data in them can be examined or modified. JAZELLE also includes a system of relational tables that connect related families of banks and input-output facilities to handle input and output of data. Data written on either VM or VMS can be read back on either system.

Revision/Update Information: Version 3.0 (SLD/MORTRAN VERSION)

Contents

Preface

Preface

Acknowledgments

The JAZELLE data management system has been written for the SLD collaboration by Tony Johnson and Dave Sherden with much help from visiting students Will Ballentyne, Helmut Hissen and Sean Sterner. In addition, much constructive input has come from many collaborators including; D.Aston, G.Baranko, C.Boeheim, M.Briedenbach, T.Burnett, R.Dubois, M.Gravina, P.Kunz, L.Rochester, and O.Saxton.

Intended Audience

This manual is intended to be read by all Jazelle users. Some small knowledge of SLD MORTRAN and the SLD ERROR system is assumed, reading the appropriate color books before this manual should prove sufficient.

Users encountering JAZELLE for the first time should find it sufficient to read the first five chapters which introduce all the concepts that are required for successful use of JAZELLE. Each chapter starts with a simple introduction and then progresses into more and more detail, so the later sections of each chapter should probably be omitted on a first reading.

For users who are already familiar with previous versions of JAZELLE, a list of changes between versions is included in the preface. Browsing through this list should alert users to any new features which they may be interested in. A comprehensive index is provided at the back of the manual to allow individual features of interest to be looked up.

The last chapter of the manual contains a reference section listing all of the user routines and their arguments. This section should be useful as a reference for experienced users.

Finally if you should experience any problems with this manual, or while trying to use JAZELLE please do not hesitate to ask me about your problems. I can be contacted at office G106 in the central lab, extension 2278, or as TONYJ@SLACVM.BITNET.

Update History

Changes Introduced for JAZELLE 3.0

To be supplied.

Changes Introduced for JAZELLE 2.2

Indexed Pointers

Indexed pointers have been added to JAZELLE. These pointers provide an alternate method of accessing banks using an array indexed by bank Id containing pointers to banks within a family. Index pointers are discussed in the chapter on referencing JAZELLE data elements.

A new user routine JZPIDX is provided to initialize indexed pointer arrays.

Templates

A new bank qualifier, MAXID is added. This allows an upper limit for Id's within a family to be specified, used primarily with the indexed pointer scheme.

Relational Tables

A new routine JZTSCN is provided to simplify the task of scanning a relational table to find occurences of a given key value.

Backward Compatibility

JAZELLE release 2.2 is fully backward compatible.

Changes Introduced for JAZELLE 2.1

Input/Output

A new set of IO routines are installed. These routines are more efficient than the old routines.

Input and output of entire contexts is now supported.

Opening files by name on VM and VMS is now supported.

Backward Compatibility

JAZELLE 2,1 is backward compatible except for the following restrictions:

i Datasets written with previous versions of JAZELLE cannot be read with JAZELLE 2.1.

ii The arguments to some Io routines, in particular JZIOPN, have changed.

iii The default action on reading in a bank which already exists has changed.

Changes Introduced for JAZELLE 2.0

Templates

TPLPAR files are now completely obsolete. The template command still exists but is just a tool for checking the syntax of TEMPLATE files.

TEMPLATE files may now contain continuation lines.

The comment delimiter may occur at any position within TEMPLATE file lines.

New derived datatypes have been defined, namely COMPLEX, STRING20 and KEY.

New element qualifiers may be specified in TEMPLATE files, namely TABULATE and HEADING.

POINTERS and KEYS in banks can now be bound to families. I.e. the syntax

```
POINTER TRACK->MCTRACK
```

is now supported in banks.

Context

The concept of CONTEXT is now supported within JAZELLE. Statistics on memory usage will be available on a per-context basis.

Context wipe (deleting all banks within a context) is now supported.

Pointers

The implementation of pointers has significantly changed. All changes should however be backward compatible with previous releases of JAZELLE.

Pointers are now fully relocatable. I.e. they will continue to point to a bank even after that bank is contracted or expanded.

Pointers in user code can now be registered using JZPREG. Dumps of all registered pointers can now be generated.

Registered user pointers and bound pointers in banks will be zeroed when ever a bank in the same context is deleted.

The concept of a pointer to a family (as returned by JZBLOC) has been formalized.

Debug Aids

An additional routine JZMAP has been provided to produce a map of virtual memory usage.

A routine JZSTAT has been produced to provide statistics on JAZELLE memory usage.

A new routine JZPM has been provided to produce a JAZELLE post mortem. In addition, a post mortem will automatically be produced whenever a fataL JAZELLE or GETVM error occurs and whenever an access violation (VMS) or OC4/5 (VM) occurs.

Dump Routines

A powerful new routine will be provided for producing dumps of banks as tables. The routine can be used to tabulate blocks inside banks or to tabulate entire families of banks (or a combination of the two). The new routine is called JZBTBL.

Elements to be tabulated can be specified in the template or using the JZTDEF routine.

Constants Management

An interim system for managing constants is provided.

Relational Tables

The concept of a relational table (modeled on relational databases) is now supported via the new KEY data type.

When banks are deleted any banks which have KEYS referring to that bank will also be deleted.

Backward Compatibility

JAZELLE 2.0 is backward compatible with previous releases of JAZELLE subject to the following restrictions.

i All code which refers to JAZELLE variables will have to be recompiled with the new release of MORTRAN available with JAZELLE 2.0.

ii No i/o package is available with JAZELLE 2.0. This will be rectified soon with JAZELLE 2.1.

iii The old Q% method of referencing variables is now officially obsolete. Code using the Q% syntax will continue to compile but JAZELLE will issue a warning message. The Q% syntax will be removed completely soon.

iv The following user routines are obsolete having been replaced by more powerful routines: JZDUMP replaced by JZINDX JVPEEK replaced by JVSHOW

1 Introduction

JAZELLE is a data management package, designed to provide facilities for data structure manipulation considerably more powerful than those provided by standard FORTRAN 77. Since JAZELLE is built on top of FORTRAN it cannot hope to provide the level of integration between program design and data structure typical of more modern languages, but by the use of data structure definitions (called "TEMPLATES" in JAZELLE jargon) and the power of MORTRAN macros, JAZELLE attempts to make the use and manipulation of data structures within programs as unobtrusive as possible.

The basic element from which JAZELLE data structures are built is called a bank. Just like the familiar FORTRAN common block, a bank may contain any combination of variable types such as INTEGER, REAL, STRING etc.. Banks may contain both scalar elements and vectors. Unlike common blocks however, JAZELLE banks can be dynamically created and destroyed during program execution and also expanded or contracted as the need arises. In addition to this JAZELLE provides facilities for inputting and outputting banks, and for interactively examining and modifying the contents of banks.

Jazelle banks are grouped into families, each of which has a unique "family name". Each family has a template which defines the structure of the banks belonging to that family. All banks within a family therefore share the same structure. In addition to the family name each bank has an associated ID, which is an integer in the range 0-65535. In order to uniquely identify a bank it is necessary to specify its family name, ID and context. Context will be defined in a later chapter, but for now it is only necessary to know that it is an 8 character name associated with every bank.

When used, banks can be referred to in two ways, either by the bank name and ID described above or by a pointer. A pointer is simply a variable which contains the address of the bank in memory, i.e. a variable that "points" to the bank. Pointers can either be normal FORTRAN variables or they can be elements in banks. In the second form they allow banks to point to other banks and this enables banks to be logically grouped together to build up arbitrarily complex data structures.

The remainder of this manual describes in more detail the various features of JAZELLE. In Chapter 2 templates are discussed in detail. Chapter 3 gives a brief introduction to creating and manipulating banks. In chapter 4 we explain the use of MORTRAN macros for accessing the contents of banks. Chapter 5 deals with producing human readable dumps of banks. The remaining chapters then go on to discuss some of the more advanced features of JAZELLE. Finally the last chapter lists all of the user routines and describes their arguments and functions.

2 Templates

The data structure of a JAZELLE bank of a given family is described by a template of the same name as the family. Template information for a given family is derived from a human readable/editable file of type TEMPLATE. The TEMPLATE file is used by MORTRAN (and IDA) in such a way as to allow named access to any element within the data structure from any MORTRAN (or IDA) program.

Template information is also read into memory during program execution and used to allocate banks and perform VMS/VM IO conversion. Because data elements within templates are named, this information can be further used in dumping, displaying, and accessing the data interactively.

The syntax of the TEMPLATE file allows descriptions to be attached to each of the banks defined, and also to each element within a bank. Thus well written TEMPLATE's will also contain all the information necessary to document the data structure defined.

2.1 Example

Before going on to describe the full syntax of a template file it will probably prove instructive to examine a simple example, and then to examine some of the more specialized features of templates.

The above template defines a family of banks whose family name is NONSENSE. The template then goes on to define the structure that each bank of family nonsense will have. Even without much explanation the structure of the bank should be fairly clear from the example. Note that the syntax and meaning of each variable declaration is very similar to FORTRAN, but a few differences should be pointed out.

- Multiple dimensions on one variable are not allowed, e.g. REAL D(10,10) is illegal. However there is a way around this problem which is described under the heading "blocks within templates" below.
- Each variable and the bank itself has a description associated with it. This description is more than just a comment since it is stored in memory and can be optionally output along with each variable when the data is dumped.
- Rather than the familiar CHARACTER type of FORTRAN, JAZELLE has a type STRING. Note that the statement STRING F(80) defines a string of length 80 characters, i.e. it is similar to the FORTRAN statement CHARACTER*80 F. JAZELLE provides a special set of routines to convert from its internal string representation to FORTRAN's character representation.

- JAZELLE has the concept of a block. In the above example the bank NONSENSE has a block called P, which is repeated 5 times. Each of the five occurrences of the block contains a string F and an integer I. The use of blocks will be described further below.

2.2 Blocks within Templates

The example discussed above already contained a simple example of a block within a template. Blocks are used whenever it is desirable to group a set of variables together logically. They are particularly useful when it is desired to repeat a set of variables a number of times as in the NONSENSE example above. Blocks may occur anywhere within a bank, including within other blocks. Blocks also provide a means of effectively creating multiple dimension arrays. Consider the following bank for instance.

```
BANK CUBE CONTEXT=XYZ NOMAXID

   PARAMETER DIM=3

   BLOCK X(DIM)
     BLOCK Y(DIM)

       PARAMETER VALUE=5
       INTEGER Z(DIM)/DIM*VALUE/

     ENDBLOCK
   ENDBLOCK

ENDBANK
```

The variable Z now has three indices associated with it. The index for block X, the index for block Y and the index of Z itself. It therefore effectively represents a 3x3x3 matrix. Also note the use of the JAZELLE parameter statement in the above example.

All of the blocks discussed so far have been what are called "internal blocks". JAZELLE also supports the concept of an external block. External blocks are defined and used as follows.

```
BLOCK TRACK

   REAL    MOMENTUM(3)
   REAL    MASS
   INTEGER CHARGE

ENDBLOCK

BANK DECAY CONTEXT=VERTEX NOMAXID

   TRACK IN(1)
   TRACK OUT(2)

ENDBANK
```

In this example an external block called TRACK has been defined. Information stored with each track consists of its momentum, mass and charge. The block TRACK is used in bank DECAY. Note that external blocks are referenced as if they were a variable type such as INTEGER or REAL. External blocks therefore allow users to define their own variable types in addition to the standard ones provided by JAZELLE.

2.3 Variably Dimensioned Blocks

So far each bank that we have discussed has been of fixed size. JAZELLE also offers the possibility to have variable dimensions within banks. Only the last element or block of a bank may be variably dimensioned. Consider the following example:

```
BANK VARIABLE CONTEXT=JUNK NOMAXID

  INTEGER NELEMS          'Repeat count of variable block VBLK'

  BLOCK   VBLK(NELEMS<20) 'Variably dimensioned block'
     INTEGER A(10)         'Array A within VBLK'
     REAL    B(20)         'Array B within VBLK'
  ENDBLOCK
ENDBANK
```

In this example the dimension of VBLK is not fixed, but is in fact specified as an integer defined previously in the bank. Since only the last block of a bank may be variable dimensioned it follows that each bank can have at most one variable dimension.

Whenever a bank contains a variable dimension JAZELLE keeps track of two quantities associated with the dimension. These are called the "allocated repeat count" and the "used repeat count". The allocated repeat count is controlled by JAZELLE and represents the maximum value for the variable dimension which JAZELLE has allocated memory for. The used repeat count, which is stored in the variable associated with the variable dimension (NELEMS is this example), represents the elements of VBLK which have actually been used by the user. JAZELLE maintains the allocated repeat count, whilst the user controls the used repeat count, and it is the users responsibility to always check that the used repeat count is less than or equal to the allocated repeat count. If necessary the allocated repeat count can be increased or decreased by expanding or contracting the bank as discussed in the following chapter.

Whenever a bank is created the used repeat count is set to zero. The allocated repeat count is set to 10, unless a different value is specified in the template. In the above example the allocated repeat count would be set to 20 (controlled by the <20 in the BLOCK VBLK declaration). The value specified in the template can also be overriden when the bank is created, as discussed later.

2.4 Scope of Variables

Bank names and external (but not internal) block names are global. Hence their names must be unique and should follow SLD naming conventions (which the examples do not). Element names within templates and the names of parameters are local. Hence their names need not follow global naming conventions and need be unique only within the scope in which they are defined. The scope of any variable is the bank or block in which it is defined.

2.5 Use of Templates

Templates are used by MORTRAN when compiling a program and they are also read by JAZELLE when a program is run in order to control the creation and deletion of banks. It is extremely important that when a program runs it accesses the same version of the template which was read by MORTRAN at compile time, otherwise the program will not be able to access the data structures correctly. A template command is available to check the syntax of template files. The form of this command is identical on the VAX and the IBM, namely:

```
TEMPLATE template_name
```

where **template_name** is the name of a file whose file type is TEMPLATE, and which contains the template(s) to be compiled.

2.6 Detailed Syntax of Templates

A template file may contain bank templates and/or block templates. The only difference between a bank and a block is that a bank template defines a complete data structure, whilst a block template defines a structure which can only exist within a bank. Comments may be placed anywhere inside a template preceded by an ! (exclamation mark). Continuation lines are allowed, and identified by a trailing - (minus sign) on the line to be continued.

BANK or BLOCK

The BANK and BLOCK statements are used to delimit banks and blocks respectively. Each TEMPLATE file must contain at least one BANK or BLOCK statement.

Format

BANK *Bank_name Bank_qualifiers*
Declarative_statements

.
.
.

ENDBANK

BLOCK *Block_name*
Declarative_statements

.

.
.

ENDBLOCK

$$\textit{Bank\_qualifiers:} \left\{ \begin{array}{l} \textit{DEMAND\_ZERO} \\ \textit{CONTEXT=name} \\ \textit{MAXID=n / NOMAXID} \end{array} \right\}$$

DESCRIPTION

Bank_name is the name of a bank. It may contain 1-8 characters. It must start with a letter and consist of any of the characters (A_Z),(0_9) $ and _. **Block_name** is the name of a block. It may contain 1-8 characters. It must start with a letter and consist of any of the characters (A_Z),(0_9) $ and _. **Bank_qualifiers** are used to modify some properties of the bank. The legal qualifiers are summarized below.

- **DEMAND_ZERO** Requests that when this bank is created all elements should be set to zero.
- **CONTEXT=name** Defines the default context for this bank. This qualifier is mandatory. **name** may be any 1-8 character name.
- **MAXID=n** Specifies a maximum Id which banks in this family can take. If specified **n** must be in the range 1-32768. If NOMAXID is specified banks may have any Id up to 65536. This qualifier is mandatory.

A **Declarative_statement** may be either a JAZELLE **Parameter_declaration** or an **Element_definition**. The syntax for both of these are given below.

Parameter declaration

Used to define constants or parameters within a TEMPLATE.

Format

PARAMETER *Parm_name=Value [,Parm_name=value]...*

DESCRIPTION

Parm_name is the name of the parameter being defined. It must obey the normal JAZELLE naming conventions (ie 1-8 characters starting with a letter). **Value** is the value assigned to the parameter. It may be any string of up to sixteen characters.

Parameters may be defined anywhere within a template, but they can only be referred to following the point at which they are defined. The scope of the parameter is within the block/bank in which it is defined, and any sub-blocks contained within that block. The name of any parameters must be unique with respect to other parameters and variables in the same bank.

Parameters may be used as dimensions, as initial values and as initial value repeat specifiers. Parameter values are also accessible within MORTRAN or IDA programs, using the P% macro discussed later.

Element definition

Used to define elements within banks.

Format

Type Variable [Qualifiers] "Description"

Variable: *Variable_name [([First:]Last)][binding_expression] [/Initial_values/])*

Initial_values: *[repeat_count*]initial_value [,...]*

Type: $\left\{ \begin{array}{l} \textit{Intrinsic\_type} \\ \textit{Derived\_type} \\ \textit{User\_defined\_type} \end{array} \right\}$

$$\textit{Intrinsic\_type:} \quad \left\{\begin{array}{l} \textit{INTEGER} \\ \textit{HEX} \\ \textit{LOGICAL} \\ \textit{REAL} \\ \textit{PTR} \\ \textit{STRING*4} \\ \textit{INTEGER*4} \\ \textit{HEX*4} \\ \textit{LOGICAL*4} \\ \textit{REAL*4} \\ \textit{POINTER} \\ \textit{STRING*8} \\ \textit{INTEGER*2} \\ \textit{HEX*2} \\ \textit{LOGICAL*1} \\ \textit{REAL*8} \\ \textit{STRING} \end{array}\right\}$$

$$\textit{Derived\_type:} \quad \left\{\begin{array}{l} \textit{STRING*20} \\ \textit{KEY} \\ \textit{COMPLEX} \end{array}\right\}$$

$$\textit{Qualifiers:} \quad \left\{\begin{array}{l} \textit{FMT=format\_descriptor} \\ \textit{TABULATE} \\ \textit{HEADING=column\_heading} \end{array}\right\}$$

DESCRIPTION

Type defines the type of the variable. There are three categories of JAZELLE types, intrinsic, derived and user defined. Generally the three types may be used in exactly the same way, although there are a few restrictions which are mentioned below. A user defined type is indicated either by using the word BLOCK for type, in which case the block definition must immediately follow, or by specifying type as the name of an external block defined elsewhere.

Qualifiers all effect the way in which a variable is displayed and will be discussed later in the chapter on displaying banks.

Description is a 1-80 character description of the variable. When a detailed dump of a bank is requested this description will be printed with each variable dumped.

Variable_name is a 1-8 character name. The name must start with a letter and must contain only letters, digits or $ and _.

First and **Last** if present denote that the variable is a vector with indices covering the range **First- Last**. If **First** is omitted it defaults to 1. The dimensions may be specified as either:

- An integer
- A JAZELLE parameter
- A variable (See "variable dimension blocks" above.)

Binding expression may only be specified when the variable type is POINTER or KEY. The format of the binding expression is –>family_name, where family_name is the name of the family which the pointer or key will point to.

Initial_Values defines values which this element will be initialized to when the bank is created. Only JAZELLE intrinsic types may have initial values.

Repeat_count is an integer or JAZELLE parameter specifying the number of times the proceeding initial value is to be repeated (c.f. FORTRAN DATA statements). The total number of values specified must be one if the associated variable is a scalar, or the number of elements if it is a vector.

Initial_value Is the value to be assigned to the variable. The type of the initial value must be appropriate to the variable. The initial_value can also be specified as a JAZELLE parameter. The format of the value specification for various types is shown in the table below.

Type	Syntax
INTEGER, HEX	[%X I %D] [+ I -] Digits
STRING	"string"
LOGICAL	T I TRUE I YES F I FALSE I NO
REAL	[+ I -] [digits][.][digits][E[+ I -]digits]
POINTER, KEY	integer I bankname(*) I bankname(id)
COMPLEX	real I (real,real)

3 Manipulating Banks

3.1 Introduction

Data in Jazelle are allocated in banks, with a single bank being a contiguous data structure, consisting of a header and a data region. Banks of common format are grouped together into families. A bank is uniquely identified by the combination of an 8-character family name, a 16-bit bank ID, and an 8-character context name. For most applications the family name and Bank ID suffice to uniquely identify the bank.

Banks may be dynamically allocated or deleted during program execution, and thus do not have static addresses in memory. Hence, to access data in a bank, one must first have a pointer to the bank specifying its location in memory. Given such a pointer, data within the bank can be accessed directly, as will be discussed in Chapter 4. The present chapter deals with the creation and deletion of banks and the means of obtaining pointers to specific banks.

3.2 Usage

The principal routines for the manipulation of single banks are:

- JZBADD - Create (add) a new bank
- JZBDEL - Delete a bank
- JZBEXP - Expand/contract the size of a bank
- JZBFND - Find pointer to an individual bank
- JZBLOC - Locate a family of banks
- JZBNXT - Find the next bank of a given family

3.3 Examples

Jazelle routines make extensive use of optional arguments in order to allow simple calling sequences. In the examples below, we do not attempt to fully describe all arguments, but rather present some simple examples as introductory material. The full description of the individual routines is given in Chapter 14.

All programs must include a one-time call to JZSTRT, which initializes the Jazelle system:

```
$CALL JZSTRT ERROR RETURN;
```

In its simplest form, a call to create a new bank in the family NONSENSE would look like

```
$CALL JZBADD('NONSENSE', NSPTR, NSID ) ERROR RETURN;
```

Here 'NONSENSE' is the name of the desired family; NSPTR and NSID are returned as the pointer and ID of the newly created bank. The same bank could be deleted through the call

```
$CALL JZBDEL('NONSENSE', NSID) ERROR RETURN;
```

If, in another subroutine, the pointer to the bank were unknown, it could be found using

```
$CALL JZBFND('NONSENSE', NSPTR, IDOUT, NSID) ERROR RETURN;
IF (NSPTR .EQ. 0) [ ... ]
```

Here 'NONSENSE' and NSID are input parameters specifying the desired family name and bank ID. NSPTR is the returned pointer to the bank. IDOUT is in this case superfluous, and is the returned ID of the bank found (= NSID in this case). The argument NSID may also be input as 'FRST' or 'LAST', or may be omitted entirely, in which case it defaults to 'FRST'. This allows one to find the first or last banks of a family. For example,

```
$CALL JZBFND('NONSENSE', NSPTR, NSID) ERROR RETURN;
IF (NSPTR .EQ. 0) [ ... ]
```

would find the first bank of family NONSENSE, and return its pointer and bank ID in NSPTR and NSID respectively.

The check against 0 of the returned pointer NSPTR is required in order to determine whether or not the requested bank exists. Most JAZELLE routines do not consider the non-existence of a bank to be an error condition, instead they return an error severity of information. Thus an alternative way to test for the existence of a bank would be to specify:

```
$CALL JZBFND('NONSENSE', NSPTR, NSID) ERROR RETURN;
IF $SEVERITY>$SUCCESS [ ... ]
```

See the SLD error system manual for more details of how to test for error conditions.

It is frequently desirable to "chain" through the banks of a family. Given a pointer NSPTR to one bank of a family, a pointer NXTPTR to the next bank of the same family can be obtained using

```
$CALL JZBNXT(NSPTR, NXTPTR) ERROR RETURN;
IF (NXTPTR .EQ. 0) [ ... ]
```

Note: **The use of JZBNXT is generally discouraged in cpu-time sensitive applications. See Section 4.6 for alternative ways of accessing the next bank in a family.**

The reader is referred to Chapter 14 for a more complete specification of these and other routines.

3.4 Implementation Details

3.4.1 Bank ordering within families

In calls to JZBADD application routines may specify a particular ID, or allow JZBADD to assign an arbitrary ID. In all cases, banks of a given family will be linked together in order of monotonically increasing, but not necessarily consecutive, bank ID. If assigned by JZBADD, the bank ID of a created bank will be one greater than that of the last previously existing bank of that family.

3.4.2 Bank header

With the exception of the header parameters JB$VBALO and JB$FORPT, application routines will rarely need to access data in the bank header. For reference we list the elements of the bank header and their byte offsets from the data region of the bank. These parameters are defined in MORTRAN macros whose use will be discussed in Chapter 4.

Parameter	Use
JB$FORPT	Pointer to next bank of family
JB$BAKPT	Pointer to previous bank of family
JB$ID	Bank ID
JB$FAMLY	Pointer to family block
JB$VBALO	Allocated variable block count (banks with variable dimensions only)

4 Referencing JAZELLE Data Elements from Mortran

4.1 Access to Normal Variables

By exploiting the power of MORTRAN it has been possible to make references to variables in JAZELLE banks look very similar to normal array references in FORTRAN. References to JAZELLE variables always contain a % sign so that people reading the code can easily differentiate JAZELLE references from normal array references.

Before a JAZELLE element can be accessed a pointer which points to the bank must be available. A pointer can be declared using the POINTER statement. A typical POINTER declaration looks like:

```
POINTER BPTR-->NONSENSE, CUBEPTR-->CUBE;
```

This statement declares a pointer BPTR which will be used to point to banks in family NONSENSE, and a pointer CUBEPTR which will be used to point to family CUBE. MORTRAN must know which family BPTR will be used with so that it can correctly calculate the offsets of elements within the bank. After BPTR has been assigned a value (see previous chapter), elements of the bank can be referred to in the following manner.

```
BPTR%(A)
BPTR%(D(5))
BPTR%(X(K-1))
```

In the first example a scalar element A in bank NONSENSE is referenced. In the last two examples array elements are referenced. The array index can consist of any valid FORTRAN expression.

References to JAZELLE variables can be used anywhere that a normal variable may be used in FORTRAN, e.g.

```
A=BPTR%(A);
BPTR%(A)=8*I;
BPTR%(D(5))=BPTR%(A)*BPTR%(B);
```

When the JAZELLE variable to be accessed exists within a sub-block then a path name must be used to access the variable. For example to access an element of bank CUBE defined in chapter 2 a reference of the form

```
CUBEPTR%(X(1),Y(2),Z(3))
```

would be used. The POINTER macro can also be used to declare arrays of pointers as in the following example.

```
POINTER APTR(10)-->NONSENSE;

APTR(4)%(A)=5;
APTR%(A)=0;       " using pointer array without index defaults "
                  " to element 1                               "
```

To access any JAZELLE variables, routines should include the signpost common block /JAZELL/. Thus to access the NONSENSE bank it is necessary to include the following line at the top of the program.

```
"&COPY FROM JAZELL.COMMON  "
```

4.2 Indexed Pointers

Indexed pointers provide an alternative method of accessing banks. Instead of using the POINTER macro users may use the $USEBANK macro. The $USEBANK macro accesses a common block consisting of an array which contains pointers to each bank in a family.

Indexed pointers are simpler to use than local pointers, especially for people who have suffered prolonged exposure to FORTRAN and who may be unfamiliar with the use of pointers in other languages. On the other hand indexed pointers suffer from some limitations compared to local pointers:

- Due to the necessity of generating a fixed size common block to contain the indexed pointer it is necessary to specify a maximum ID that this family will contain. This maximum ID is then used as the upper dimension of the indexed pointer array. The maximum ID is specified using the MAXID qualifier in the bank template.
- Indexed pointers are implicitly one dimensional, whereas local pointers may have any number of dimensions.
- The use of explicit ID's for banks can be very restrictive in some advanced applications. For instance overlaying two event is trivial if they are described using local pointers, but is much more difficult if they are described using explicit ID's as the ID's in the two events will clash.

Indexed pointer arrays are declared in each routine that they are used in, using the $USEBANK macro. The format is:

```
$USEBANK family1 [,family2] ;
```

Any number of families may be specified on one $USEBANK statement, with each name separated from the previous name by a comma. Each family specified causes an indexed pointer array of the same name as the family to be accessed. The layout and contents of the indexed pointer array are given below:

Element[1]	Meaning
J$FIRST	ID of first bank in family[2] (i.e. lowest ID existing)
J$LAST	ID of last bank in family[2] (i.e. highest ID existing)
J$MAXID	Maximum ID allowed (as specified by MAXID in template)

[1]Array is dimensioned (J$START:MAXID).

[2]If there are no banks existing then element J$FIRST will be greater than element J$LAST, thus ensuring that loops from first to last will be zero trip.

Element[1]	Meaning
J$NUMBER	Number of existing banks in family
J$FAMILY	Family pointer for this family
0-MAXID	Pointer to bank of this ID or 0 if bank does not exist

[1]Array is dimensioned (J$START:MAXID).

The indexed pointer is in a common block of the same name as the family. Due to the temporary restriction that SLD common block names should be limited to seven characters, the names of families used with the $USEBANK statement should also be kept to seven characters. Failure to comply with this restriction will cause PREPMORT to generate a warning message when the routine is compiled.

In the current version of JAZELLE it is necessary for each indexed pointer array to be initialized using the routine JZPIDX. Since the array is stored in a common block it is only necessary for it to be initialized once in any program. (Subsequent initializations of the same indexed pointer will be ignored.) Once the array has been initialized for a given program JAZELLE will continue to keep it up to date as banks are added or deleted.

An example of the use of indexed pointers is given below:

```
$USEBANK TRACK;

ETOT=0;

DO ID=TRACK(J$FIRST),TRACK(J$LAST)
  [ IF TRACK(ID)<>0 [ ETOT=TRACK(ID)%(ENERGY); ] ]
```

4.3 Accessing STRING Variables from JAZELLE banks

The representation of STRING variables within JAZELLE is not exactly the same as the representation of CHARACTER variables within FORTRAN. Therefore a set of JAZELLE routines exist which can be used to convert CHARACTER to STRING and STRING to CHARACTER. There are three types of JAZELLE string variables, namely STRING*4, STRING*8 and STRING. One routine exists for converting to/from each type of string variable. The following example demonstrates the use of the conversion routines.

```
CHARACTER*80 C80,JZC ;  "JZC must be declared as large as the largest "
CHARACTER*8  C8 ,JZC8;  "string it will be used to convert            "
CHARACTER*4  C4 ,JZC4;

INTEGER      JZS,JZS4;
REAL*8       JZS8;      "JZS8 must be declared REAL*8!                "

POINTER SPTR-->SBANK;

" String --> Character "

C80=JZC(80,SPTR%(S80));
C8 =JZC8(SPTR%(S8));
C4 =JZC4(SPTR%(S4));

" Character --> String "
```

```
SPTR%(S4)=JZS4(C4);
SPTR%(S8)=JZS8(C8);
$CALL JZS(80, SPTR%(S80),C80);
```

Where the template for SBANK is:

```
Bank SBANK CONTEXT=EXAMPLE NOMAXID "Example of using STRINGS"

    STRING    S80(80) "String of length 80 characters"
    STRING*8 S8      "String of length  8 characters"
    STRING*4 S4      "String of length  4 characters"

Endbank
```

4.4 Referencing Items in the Bank Header

Elements of the bank header are referenced in a manner completely analogous to elements of the data. All that is needed is to know the names by which JAZELLE refers to the items in the header. These names are all given in Section 3.4.2.

Items in the header can now be referred to using the references of the form:

```
BPTR%(JB$ID)
BPTR%(JB$FORPT)
```

The elements in the bank header may be freely examined, but the only items normally of interest to users are JB$VBALO, JB$FORPT and JB$BAKPT. Care should be taken never to attempt to modify any element in the bank header as this will almost certainly result in disaster.

4.5 Access to JAZELLE parameters

The values assigned to JAZELLE parameters in TEMPLATE definitions may be accessed from MORTRAN programs using the P% macro. For instance, to access the parameter DIM from bank CUBE (Section 2.2) a statement of the form

```
A=P%(CUBE,DIM);
```

is used. It is, of course, not possible to assign values to parameters. Parameters defined within sub-blocks are accessed using a path-name is direct analogy to accessing variables from within sub-blocks; e.g.

```
A=P%(CUBE,X,Y,VALUE);
```

4.6 Looping over Banks

It is often desirable to loop over all of the banks in a family. One way of doing this is to use the JZBFND routine to find the first bank in the family and then the JZBNXT routine to progressively step to the next bank until the last bank is reached. This method has the disadvantage of requiring one routine call for each bank and is thus too inefficient for many applications.

A better way of looping over families is to make use of the JZBLOC routine and the JB$FORPT element in the bank header. The JZBLOC routine returns a pointer to the entire family of banks. This pointer is referred to as the "family pointer". The family pointer is always available and never changes during the execution of a program and can therefore be found once and then stored in a local variable for future use. The following example illustrates how to loop over a family of banks using JZBLOC and JB$FORPT.

```
POINTER BPTR-->NONSENSE, BPTR0-->NONSENSE;
LOGICAL FIRST/.TRUE./;

IF FIRST [ $CALL JZBLOC('NONSENSE',BPTR0) ERROR RETURN; ]

BPTR=BPTR0;
LOOP [
  BPTR=BPTR%(JB$FORPT);
  IF BPTR=0 [ EXIT; ]

  ...

  ]
```

Note that this method of looping over banks will work even if there are no banks in the family (the loop will then be a zero-trip loop). Since this construct is so common in programs written using JAZELLE a special statement called BANKLOOP has been incorporated into SLD MORTRAN. Using BANKLOOP the above example would be replaced with:

```
POINTER BPTR-->NONSENSE, BPTR0-->NONSENSE;
LOGICAL FIRST/.TRUE./;

IF FIRST [ $CALL JZBLOC('NONSENSE',BPTR0) ERROR RETURN; ]

BANKLOOP BPTR0,BPTR [

  ...

  ]
```

The BANKLOOP example would work in exactly the same manner as the previous example, but is slightly clearer to read. EXIT and NEXT statements can be used inside a BANKLOOP construct in the normal way. See the SLD MORTRAN manual for a complete description of the BANKLOOP statement.

Yet another way to loop over banks in a family is to use indexed pointers. The following example illustrates how to create a loop using indexed pointers.

```
$USEBANK NONSENSE;
LOGICAL FIRST/.TRUE./;
INTEGER I;

IF FIRST [ $CALL JZPIDX('NONSENSE') ERROR RETURN; ]

DO I=NONSENSE(J$FIRST),NONSENSE(J$LAST) [
  IF NONSENSE(I)=0 [ NEXT; ]

  ...

  ]
```

4.7 The Q% Macros

In addition to the ways of accessing variables described earlier another means of accessing variables also exists, namely to use the Q% set of macros. These macros are obsolete having been superceded by the macros described earlier but they are described briefly here as they may be encountered in some older programs.

```
Q%(BPTR, NONSENSE, A)       is equivalent to   BPTR%(A)
Q%(BPTR, NONSENSE, D(5))    is equivalent to   BPTR%(D(5))
Q%(BPTR, NONSENSE, P(3),I)  is equivalent to   BPTR%(P(3),I)
```

5 Producing Dumps of JAZELLE Banks

5.1 Introduction

This chapter deals with producing human readable dumps of the contents of JAZELLE banks. Routines are available to produce an index of all the existing banks, to produce dumps of banks, and to produce tabulated output from banks.

When outputting banks JAZELLE attempts to make the output as readable as possible. In so doing it uses an algorithm which attempts to maximize the aesthetic quality of the output by varying the number of columns in the output and the space allocated to each element. When outputting individual elements JAZELLE always attempts to output the variable using the minimum space possible, except that all elements of an array, or all items in one column are always output using the same format. Real numbers are always output to five significant figures [1]. Users may override JAZELLE's default output format by using an explicit format qualifier in the TEMPLATE for any element, e.g.

```
REAL X FMT=F6.3 "This variable is always output in F6.3 format"
```

All JAZELLE dump routines have a common set of three optional arguments that control the output from the routines. These arguments are described below:

- **LUN** - Logical unit number for output, default 6.
- **LEVEL** - Level of detail requested, 0=Brief, 1=Average, 2=Verbose. Default varies.
- **WIDTH** - Width of output, default 80. This argument controls the maximum width of the output produced. Useful values are normally 80 when the output will be viewed at the terminal, and 132 when it will be printed on the line printer.

5.2 Usage

The following routines are discussed in this chapter:

- JZINDX - Produce an index of all banks or of one family
- JZBDMI - Dump a bank or a family of banks
- JZBDMP - Dump a bank
- JZBTBL - Tabulate one bank or a family of banks
- JZTDEF - Add a column to a table

[1] See Appendix A for a method of overriding this default

5.3 Examples

This chapter gives only some example of how to use the various routines discussed. Complete details of all arguments etc. are given in Chapter 14.

5.3.1 Producing an Index of Banks

The routine JZINDX is used to produce an index of currently defined banks. It can be used to produce an index of all banks, or of the banks of one particular family, as shown by the examples below:

```
$CALL JZINDX(' ')       ERROR RETURN;
$CALL JZINDX('MCROOTS') ERROR RETURN;
```

The output produced by the calls might look something like:

```
Bank      Number Title
MCLUNDPM       1 MC generator control bank: one per run
MCROOTS       22 Particle genealogy bank
MCTRACK       57 MC particle bank, 1/particle
MCEVENT        1 Monte Carlo event bank, one per event
EVENTHDR       1

Family: MCROOTS       Version:  0.00      Context: EVENT
 Title: Particle genealogy bank
        ID     Pointer   Size  Alloc    Used
         1      157812    100      5       5
         2      157866     60      0       0
         3      157944    156     12      12
         5      158034     68      1       1
         6      158092     76      2       2
        10      158252     76      2       2
        11      158308     68      1       1
        13      158400     76      2       2
        14      158458     76      2       2
        15      158516     76      2       2
        16      158574     76      2       2
        17      158632     76      2       2
        18      158692     84      3       3
        20      158786     84      3       3
        23      158912     76      2       2
        24      158968     68      1       1
        28      159128     76      2       2
        29      159186     76      2       2
        30      159244     76      2       2
        32      159336     76      2       2
        39      159598     76      2       2
        42      159724     76      2       2
```

5.3.2 Producing a Dump of a Bank or Family of Banks

Complete dumps of banks can be produced by the routines JZBDMI and JZBDMP. The only difference between these two routines is that JZBDMI accepts a bank name and Id as arguments whereas JZBDMP accepts a pointer as an argument. Typical uses of these routines plus some typical output are given below.

```
$CALL JZBDMI('MCEVENT')         ERROR RETURN;
$CALL JZBDMI('MCEVENT',1,6,2)   ERROR RETURN;
$CALL JZBDMI('MCTRACK',1)       ERROR RETURN;
$CALL JZBDMI('MCTRACK','ALL*')  ERROR RETURN;
$CALL JZBDMP(TRKPTR)            ERROR RETURN;
```

```
Family: MCEVENT    ID:     1   Template Version: 0.00
Title:  Monte Carlo event bank, one per event

Name     Value    Name     Value    Name     Value    Name     Value
----     -----    ----     -----    ----     -----    ----     -----
EVENTNUM 1        BEAMWT   1.0000   ANNIWT   1.0000   HADRWT   1.0000
BEAMCOMP 0        RADCCOMP 0        ANNICOMP 0        HADRCOMP 2
DCAYCOMP 0        EMINENGY 47.000   EPLSENGY 47.000   PCM4VEC  Below
PTOTCM   0.0000   XPRIMVTX Below    TPRIMVTX 0.0000   NTRACKS  57
NCHGTRCK 14       NNEUTRCK 21

PCM4VEC  1: 0.0000  0.0000  0.0000 94.000
XPRIMVTX 1:0.0000  0.0000  0.0000

Family: MCEVENT    ID:     1   Template Version: 0.00
Title:  Monte Carlo event bank, one per event

Name     Value  Offset Type     Comment
----     -----  ------ ----     -------
EVENTNUM 1           0 I*4      Event number
BEAMWT   1.0000      4 R*4      Beam weight
ANNIWT   1.0000      8 R*4      Annihilation weight
HADRWT   1.0000     12 R*4      Hadronization weight
BEAMCOMP 0          16 I*2      Beam generator completion code
RADCCOMP 0          18 I*2      Radiative correction completion code
ANNICOMP 0          20 I*2      Annihilation completion code
HADRCOMP 2          22 I*2      Hadronization completion code
DCAYCOMP 0          24 I*2      Decay generator completion code
EMINENGY 47.000     28 R*4      Electron energy before radiation
EPLSENGY 47.000     32 R*4      Positron energy before radiation
PCM4VEC  Below      36 R*4      Px,Py,Pz,E  of annihilation cm system
PTOTCM   0.0000     52 R*4      /P/ of annihilation cm system
XPRIMVTX Below      56 R*4      x,y,z of primary vertex in cm
TPRIMVTX 0.0000     68 R*4      c*time of primary vertex in cm
NTRACKS  57         72 I*4      Total no of particles
NCHGTRCK 14         76 I*4      No of charged tracks in final state
NNEUTRCK 21         80 I*4      No of neutral particles in final state

PCM4VEC  1: 0.0000  0.0000  0.0000 94.000
XPRIMVTX 1:0.0000  0.0000  0.0000
```

5.3.3 Tabulating Banks

Outputting banks in tabular format is slightly more complex than producing dumps of banks. The reason for this is that in general it is desirable to restrict the table to a subset of the bank elements so that the table fits onto the screen or output device being used. Therefore, before a table can be produced the subset of elements of a given bank must be selected. There are two ways to do this, either by using the TABULATE qualifier in the TEMPLATE or by using the JZTDEF routine.

To define a table using the TEMPLATE it is merely necessary to add the qualifier TABULATE to each element that is to be output. By default JAZELLE uses the name of the variable as the heading for the column. The default can be changed by using the HEADING="heading" qualifier to specify a different heading. If the HEADING qualifier is specified it is not necessary to also specify the TABULATE qualifier.

```
Bank Nonsense Nomaxid Context=Junk

   Integer I   TABULATE
   Integer J

   Block A     TABULATE

     Real E
     Real P    HEADING="Momentum"

   Endblock

Endbank
```

In the above example the elements I, A and P of bank nonsense will be included in the table. Note that elements of a sub-block will only be included in a table if the TABULATE option is specified on the declaration of the block in which they are included, i.e. If the second TABULATE in the above example were removed only element I would be included in the table.

A table defined using the TABULATE or HEADING qualifiers in the TEMPLATE for a family is called the default table for that family. Tables can also be defined using the JZTDEF routine as in the following example:

```
Table=0;
$CALL JZTDEF(table,'MCTRACK.TRACKTYP','Type'  ,' '   ) ERROR RETURN;
$CALL JZTDEF(table,'MCTRACK.CHARGE'  ,' '     ,' '   ) ERROR RETURN;
$CALL JZTDEF(table,'MCTRACK.MASS'    ,' '     ,' '   ) ERROR RETURN;
$CALL JZTDEF(table,'MCTRACK.P4VEC(1)','Px'    ,' '   ) ERROR RETURN;
$CALL JZTDEF(table,'MCTRACK.P4VEC(2)','Py'    ,' '   ) ERROR RETURN;
$CALL JZTDEF(table,'MCTRACK.P4VEC(3)','Pz'    ,' '   ) ERROR RETURN;
$CALL JZTDEF(table,'MCTRACK.P4VEC(4)','Energy','F6.3') ERROR RETURN;
```

The first argument to this routine is a pointer to the table being defined. If JZTDEF is called with the first argument set to zero it creates a new table and sets the pointer to point to it. Otherwise it adds a new column to the table pointed to.

The second argument describes the bank element to be tabulated. The third argument is the heading for the column. If the third argument is blank the name of the element is used. The fifth and final argument may be used to specify the Fortran format to be used to output elements in the column. If specified as a blank JAZELLE will substitute a suitable default.

To actually produce the table the JZBTBL routine is used. For example:

```
$CALL JZBTBL('MCTRACK','ALL*',table) ERROR RETURN;
```

If a table is specified as the third argument it is used, otherwise the default table (as defined in the template) is used. The output produced by the above call might look something like:

```
Family: MCTRACK  Template Version: 0.00
Title:  MC particle bank, 1/particle
```

```
 #  Type Charge   Mass    Px          Py          Pz          Energy
 -- ---- -------- ------- ----------- ----------- ----------- -----------
  1  502 -0.33333 0.32500 -41.344     -17.252     + 9.1365    +45.722
  2  500 +0.0000  0.0000  - 0.51671   + 3.1523    - 1.1134    + 3.3828
  3 -502 +0.33333 0.32500 +41.860     +14.100     - 8.0230    +44.895
  4   17 +1.0000  0.14000 +24.617     + 8.7517    - 5.1469    +26.629
  5   19 +0.0000  0.49800 - 7.6467    - 3.1907    + 1.9733    + 8.5319
  6  -27 -1.0000  0.76700 + 6.0034    + 1.5807    - 0.51009   + 6.2760
  7  -42 +0.0000  0.94000 + 4.7287    + 1.5774    - 1.0431    + 5.1788
  8   42 +0.0000  0.94000 + 4.0263    + 1.9004    - 1.1558    + 4.6949
  9   17 +1.0000  0.14000 + 0.50161   + 0.43537   + 0.17883   + 0.70196
 10  -28 -1.0000  0.89200 + 0.25688   + 1.1790    - 1.0720    + 1.8441
 11  -19 +0.0000  0.49800 - 9.0844    - 3.0334    + 2.0335    + 9.8036
 12   18 +1.0000  0.49400 + 0.75052   - 0.090912  + 0.093900  + 0.90797
 13  -28 -1.0000  0.89200 + 0.069470  + 0.45814   + 0.43293   + 1.0944
 14  -67 +1.0000  1.3870  - 0.55262   - 0.13075   + 0.33953   + 1.5367
 15  -27 -1.0000  0.76700 - 8.4975    - 4.2734    + 1.5031    + 9.6600
 16   64 -1.0000  1.2330  - 2.3704    - 0.23836   - 0.60193   + 2.7492
 17   23 +0.0000  0.13500 - 3.0519    - 0.73999   + 1.0949    + 3.3284
 18   24 +0.0000  0.54900 - 2.4240    - 0.81817   + 1.0349    + 2.8138
 19   17 +1.0000  0.14000 - 4.7062    - 1.7182    + 0.55913   + 5.0431
 20   34 +0.0000  0.78300 - 2.6202    - 1.6488    + 0.28584   + 3.2061
 21   38 +0.0000  0.49800 - 7.6467    - 3.1907    + 1.9733    + 8.5319
 22  -17 -1.0000  0.14000 + 4.1643    + 1.3143    - 0.60319   + 4.4105
 23   23 +0.0000  0.13500 + 1.8391    + 0.26643   + 0.093094  + 1.8655
 24  -19 +0.0000  0.49800 + 0.46076   + 0.83127   - 0.79479   + 1.3353
 25  -17 -1.0000  0.14000 - 0.20389   + 0.34771   - 0.27717   + 0.50882
 26   38 +0.0000  0.49800 - 9.0844    - 3.0334    + 2.0335    + 9.8036
 27  -18 -1.0000  0.49400 + 0.23707   + 0.29870   + 0.52052   + 0.81265
 28   23 +0.0000  0.13500 - 0.16760   + 0.15944   - 0.087586  + 0.28179
 29  -45 +1.0000  1.1970  - 0.60968   - 0.14671   + 0.30755   + 1.3859
 30   23 +0.0000  0.13500 + 0.057061  + 0.015966  + 0.031982  + 0.15086
```

6 Lists

6.1 Introduction

Jazelle lists provide a mechanism for grouping banks belonging to different families. Lists are useful in situations where a single operation can be performed on multiple banks (e.g. bank deletion, IO). As an alternative to lists, see also the chapter on context.

Two types of lists are implemented in Jazelle: symbolic and pointer. For symbolic lists, banks belonging to the list are stored in symbolic form, i.e. by family name and bank ID. These are intended for long-lived applications (e.g. applicable to all events) in which the banks constituting the list are frequently created and deleted over the life of the list. For pointer lists, the banks belonging to the list are directly specified by pointers to the banks. Pointer lists are primarily intended for applications in which lists are dynamically created to access particular sets of already existing banks (e.g. the lifetime of the list is less than or equal to that of the relevant banks). All Jazelle list routines are applicable to both symbolic and pointer lists.

In its simplest form, a Jazelle list consists of a set of bank identifiers, either by family name and ID, or by pointer. Additionally, members of a list may be specified by the keyword 'ALL*', in order to specify all members of a family.

To gain additional structure, each entry in a list also contains an indirection flag. When a list operation is performed, and the indirection flag for a given list entry is set, then the specified operation is not performed on the indicated bank. Instead, the indicated bank is interpreted as another list upon whose entries the operation is to be performed. Thus one can construct a hierarchy of lists, with any given list referring to additional lists.

6.2 Usage

The user is referred to the detailed description of these routines in Chapter 14.

- JZLCRE - Create a list
- JZLINC - Add (INClude) an entry to a list
- JZLREM - Remove an entry from a list
- JZLWIP - Delete (WIPe) all banks specified by a list
- JZIOWR - Write a Jazelle record containing all banks specified by a list

7 Context

7.1 Introduction

Jazelle Context is used for two purposes. The first is to allow one to group families together without having to specifically generate Jazelle lists. For each defined family, the Jazelle template defines a default context for that family. Thus global operations can be performed all families of the same context (e.g. EVENT) in a fashion similar to that of lists (e.g. bank deletion, I/O).

The second purpose of Jazelle context is to provide a finer-grained mechanism for grouping banks which belong to the same family. For example, in online applications it may be desirable for a single process to store a single event for one-event purposes, while at the same time be analysing events as they occur. Since many of the families used may be common to the two applications, a mechanism is needed to distinguish the two events. Jazelle allows families to contain banks belonging to different Contexts, thus allowing one to distinguish between two. As a second example, in Monte Carlo applications, it may be desirable to combine two different events or fragments of events into a single event. Since these share families, with each event using the same bank ID's for the same family, a similar mechanism is needed. Thus, while for most purposes a bank is uniquely defined by family and bank ID, for some applications it is desirable to define uniqueness by context, family, and bank ID.

The logic of banks, families, and context is as follows:

- A bank has a single, unchanging bank ID and belongs to a single unchanging Family. At any one time a bank is associated with a single context, but may be switched from one context to another. Within a family, bank ID's must be unique within a single context, but need not be unique across different contexts.
- A Family may be associated with more than one context. At any one time only a single context is active within a family. Any calls to Jazelle routines specified by Family name rather than direct bank pointers will find or operate on only those banks associated with the currently active context for that family.
- Different families may be associated with the same context. A given context may be activated and deactivated. When a given context is activated, it becomes the currently active context for all families with which it has been associated. When a context is deactivated, the active context of a given associated family reverts to that which was in effect immediately prior to the activation.

7.2 Usage

When the first bank of a family is created, the active context for that family is set to the default specified in the template. In most applications this never need be changed, and context name may be used interchangeably with Jazelle List operations (e.g. JZLWIP, JZIOWR).

Note: **Currently context in JAZELLE is only partially implemented. Currently banks can only exist in the default context specified in their template. The only operations currently available are context wipe JZXWIP (delete all the banks in a given context) and I/O operations. In general performing an operation on an entire context is much quicker than performing the same operation on many individual banks within the context.**

The remainder of this discussion is given only as an indication of the direction in which JAZELLE implementation is expected to proceed.

For applications in which Context switching is desired, banks (or all banks of a family) may be switched from their current context to another through the routine JZXCHB, specifying the new context name, the family name and bank ID (or 'ALL*'). Similarly a Jazelle list of banks may be switched through the routine JZXCHL. These routines associate the specified banks and families with the new context, but do NOT activate that context. (Thus the switched banks become "invisible").

A context may be activated through the routine JZXACT, specifying the desired context name. The specified context becomes the active context for all families which have been associated with that context. (Within those families all banks not associated with the new context become "invisible".)

In order that application routines may obtain predictable results, it is important that applications which activate a context also take the responsibility for deactivating the context when finished (the push/pop concept) to restore the previous context(s). This is done through the routine JZXDAC, specifying the context name to be deactivated. This will cause the active context of all associated families to revert to those which were in effect before the activation.

7.3 Implementation Details

Contexts in JAZELLE are implemented using the concept of virtual memory zones. The native virtual memory services of the host operating system are used to keep each jazelle context in a separate virtual memory pool. This enables operations such as deleting the context (context wipe) or performing IO of an entire context to be achieved far more efficiently than operations on arbitrary sets of banks.

8 Using Pointers with JAZELLE

8.1 Introduction

Because JAZELLE banks can be continually allocated and deallocated during the execution of a program the compiler cannot know where in memory the banks will be located. Because of this it is always necessary to have a pointer to a bank before elements in the bank can be accessed.

Pointers are supported as a new variable type in JAZELLE, and can be declared either in banks (using the template) or in user's programs (user pointers). As discussed in earlier chapters, pointers are declared, in either of these cases, using a POINTER statement of the form:

```
POINTER  name-->family
```

where **name** is the name of the pointer and **family** is the name of the family of banks that the pointer will be used to point to.

Pointers in user routines can optionally be registered, as discussed in the next section. Registering of pointers is primarily a debugging aid.

8.2 Registering Pointers

There are two routines that are used in conjunction with registering pointers:

- JZPREG - Register user pointers
- JZPDMP - List user pointers

The arguments for JZPREG are as follows:

```
$CALL JZPREG(comment,context,pointer,nptrs) ERROR RETURN
```

Where the arguments have the following meaning:

- **comment** is a comment to be associated with this pointer. This comment is used to identify the pointer when the pointers are printed out by a call to JZPDMP or during a JAZELLE post mortem. The comment can be any string but will typically be the name of the pointer or the common block containing the pointer.
- **context** is a string describing the context in which the pointer is to be registered. This argument is discussed further below.
- **pointer** is the pointer, or array of pointers, to be registered.
- **nptrs** is the dimension of pointer.

Registering a user pointer has two effects, firstly the pointer and its value will be output whenever JZPREG is called or during a JAZELLE post mortem, and this can be a powerful debugging tool. Secondly registered pointers will be zeroed if the bank which they point to is deleted.

The second argument to JZPREG is the context in which the pointer is to be registered. The context of a pointer has two functions, first when a context is wiped, any pointers registered in that context are automatically deregistered. This can be useful for registering pointers which are only used in a particular section of the code and which are of no interest once the program leaves this sections of code. Secondly, when a bank is deleted only pointers which are registered in the same context as the deleted bank and which point to the deleted bank are zeroed. This is mainly to increase execution efficiency.

8.3 Implementation Details

JAZELLE pointers are actually implemented as indirect pointers. That is to say that the pointer actually points to an intermediate link area which in turn points to the bank, as shown below.

```
+----------+                +------------+               +-------+
| Pointer  | ------------>  | Link area  | ---------->   | Bank  |
+----------+                +------------+               +-------+
```

Pointers are implemented in this way so that if a bank is moved it is only necessary to change the value in the link area and the user pointer will continue to point to the bank. The indirection is normally totally transparent to the user since all of the Mortran macros and JAZELLE routines automatically take care of the indirection.

A different problem occurs when a bank is deleted, in which case the pointer is left pointing to the middle of nowhere. In the case of registered user pointers and of bound pointers in banks, JAZELLE will take care of zeroing the pointer. In other cases it is up to the user to be aware of the fact that the bank has been deleted. In practice this should not prove to be a problem.

9 Sequential Input/Output

9.1 Introduction

Jazelle contains facilities for both sequential and indexed IO. In sequential IO records are always read in the same order in which they were written, while indexed IO allows records to be read in an arbitrary order, using an index to specify which record is to be read next. This chapter describes the basics of sequential IO while Chapter 10 goes on to describe the differences for indexed IO.

Jazelle provides for input and output of multiple banks as single logical records to sequential devices. The banks to be written can either be specified by a Jazelle list or by context name (e.g. EVENT). In general it is much faster to write out an entire context than to write out the same banks using a JAZELLE list. JAZELLE contains facilities to allow data to be written on one machine and later be read on a different type of machine (at present IBM and VAX machines are supported).

In addition to the banks themselves, Jazelle IO records contain an application defined CHARACTER*8 record type and two INTEGER*4 parameters to identify the information within the record. On input, all banks from the IO record are linked into the Jazelle bank structure, and can be accessed in the normal way. For files containing different types of records (e.g. event, begin/end run, constants), the record type can be used by the application routines to take the action appropriate to that record.

9.2 Usage

Files or tapes are opened through the routine JZIOPN , in which the device and disposition parameters are input. A device identifier is returned, which is used as input to all other Jazelle IO routines to identify the device. The first argument to the JZIOPN routine specifies the device/file which is to be opened. On the IBM this may either be a DDNAME (assigned using a FILEDEF command) or an explicit filename (e.g. MY DST A). The default filetype is JAZZDATA and the default filemode is '*'.

Note: **There is a temporary restriction that the DDNAME on VM must be of the form TAPEnn where nn are digits.**

On the VAX the argument can either be a logical name or any valid VMS file specification. Again the default filetype is .JAZZDATA.

Records are output through the routines JZIOWR or JZIOWC. Input arguments consist of a device identifier, the Jazelle list and ID specifying the banks or families to be output (JZIOWR) or the context to be written (JZIOWC), and the application defined record type and its two parameters.

Records can be input through the routine JZIORD with the device identifier as input, and with the record type and its parameters returned as output arguments. If the record was written using JZIOWC, the specified context will first be wiped (i.e. all existing banks in that context will be deleted) and then the new record will be read in to memory. If the record was written using JZIOWR individual banks will be read in and if the bank already exists the old version will be deleted.

Finally the routine JZIOCL allows a device to be closed. If JZIOCL is not called the device will be closed when the program terminates.

9.3 Machine Independence of IO

When JAZELLE writes data out it always does so in the native format of the host operating system. This allows the files to be read in again on the same machine very efficiently since no conversion to machine independent format is required. When a file is opened for reading JAZELLE checks the file header to see what type of machine the file was written on. If necessary IO conversion is enabled for the file and each record is converted to the host format on reading. This is totally transparent to the user, apart from the fact that the conversion operation takes some time.

A separate command, JIO, described in Appendix B allows a complete file to be converted from one format to another to avoid the overhead of conversion each time the file is read.

In order for files to be readable when written on one machine and read on another one it is important that the method used to transfer the file does not itself attempt to do any conversion of the file, including no attempt to perform byte-swapping. The following methods can be used to transfer files from the VAX to the IBM and vice-versa at SLAC.

- **BITNET** (subject to size constraints).

```
            VM                                     VMS
            --                                     ---
SENDFILE fn ft TO xyz AT abc       -->     RECEIVE/BINARY fn.ft

RECEIVE                            <--     SEND/FILE/BINARY fn.ft xyz@abc
```

- **SLACNET** (recommended for large files).

```
            VM                                     VMS
            --                                     ---
SLACNET EXPORT fn ft TO            -->
  [xyz]fn.ft AT abc ( AUTH ?
  INRECFM VARIABLE BINARY 1

RECEIVE                            <--     TRANSFER/BINARY=1 fn.ft/VARIABLE
                                             abc#[xyz.RDR]fn.ft
```

- **DUCS**

```
            VM                                     VMS
            --                                     ---
TODUCS fn ft ( BINARY                      TODUCS/BINARY fn.ft
```

9.4 Pointer Relocation

Pointers inside banks will be relocated during IO (i.e. will continue to point to the correct place after IO) so long as both the bank containing the pointer, and the bank pointed to are output as part of the same record. Pointers pointing to banks which are not part of the same record will be zeroed.

9.5 Example

In order to create a file of events the following code could be used:

```
$CALL JZIOPN('MYFILE','WRITE',' ',DID) ERROR RETURN; "Open file"
DO I=1,100 [
  "Generate event here"
  $CALL JZIOWC(DID,'EVENT',RNAME,PARAM) ERROR RETURN; "Write record"
  ]
$CALL JZIOCL(DID) ERROR RETURN; "Close file"
```

To read the same file the following code could be used:

```
$CALL JZIOPN('MYFILE','READ',' ',DID) ERROR RETURN; "Open file"
LOOP [
  $CALL JZIORD(DID,RNAME,PARAM) ERROR RETURN;  "Read event"
  IF $SEVERITY>$SUCCESS [ EXIT; ]              "Check for end-of-file"
  "Process event here"
  ]
$CALL JZIOCL(DID) ERROR RETURN; "Close file"
```

9.6 Implementation Details

Jazelle supports two types of output records, context IO and list IO, created by JZIOWC and JZIOWR respectively. Context IO allows only a single context to be written per record, while list IO allows any arbitrary combination of banks to be written out.

Context IO is optimized to minimize the CPU time needed for reading records, at the expense of some extra overhead in terms of size of the output file and time taken to write records. List IO is optimized to minimize the size of the file created, but requires slightly more cpu time to read a record.

The action JAZELLE takes on reading a record depends on the type of record and the **option** parameter specified on the JZIORD routine. If the record was written using the JZIOWC routine then before reading the record the specified context is cleared (wiped) and the new banks are read in. In this case the **option** argument to JZIORD is ignored.

If the record was written using the JZIOWR routine then the logic depends on the option specified to JZIORD as follows:

- ADD - Banks are read from the input record and placed into memory with the same ID that they had when they were written. If the bank already exists in memory an error occurs (this is the default action).
- REPLACE - Banks are read from the input record and placed into memory with the same ID that they were written with. If the bank already exists in memory it is replaced.
- APPEND - Banks are read from the input record and added to the end of their family. If there are no banks existing in that family the first bank is given Id 1. When this option is specified banks will not necessarily be created with the same Id that they had when they were written. Despite the ID of the banks changing any pointers between the banks will continue to point to the correct banks.

Note that if an entire family was written out, then on reading the entire family must be empty (ADD) or will be replaced (REPLACE).

10 Indexed Input/Output

Write-up not yet available.

11 Relational Tables

11.1 Introduction

Unlike most similar memory management systems JAZELLE supports the concept of a relational table, modeled on ideas taken from relational database architecture. An example of the use of a relational table would be to link together tracks and vertices in an event. There may be any number of TRACK banks each representing a track found in the detector, and in addition a number of VERTEX banks representing vertices. Some means is required of specifying which tracks are attached to which vertices, taking into account that, due to ambiguities in reconstruction, tracks may be associated with more than one vertex.

One way of generating this mapping is by use of a relational table. In JAZELLE a relational table is a family of banks, each of which links one TRACK bank with one VERTEX bank. A possible template for this track/vertex table is shown below.

```
Bank Trk_Vtx    Context=Event

  KEY      Track -->Track   "The track associated with this table entry"
  KEY      Vertex-->Vertex  "The vertex associated with this table entry"
!
! The rest of the bank would typically contain information relevant only
! in the context of a track/vertex association.
!
  REAL     P4VEC   "Track four-vector at this vertex"
  REAL     CHI2    "Chi2 of track fit to this vertex"

Endbank
```

The only thing which distinguishes a table entry from a normal bank is the presence of one or more elements of type KEY. In the example above there are two keys, one for the TRACK and one for the VERTEX.

11.2 Assigning Values to Keys

From a user point-of-view keys are very similar to pointers, but from a system point of view they are different in that the system maintains much stricter control over keys than it does over pointers. One consequence of this is that values should NEVER be directly assigned to keys by users. E.g. the following statement would be ILLEGAL:

```
TRK_VTX%(TRACK)=TRACK_POINTER; " THIS IS ILLEGAL!!! "
```

Values must be assigned to keys when the bank containing the keys is created This is achieved using the JZBADD routine as demonstrated in the following example:

```
$CALL JZBADD('TRK_VTX',          "Name of bank being created        "
             TRK_VTX_POINTER,    "Pointer to created bank           "
             TRK_VTX_ID,         "Id of created bank                "
             "DFLT",             "Id to be assigned to created bank "
             "DFLT",             "Repeat count for created bank     "
             TRACK_POINTER,      "Value to be assigned to first key "
             VERTEX_POINTER );   "Value to be assigned to second key"
```

The number of key values specified must correspond to the number of keys in the bank (2 in this example). The values of the keys specify which banks this table entry is associated with, in this example which TRACK bank and which VERTEX bank.

Values assigned to keys must never be changed in an assignment statement. The only legal way to change the value of a key after the bank containing it has been created is by the use of the JZTMOD routine. See Chapter 14 for a description of this routine.

11.3 Using Keys

In the simplest case KEYS can be used in exactly the same way as pointers, that is to get from a TRK_VTX bank to the corresponding TRACK bank all that is necessary is to write:

```
TRACK_POINTER =TRACK_VERTEX_POINTER%(TRACK);
VERTEX_POINTER=TRACK_VERTEX_POINTER%(VERTEX);
```

A more complicated example is to loop over all the TRACK banks associated with a particular VERTEX bank. The routine JZTSCN has been provided for this purpose. It searches a table for all occurrences of a given KEY with a particular value and returns a list of all the matches found. An example of the use of JZTSCN is given below:

```
$PARAMETER MAXTRACK=150;      " Maximum number of tracks at vertex "

POINTER MATCH_ARRAY(MAXTRACK)-->TRK_VTX;
POINTER VERTEX-->VERTEX;
POINTER TRACK -->TRACK;
POINTER TV_TABLE;

. . .

$Call JZBLOC('TRK_VTX',TV_TABLE) Error return;

$Call JZTSCN(TV_TABLE,        " Table to be scanned              "
            'VERTEX',         " Name of KEY to be scanned        "
            VERTEX,           " Pointer to vertex of interest    "
            MAXTRACK,         " Size of array to receive matches "
            MATCH_ARRAY       " Said array                       "
            MATCHES_FOUND     " Number of matches found          "
           ) Error return;

DO I=1,MATCHES_FOUND [ "Loop over tracks at vertex"

  TRACK=MATCH_ARRAY(I)%(TRACK); "Get pointer to track"

  PX=PX+TRACK%(Px);
  PY=PY+TRACK%(Py); " Do whatever needs to be done inside the loop "
  PZ=PZ+TRACK%(Pz);
]
```

The details of the arguments for JZTSCN are given in the final chapter of this manual. While JZTSCN is relatively easy to use it suffers from two disadvantages; a fixed size array is needed to receive the results which may not always be sufficiently large (in which case JZTSCN returns an error condition), and there is some overhead in calling JZTSCN. In most applications these limitations will probably be unimportant, but for other applications an alternative method of scanning tables is described in the next section.

11.4 Effects of KEYS during bank deletion

When a bank is deleted a search is made for any KEYS in other banks which point to that bank. If any such KEYS are found then the banks which contain those KEYS are also deleted. The search then continues for any banks containing KEYS pointing to the newly deleted bank and these banks are deleted etc.

11.5 Using Relational Tables without JZTSCN

In addition to the value of each KEY, JAZELLE maintains two additional pieces of information for each KEY, namely:

- **SAME** - A pointer to the next entry in the table which has the same KEY value.
- **FIRST** - A pointer to the next entry in the table which has a different KEY value.

Thus to find all of the tracks corresponding to a particular table it is necessary to follow three steps:

- Locate the beginning of the track vertex table.
- Find the first entry corresponding to the vertex of interest.
- Loop over all entries corresponding to the same vertex.

The following example demonstrates how to search the example TRK_VTX table to find all tracks at a given vertex without suffering from the limitations implicit in the use of JZTSCN.

```
" Find all tracks associated with vertex VERTEX   "
" First find the first entry in the TRK_VTX table "

$Call JZBFND('TRK_VTX',FIRST,IDOUT,'FRST') Error return;
If FIRST=0 [ $Return; ]

" Find the first entry associated with VERTEX "

TRK_VTX = FIRST;
Until TRK_VTX%(VERTEX)=VERTEX

  [ TRK_VTX = TRK_VTX%(VERTEX,FIRST);
    IF TRK_VTX=FIRST [ $Return; ] "No tracks found"
  ]

" Now loop over all the tracks associated with this vertex "

Loop [ TRACK = TRK_VTX%(TRACK);

       PX = PX+TRACK%(Px);
       PY = PY+TRACK%(Py); " Do whatever needs to be done"
       PZ = PZ+TRACK%(Pz);

       TRK_VTX=TRK_VTX%(VERTEX,SAME);

     ] Until TRK_VTX=0;
```

Since the above code is somewhat difficult to read a special statement, TABLELOOP, has been introduced into SLD MORTRAN. Using TABLELOOP the above example may be rewritten.

```
IF FIRST [ $CALL JZBLOC('TRK_VTX',TRK_VTX0) ERROR RETURN; ]

TABLELOOP TRK_VTX0,VERTEX=VERTEX,TRK_VTX [

       TRACK = TRK_VTX%(TRACK);

       PX = PX+TRACK%(Px);
       PY = PY+TRACK%(Py); " Do whatever needs to be done"
       PZ = PZ+TRACK%(Pz);

       TRK_VTX=TRK_VTX%(VERTEX,SAME);

     ]
```

EXIT and NEXT statements may be freely used within the TABLELOOP construct. See the SLD MORTRAN manual for a complete description of the TABLELOOP statement.

12 Constant Management System

12.1 Introduction

The JAZELLE constants management system is designed to handle the wide range of constants that will be generated as part of the offline analysis package. Specific aims of the system are to provide a uniform method of storing all constants, to handle run dependent constants as transparently as possible, and to avoid wasting time loading in constants which will not be used for a particular job.

So far only a subset of the final capabilities of the system have been implemented, in particular there is so far no handling of run dependent constants and no central database in which data is stored. The system has been designed however so that these omissions can be rectified in the future without the need to change existing code.

In an earlier chapter the initialization of elements inside TEMPLATEs was described, using a construct similar to the FORTRAN data statement. The constant system allows elements to be initialized in a similar, although much more powerful, manner using a file known as a CONSTANT file. An example of a typical constant file is given below.

```
!
!     OWNER: Dubois, Richard        CO-OWNER: Waite, A.P.
!   SECTION: LAC                        FILE: LGTMED TEMPLATE
!

Bank LGTMED(1)

    XVAR  / 3.1, 4.1, 5.9,        ! Assign values to XVAR
            2.6, 5.3, 5.8 /       !

    FRED(1).I /3/
    FRED(2).I /4/                 ! This initializes elements inside
    FRED(3).I /4/                 ! different occurrences of block FRED

    BLOCK JUNK(3)

      NAME /"Abcdefg"/            ! This refers initializes elements
                                  ! inside block JUNK(3)
    ENDBLOCK

Endbank

Bank LGTMED(2)

   XVAR / 6*0.0 /

Endbank
```

This example should be reasonably self explanatory. A few points that are worth further mention are:

- Comments delimited by ! can appear anywhere within a constant file.
- Constant files must begin with a BANK statement, this defines which bank is to be initialized. An explicit ID may also be specified.

- Variables are initialized using a method again very similar to the FORTRAN data statement. No explicit continuation character is needed, each line continues until the closing /.
- The values specified must correspond to the type and dimension of the element as declared in the corresponding template.
- Elements inside sub-blocks can be initialized either by explicitly giving the sub-block name on each line (eg FRED(2).I) or by using the BLOCK, ENDBLOCK construct shown.
- A constant file may initialize any number of banks. The banks do not all have to belong to one family.

An additional construct available within a constant file is the TABLE construct. This is very useful when a large number of elements inside a sub-block have to be initialized. The following example illustrates the use of the TABLE construct.

```
Bank PMDSIMP

 Table CHAMBER(0:4) DOMNAME  ABSZMIN  ABSZMAX   RHOMIN   RHOMAX
                   /'VTX'      0.00     5.00      3.1      3.9 /
                   /'VTX'      0.00     5.00      3.9      4.5 /
                   /'DRFT'     0.00   100.00     20.0    100.0 /
                   /'DEND'   100.00   124.80     20.0    100.0 /
                   /'DEND'   196.50   218.00     20.0    176.0 /
 Endtable

 Table CHAMBER(0:4) PTMIN    ARCCOMP  ARCMIN   ARCMAX
                   /0.000    TOTAL     0.0     20.0    /
                   /0.000    TOTAL     0.0     20.0    /
                   /0.050    TRANS    20.0     76.0    /
                   /0.000    LONGI    18.0     20.0    /
                   /0.000    LONGI    18.0     20.0    /
 Endtable
Endbank
```

The first argument to the TABLE statement is the name of the sub-block within which elements are to be initialized. The range of the sub-block index for which initialization is to be performed must be specified explicitly. The remaining elements are the names of elements within the sub-block to be initialized. This is followed by the values to be assigned to these elements in the natural order. The list of initial values is terminated by the ENDTABLE statement.

12.2 Usage

There are currently only a few routines which comprise the SLD constants system. These routines are:

- JZKGET - Read a CONSTANT file
- JZKLNK - Link a routine to a bank
- JZKUPD - Call linked routines

Of these routines JZKGET will be most used. This routine is discussed first below.

12.2.1 Reading CONSTANT Files

The JZKGET routine is used to read a constant file and initialize the associated banks. The calling sequence for JZKGET is as follows:

```
$CALL JZKGET(filename,options,ptr,routine) ERROR RETURN;
```

where the arguments have the following meanings:

- **filename** is the name of the constants file to be read. The file must have filetype CONSTANT. As JZKGET reads the constant file it creates each bank that is specified and then initializes any elements specified. If an explicit ID is given on the bank statement the created bank is given that ID. If no bank Id is given then the new bank is added to the end of the family, or given Id 1 if no banks yet exist in that family.
- **options** is a string specifying options for the JZKGET routine. At present this string can have only two legal values, namely 'RELOAD' or ' ' (blank). If blank is specified then an attempt to initialize a bank which already exists will be ignored. This is so that JZKGET can be called multiple times for the same bank with no adverse effect. This feature is useful where constants are used in two or more places, for example geometry constants might be used in the MC and in the analysis. JZKGET can be called in both places to make sure the constants are always available when needed with no danger of interference if both MC and analysis routines are used in the same job.

  If the RELOAD option is specified it will cause constants to be loaded into banks even if that bank already exists.
- **ptr** is set to point to the bank created. If more than one bank is created by reading the constants file then the pointer will point to the last one created. This argument is optional.
- **routine** is a routine to be linked to this bank. Linking routines to banks is discussed in the next section. This argument is optional.

Note: **The use of the routine argument to JZKGET is not recommended since it can, under certain circumstances, lead to problems with recursive calls to JZKGET. A better method is to link the routine to the banks by using the JZKLNK routine described later.**

JZKGET should be called in each routine, or at the head of each set of routines in which the associated constants are required. The advantage of calling JZKGET here, rather than in some global initialization routine, is that the constants will then only be loaded if the routines which use them are actually used in a particular job. This avoids wasting time managing constants which may not actually be needed. When analysing compressed DST's which may contain events from many different runs this can save appreciable time as the constants may be changing rapidly.

Calling JZKGET more than once will not cause any problems, as discussed above, but since it still takes some time the call to JZKGET in any routine should only be made the first time that routine is called. E.g.

```
IF FIRST [ FIRST=.FALSE.; $CALL JZKGET('MYFILE',' ') ERROR RETURN; ]
```

Once JZKGET is called the constant system will take care of updating banks when, for instance, the run number changes (although this is not yet implemented). Because there may be other constants derived from these constants some mechanism is needed to enable these derived constants to be recalculated when the base constants change. For this purpose it is possible to link routines to banks, and this is the subject of the next section of this chapter.

12.2.2 Linking Routines to Banks

Consider the case where the position of each wire of some chamber is to be stored in an array. Since there may be many wires it may be useful to have the position of the chamber specified in some constant file and then have a routine which calculates the position of the wires based on the chamber position. At some time during a job the run number may change and a new set of chamber coordinates may be loaded into the chamber position bank. At this point it is clearly necessary that the routine to calculate wire positions is called again to calculate the new wire positions. For this reason it is possible to link routines to banks so that the routine will be called whenever the values stored in the bank are changed.

Routines can be linked to banks in one of two ways, by specifying the routine as the forth argument in the call to JZKGET or by using the JZKLNK routine. To link the routine mentioned above to the chamber geometry bank the following call could be used.

```
$CALL JZKLNK(WIRPOS,'GEOMBANK') ERROR RETURN;
```

where GEOMBANK is, for example, a chamber geometry bank and WIRPOS is the routine which calculates the wire positions. Once a routine has been linked to a bank it will be called whenever the associated bank is changed by new constants being read in (INCLUDING the first time constants are read in) or whenever a JAZELLE interactive command is used to modify the bank from the debugger or from IDA.

Continuing to consider the example discussed above, the chamber positions in GEOMBANK may in turn be specified relative to an overall detector position bank. In this case the wire positions may have to be recalculated whenever either the chamber geometry bank changes or the overall position bank changes. To handle this and similar cases a routine can be linked to several banks and will be called whenever any of these banks change. E.g.

```
$CALL JZKLNK(WIRPOS,'GEOMBANK') ERROR RETURN;
$CALL JZKLNK(WIRPOS,'OVERALL' ) ERROR RETURN;
```

The last subtlety that needs to be mentioned is that the routines are not called immediately after the banks are updated. In fact when a bank is changed JAZELLE merely marks the associated routine to be called, but it is not actually called until the routine JZKUPD is called. JZKUPD is not normally called by the user but would typically be called by the main analysis program (IDA) before entering the event loop, and after each run number change.

As always, complete details on the routines described in this chapter are given in Chapter 14.

13 Debugging Aids with JAZELLE

JAZELLE includes a number of debugging aids some of which are described in this chapter. The debugging aids fall into two categories, routines which can be called to output useful information, and interactive commands that can be used to examine JAZELLE banks. These two categories are described below.

13.1 Useful Routines

The following routines output useful information concerning JAZELLE.

- JZINDX - Produce an index of all existing banks
- JZSTAT - Produce statistics on JAZELLE memory usage
- JZPDMP - Produce a list of all registered pointers
- JZMAP - Produce a complete map of JAZELLE memory
- JZPM - Produce a JAZELLE post mortem

Detailed descriptions of all of these routines can be found in Chapter 14.

As an additional debugging aid JAZELLE automatically traps all fatal JAZELLE or GETVM errors as well as access violations (VMS) and OC4/5's (VM) and produces a post mortem output.

13.2 Interactive commands

Jazelle supports a set of interactive commands which can be used for debugging purposes. The same set of commands is available both from the VAX debugger and within IDA. These commands are summarized below:

- PEEK - Examine a bank or an element of a bank
- POKE - Modify an element of a bank
- ADD - Create a new bank
- REMOVE - Delete a bank
- GET - Read a constant file
- STATUS - Output summary of JAZELLE memory usage
- INDEX - Produce an index of existing banks
- MAP - Produce a complete map of JAZELLE memory
- PM - Produce a JAZELLE postmortem
- POINTER - Examine a user pointer

Each of these commands is described in more detail below.

PEEK

The peek command can be used to examine the contents of a whole family of banks, of one particular bank, an element of one bank, or a sub-block of one bank. The PEEK command can also be used to examine items from bank headers or parameters defined in banks. The format of the output will depend on what is requested.

FORMAT

PEEK *Item*

Command Qualifiers	**Defaults**
/LEVEL=n	*0*

PARAMETERS

Item

Specifies the bank, family or element to be dumped.

Specifying JUNK(*) would output all banks of family JUNK; JUNK(4) would output bank JUNK id 4 and JUNK(3).FRED would output element FRED of bank JUNK id 3 Fred may be either a single element of a sub-block of family junk.

QUALIFIERS

/LEVEL

Selects the level of detail required,

A value of 0 (the default) provides brief output, 1 medium and 2 verbose.

EXAMPLES

1
```
PEEK MCTRACK(*)
```

The PEEK command causes all banks of family MCTRACK to be output using level 0 (brief) format.

2
```
PEEK MCTRACK.FRED.X.Y.Z
```

Element FRED.X.Y.Z of the last MCTRACK bank is output.

3
```
PEEK/LEVEL=2 FRED(4).H(3)
```

Sub-block or element H(3) of bank FRED id 4 is output using level 2 (verbose) format.

4
```
PEEK MCTRACK.JB$VBALO
```

The element JB$VBALO from the header of the last MCTRACK bank is output.

POKE

The poke command is used to modify elements within banks. Values can be assigned to any element.

FORMAT

POKE *Element=Value*

Command Qualifiers	**Defaults**
None.	*None.*

PARAMETERS

Element

Specifies the element to be modified.

A single element must be specified. No wildcards are allowed.

Value

Specifies the value to be assigned to element.

The syntax of value depends on the type of element being modified. For a list of valid value specification see the section on initial values in templates.

EXAMPLES

1

```
POKE FRED.INTEGER=10
```

Assign an integer value to element INTEGER of the last FRED bank.

2

```
POKE FRED.BLOCK.REAL=56.8
```

Assign a real value to element X.Y.Z of bank FRED id 4. Element FRED(4).X.Y.Z of the last MCTRACK bank is output.

3

```
POKE FRED.ARRAY(4)=8
```

Assign an integer value to element 4 of array ARRAY in the last FRED bank.

4

```
POKE FRED(FIRST).STRING="Abcdefghijkl"
```

Assign a STRING value to element STRING of the first FRED bank. The element JB$VBALO from the header of the last MCTRACK bank is output.

5

```
POKE FRED.POINTER=MCTRACK(4)
```

Assign a pointer value to a pointer in the last FRED bank. The value of the pointer is evaluated when the POKE command is executed so the target bank must exist at that point (or the value will be stored).

ADD

The ADD command is used to create a new bank.

FORMAT

ADD *Bank*

Command Qualifiers	**Defaults**
None.	*None.*

PARAMETERS

Bank

Specifies the bank to be created.

Bank must specify a family name and optionally an ID. If no id is specified a new bank is added to the end of the specified family.

EXAMPLES

1
```
ADD FRED(25)
```

Create bank FRED with ID 25. If the specified bank already exists an error message will be generated.

2
```
ADD FRED
```

Add an new member of family FRED with an Id one greater than the highest existing ID. If no banks already exist in family FRED the new bank will have ID 1.

REMOVE

The remove command is used to delete a bank or a family of banks.

FORMAT

REMOVE *Bank*

Command Qualifiers	**Defaults**
None.	*None.*

PARAMETERS

Bank

Specifies the bank or family to be deleted.

Bank must specify a family name. If an explicit ID is specified that bank is deleted, otherwise the entire family is deleted.

EXAMPLES

1
```
REMOVE FRED(25)
```

Delete bank FRED with ID 25. If the specified bank does not exist no error message will be generated.

2
```
REMOVE FRED
```

Delete all members of family FRED. If no banks already exist in family FRED no error message will be generated.

GET

The get command causes a constant file to be read (c.f. JZKGET). Any banks referred to in the constant file will be created and initialized.

FORMAT

GET *Name*

Command Qualifiers	**Defaults**
None.	*None.*

PARAMETERS

Name

Specifies the name of the constant file to be read.

The file named must exist in the normal JAZELLE search order and have filetype CONSTANT.

EXAMPLES

1
```
GET JUNKFILE
```

JAZELLE will search for and read the file JUNKFILE.CONSTANT. If the file can not be found an error message will be issued.

STATUS

The status command outputs statistics on JAZELLE memory usage (c.f. JZSTAT).

FORMAT

STATUS

Command Qualifiers	**Defaults**
/LEVEL=n	*0*

QUALIFIERS

/LEVEL

Selects the level of detail required,

A value of 0 (the default) provides brief output, 1 medium and 2 verbose.

EXAMPLES

1

```
STATUS
```

Produce a brief summary of JAZELLE memory usage.

2

```
STATUS/LEVEL=2
```

Produce a detailed breakdown of JAZELLE memory usage.

INDEX

The index command can be used to produce an index JAZELLE banks. All the banks in one family may be selected, or all of the banks known to JAZELLE (c.f. JZINDX).

FORMAT

INDEX *[family]*

Command Qualifiers	**Defaults**
/LEVEL=n	*0*

PARAMETERS

Name

Optionally specified a family about which information is required.

If family is specified only banks in that family are listed, otherwise all banks will be included in the index.

QUALIFIERS

/LEVEL

Selects the level of detail required,

A value of 0 (the default) provides brief output, 1 medium and 2 verbose.

EXAMPLES

1
```
INDEX
```

Produce a brief index of all existing families.

2
```
INDEX/LEVEL=2
```

Produce a detailed index of all JAZELLE banks.

3
```
INDEX MCTRACK
```

Produce a detailed index of banks within family MCTRACK.

MAP

The map command produces a complete map of JAZELLE memory usage (c.f. JZMAP).

FORMAT

MAP

Command Qualifiers	**Defaults**
/LEVEL=n	*0*

QUALIFIERS

/LEVEL

Selects the level of detail required.

A value of 0 (the default) provides brief output, 1 medium and 2 verbose.

EXAMPLES

1
```
MAP
```

Produce a brief map of JAZELLE memory usage.

2
```
MAP/LEVEL=2
```

Produce a detailed map of JAZELLE memory usage.

PM

The PM command generates a JAZELLE post mortem (c.f. JZPM).

FORMAT

PM

Command Qualifiers	Defaults
/LEVEL=n	*0*

QUALIFIERS

/LEVEL

Selects the level of detail required.

A value of 0 (the default) provides brief output, 1 medium and 2 verbose.

EXAMPLES

1
```
PM
```

Produce a brief post-mortem dump.

2
```
MAP/LEVEL=2
```

Produce a detailed post-mortem dump.

POINTER

The POINTER command is only available inside the VAX debugger, It allows the value of any pointer to be examined (c.f. the debugger EXAMINE command).

FORMAT

POINTER *Variable-list*

PARAMETERS

Variable-list
Specifies one or more variables, using the normal debugger syntax.

EXAMPLES

1

```
POINTER MYPTR
```

Shows the value of pointer MYPTR, which must be a variable in the current debugger scope.

2

```
POINTER PTRA,PTRB
```

Shows the values of pointers PTRA and PTRB, which must be variables in the current debugger scope.

3

```
POINTER MCLUND\MCHITS
```

Shows the value of pointer MCHITS in routine MCLUND.

13.3 Use with VAX Debugger

To use the JAZELLE interactive commands with the VAX debugger it is necessary to ensure that the JAZELLE commands are known to the debugger. The method of doing this depends whether or not you have your own DBG$INIT file or not. If not you have to issue the command:

```
$ DEFINE DBG$INIT DUCSJAZELLE:JAZELLE.DBGCOM
```

before entering the debugger. This command could well be included into your LOGIN.COM file. If you do have your own DBG$INIT file you should include the line:

```
$ @DUCSJAZELLE:JAZELLE.DBGCOM
```

somewhere in that file.

Inside the debugger JAZELLE interactive commands can be typed directly at the debug prompt (which is changed to jDBG to remind you). Note that JAZELLE commands cannot be issued until after JZSTRT has been called by the program being debugged.

13.4 Use with IDA

See the IDA manual for details of using JAZELLE commands from IDA. The syntax of the commands may vary slightly from that given above.

14 Detailed Description of JAZELLE Routines

14.1 Summary of User Routines

The following is a complete list of all of the JAZELLE user routines, with a brief one line summary of their function. A more complete description of each one can be found in the following pages.

14.1.1 Initialization

- JZSTRT - Initialize JAZELLE

14.1.2 Bank Creation and Manipulation

- JZBADD - Create (add) a new bank
- JZBDEL - Delete a bank by name/id
- JZPDEL - Delete a bank by pointer
- JZBEXP - Expand/Contract a bank by ID
- JZPEXP - Expand/Contract a bank by pointer
- JZBFND - Find a pointer to an existing bank
- JZBLOC - Locate a complete family of banks
- JZBNXT - Find the next bank of a given family
- JZBCPY - Create a new bank which is a copy of an existing bank
- JZPCPY - Copy the contents of a bank to another existing bank

14.1.3 Producing Readable Dumps of Banks

- JZBDMI - Dump a bank or family of banks (by NAME/ID)
- JZBDMP - Dump a bank (by pointer)
- JZBTBL - Output banks in tabular form
- JZTDEF - Add a column to a table or create a new table
- JZTDFL - Make a table the default table for a family

14.1.4 Constant Manipulation

- JZKGET - Read a CONSTANT file (initialize banks)
- JZKLNK - Link an update routine to a bank
- JZKUPD - Call update routines for changed banks
- JZKPOK - Mark a bank as changed

14.1.5 List Manipulation

- JZLCRE - Create a list
- JZLINC - Add an entry to a list
- JZLREM - Remove an entry from a list
- JZLWIP - Delete all of the banks pointer to by a list

14.1.6 Input Output

- JZIOPN - Open a file for JAZELLE io
- JZIOCL - Close a file
- JZIOWR - Write a record using a list
- JZIOWC - Write a record using a context
- JZIORD - Read a record

14.1.7 Pointer Handling

- JZPREG - Register a user pointer
- JZPDMP - Produce a list of all registered user pointers
- JZPIDX - Register an indexed pointer array

14.1.8 Context Manipulation Routines

- JZXWIP - Delete entire context

14.1.9 STRING/CHARACTER Conversion Routines

- JZC - Convert a JAZELLE STRING to a character variable
- JZC4 - Convert a JAZELLE STRING*4 to a character variable
- JZC8 - Convert a JAZELLE STRING*8 to a character variable
- JZS - Convert a character variable to a STRING variable

- JZS4 - Convert a character variable to a STRING*4 variable
- JZS8 - Convert a character variable to a STRING*8 variable

14.1.10 Miscellaneous

- JZFDEF - Attach input/output routines to a family or block
- JZTSCN - Scan a table for particular values of a given key
- JZINDX - Produce an index of currently defined banks
- JZMAP - Produce a complete JAZELLE memory map
- JZPM - Produce a JAZELLE postmortem
- JZSTAT - Produce a summary of JAZELLE statistics
- JZBLONG - Obtain the length of a bank
- JZBNAME - Obtain the name of a bank from a pointer
- JZBMAP - Map a common block to a bank
- JZPCMP - Compare two banks
- JZTMOD - Change the value of a KEY
- JZVERS - return information about the current version of JAZELLE

14.2 Details of User Routines

The following is an alphabetical list of all of the JAZELLE user routines, giving a complete description of all arguments, and a description of each routine.

JZBADD — Add Bank to family

This function adds a Jazelle bank to the specified family.

FORMAT **JZBADD** *FNAME [, UDATPTR [, UIDOUT [, UIDIN [, URPTCNT [, KEY1 [, KEY2 [, KEY3 [, KEY4 [, KEY5]]]]]]]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

FNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Family name of bank to be added.

UDATPTR

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only (optional)**
mechanism:	**by reference**

Pointer to data region of created bank, byte-relative to JAZELLE signpost variable /JAZELL/JZL$B(0). (= 0 if error.)

UIDOUT

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only (optional)**
mechanism:	**by reference**

Bank ID actually assigned. (0 < ID < 2**16). (= 0 if error.)

UIDIN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Bank ID requested or 'LAST' (default).

URPTCNT

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Repeat count of variable length block or 'DFLT'.

KEY1

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Keys (if any) associated with this bank.

KEY2

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Keys (if any) associated with this bank.

KEY3

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Keys (if any) associated with this bank.

KEY4

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Keys (if any) associated with this bank.

KEY5

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Keys (if any) associated with this bank.

DESCRIPTION

JZBADD allocates memory for a new bank and links it into to a family. If the family does not already exist in memory, then the template for the family is read from disk, and the family block is created. The number of variable blocks which are allocated is taken from argument RPTCNT, if present, or from the default value specified in the template. If a specific bank ID is requested, it will be linked into the family to maintain monotonically increasing ID. Otherwise it will be linked onto the end of the family with ID one greater than the last previously existing bank.

RETURN VALUES

SLD$NORMAL	(S) Successful completion
JZL$IDEXISTS	(E) Requested ID already exists
JZL$BADID	(F) Illegal ID requested
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZBCPY — Copy a Bank

This routine creates a new Jazelle bank and copies data to it from another bank.

FORMAT

JZBCPY *FROMPTR, UTOPTR [, UIDOUT [, UIDIN [, URPTCNT]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

FROMPTR

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Family name of bank to be added.

UTOPTR

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**write only**
mechanism:	**by reference**

Pointer to created bank.

UIDOUT

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only (optional)**
mechanism:	**by reference**

Bank id of the created bank.

UIDIN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Bank id for the new bank (requested by user).

URPTCNT

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Number of VRBs (variable repeat blocks) to be allocated in the new bank. By default, a bank will be created that is sufficiently large to hold the data found in the source bank.

DESCRIPTION

JZBCPY creates a bank and copies the contents of the bank pointed to by FROMPTR to the newly created bank. UTOPTR is set to point to the new bank and UIDOUT is set to the bank ID of the new bank.

RETURN VALUES

SLD$NORMAL	(S) Successful completion
JZL$NOCOPY	(E) Copy Failed
JZL$STATIC	(E) Copy Failed
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZBDEL — Delete a Bank

This routine will delete a bank or a series of banks currently in the data structure.

FORMAT

JZBDEL *FNAME [, ID]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

FNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Family name of JAZELLE bank(s) to be deleted.

ID

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

= 'ALL*' to delete all banks of family (default).

= Explicit ID to delete a particular bank.

RETURN VALUES

SLD$NORMAL	(S) if requested bank was successfully deleted.
JZL$NOBANK	(I) if requested bank was not found.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZBDMI — Formatted dump of named bank

Allows users to dump out a bank by name.

FORMAT **JZBDMI** *NAME [, ID [, LUN [, LEVEL [, WIDTH]]]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

NAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Family name of bank to be dumped.

ID

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

ID of bank to be dumped (Def 'ALL*').

LUN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

LUN to which bank should be dumped (def 6).

LEVEL

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Dump level (min=0,1,max=2) (def 0).

WIDTH

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Width of output (default 80).

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZBDMP — Formatted dump of bank - by pointer

This routine allows the caller to dump a bank referenced by the bank pointer.

FORMAT

JZBDMP *PTR [, LUN [, LEVEL [, WIDTH]]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

PTR

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

Pointer to bank to be dumped.

LUN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

LUN to which bank should be dumped (def 6).

LEVEL

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Dump level (def 0).

WIDTH

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Width of output (default 80).

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZBEXP — Expand a JAZELLE bank

This routine expands the specified bank without changing the bank contents.

FORMAT

JZBEXP *NAME, ID [, COUNTI [, PTRI]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

NAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Name of bank to be expanded.

ID

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

ID of bank to be expanded.

COUNTI

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

New allocation for number of variable blocks.

PTRI

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**write only (optional)**
mechanism:	**by reference**

Pointer to expanded bank.

DESCRIPTION

JZBEXP expands (or contracts) a bank by creating a new bank with different allocated variable block count (COUNTI), copying the contents of the old bank into the new, and deleting the old bank.

If COUNTI is omitted the bank is expanded to twice its current size.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZPEXP — Expand a JAZELLE bank

This routine expands the specified bank without changing the bank contents.

FORMAT

JZPEXP *PTRI, COUNTI*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

PTRI

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Pointer to bank to be expanded.

COUNTI

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

New allocation for number of variable blocks.

DESCRIPTION

JZPEXP expands (or contracts) a bank by creating a new bank with different allocated variable block count (COUNTI), copying the contents of the old bank into the new, and deleting the old bank.

If COUNTI is omitted the bank is expanded to twice its current size.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZBFND — Find pointer to named JAZELLE bank

This routine finds the pointer to the bank specifed by family name and (optionally) bank ID.

FORMAT

JZBFND *FNAME, DATPTR [, IDOUT [, IDIN]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

FNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Family name of requested bank.

DATPTR

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only**
mechanism:	**by reference**

Pointer to requested bank byte-relative to /JAZELL/JZL$B(0). (= 0 on error.)

IDOUT

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only (optional)**
mechanism:	**by reference**

ID of located bank (= 0 on error).

IDIN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

'FRST' to locate first bank of family (default).

'LAST' to locate last bank of family Explicit ID to locate bank with particular ID.

RETURN VALUES

SLD$NORMAL	if requested bank was found
JZL$NOBANK	if bank does not exist
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZBLOC — Locate a JAZELLE family

Locates the specified family returning the family pointer and bank count.

FORMAT

JZBLOC *FNAME, FAMPTR, NBANKS*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

FNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Requested family name.

FAMPTR

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**write only**
mechanism:	**by reference**

Returns a pointer to a jazelle family. If used as input to JZBNXT or as FAMPTR%(JB$FORPT) yields pointer to first bank.

Note: If family exists, but contains no banks, FAMPTR will be valid if banks are subsequently added to the family.

NBANKS

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only**
mechanism:	**by reference**

Number of banks in family (=0 if family does not exist) (optional).

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZBLONG — Length of a bank

This routine is a replacement for the version 2 header element JB$LONG. The routine returns the length (in bytes) of a bank including header.

FORMAT

JZBLONG *PTR*

RETURNS

VMS Usage:	**longword**
type:	**longword {INTEGER*4}**
access:	**write only**
mechanism:	**by value**

Value corresponding to the requested data as defined in the function overview.

ARGUMENTS

PTR

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Pointer to bank.

JZBMAP — Make Common block a bank

This routine makes the common block look like a newly added bank.

FORMAT

JZBMAP *TNAME, COMPTR, IOCHAR [, STCPTR]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

TNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Template name to be used.

COMPTR

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

Pointer to the common block.

IOCHAR

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER}**
access:	**read only**
mechanism:	**by descriptor**

Type of operation to be performed on the common block by the user program. R = read, W = write, B = both.

STCPTR

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only (optional)**

mechanism: **by reference**

Pointer to the static bank.

DESCRIPTION

JZBMAP makes the common block look like a new bank and links it into a family. If the family already exists (there can only be one static bank per family) and the same input pointer is given then an error does not occur and the indirect pointer is returned as if JZBMAP had actually done the mapping. A specific bank ID can not be requested, it will always be 1. NO variable blocks can be allocated (remember this is a STATIC bank).

This routine actually makes a copy of the common block and this copy is the static bank everyone will see. A table is kept that contains both a pointer to the copy and the address of the common block. This is so that any changes made to the common block or the static bank can be reflected in the other.

RETURN VALUES

SLD$NORMAL	(S) Successful completion.
JZL$BADARGC	(E) Invalid IOCHAR given.
JZL$ARGS	(E) Incorrect number of parameters passed.
JZL$NOTSTIC	(E) Tried to map a non static bank.
JZL$STCARG	(E) The static bank is already mapped to different address.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZBNAME — Get bank name from pointer

Returns the name of a bank given a pointer to the bank. This replaces JB$NAME.

FORMAT

JZBNAME *PTR*

RETURNS

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**write only**
mechanism:	**by value**

This function returns a character-coded text string

ARGUMENTS

PTR

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Pointer to bank.

JZBNXT — Find next bank of JAZELLE family

FORMAT

JZBNXT *PTRIN, PTROUT [, ID]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

PTRIN

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Pointer to a JAZELLE bank, byte relative to /JAZELL/JZL$B(0).

PTROUT

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**write only**
mechanism:	**by reference**

Pointer to next bank in same family (= 0 on error).

ID

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only (optional)**
mechanism:	**by reference**

ID of located bank (= 0 on error).

DESCRIPTION

JZBNXT spends some time checking the integrity of the data structure in applications where speed is at a premium, inside nested loops for example, an alternative method of finding the next bank may be prefered, for instance:

```
PTROUT=PTRIN%(JB$FORPT);
```

RETURN VALUES

SLD$NORMAL	(S) if requested bank was found
JZL$LASTBANK	(I) if INPUT bank is last bank in family
JZL$BADBANKP	(F) if corrupted pointers detected
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZBTBL — Formatted dump of named bank

Allows users to dump out a bank by name.

FORMAT **JZBTBL** *NAME [, ID [, TABLE [, LUN [, LEVEL [, WIDTH]]]]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

NAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Family name of bank to be dumped.

ID

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

ID of bank to be dumped (Def 'ALL*').

TABLE

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only (optional)**
mechanism:	**by reference**

Table to be used (0 or missing, default used).

LUN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**

mechanism:	**by reference**

LUN to which bank should be dumped (def 6).

LEVEL

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Dump level (min=0,1,max=2) (def 0).

WIDTH

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Width of output (default 80).

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZC — Convert string to character

This routine converts a Jazelle string (Hollerith) to a character-coded text string of length 1.

FORMAT

JZC *N, STRING*

RETURNS

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**write only**
mechanism:	**by value**

This function returns a character-coded text string

ARGUMENTS

N

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

Length of STRING in bytes.

STRING

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

JAZELLE string to be converted.

JZC4 — Convert 4-byte string to character

This routine converts a STRING*4 (hollerith) Jazelle data type to a character coded text string.

FORMAT

JZC4 *STRING*

RETURNS

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**write only**
mechanism:	**by value**

This function returns a character-coded text string

ARGUMENTS

STRING

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

JAZELLE (hollerith) string to be converted.

JZC8 — Convert 8-byte string to character

This routine converts a STRING*8 (hollerith) Jazelle data type to a character coded text string.

FORMAT

JZC8 *STRING*

RETURNS

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**write only**
mechanism:	**by value**

This function returns a character-coded text string

ARGUMENTS

STRING

VMS Usage:	**quadword**
type:	**quadword {REAL*8}**
access:	**read only**
mechanism:	**by reference**

JAZELLE (hollerith) string. The Fortran data type corresponds to a REAL*8.

JZFDEF — Attach routine to a family or block

Routine is used to define routines for dumping, inputing or outputing a block or family.

FORMAT **JZFDEF** *NAME, DUMP, PEEK, POKE [, ARG]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

NAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Name of block or family.

DUMP

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {EXTERNAL}**
access:	**read only**
mechanism:	**by reference**

Dump routine.

PEEK

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {EXTERNAL}**
access:	**read only**
mechanism:	**by reference**

Peek routine.

POKE

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {EXTERNAL}**
access:	**read only**

mechanism:	**by reference**

Poke routine.

ARG

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Argument passed to routines.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZINDX — list all families and banks

This routine dumps a list of all the families and banks in memory.

FORMAT

JZINDX *FAMILY [, LUN [, LEVEL [, WIDTH]]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

FAMILY

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Family to be dumped (Default ALL*).

LUN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

LUN to which bank should be dumped (default 6).

LEVEL

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Dump level (default 0).

WIDTH

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Width of output (default 80).

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZIOCL — Close Jazelle IO device

FORMAT

JZIOCL *DID*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

DID

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Device ID obtained from JZIOPN.

RETURN VALUES

SLD$NORMAL	(S) Successful completion
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZIOPN — Open IO device

This routine opens an IO device and sets up the header bank for the device.

FORMAT

JZIOPN *DDNAME, RW, DISP, DID [, RECL [, OLDDID]]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

DDNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Device name. On VM this is the file name or DDNAME of the device; On VMS it is the file name or logical name of the device.

Default filetype is JAZZDATA.

RESTRICTION: On VM DDNAME is restricted to be TAPEnn.

RW

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

'Read', 'Write', or both seperated by a comma. May be abbreviated to 'R' and 'W'. 'READ,WRITE' is only valid if DIRECT is specified in the DISP argument.

DISP

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

'DIRECT', 'DELETE', 'APPEND', and 'EXCLUS'. More than one of these values can be specified by seperating them by commas, eg 'DIRECT,DELETE'. Other values will be ignored for backward compatability. DIRECT must be present for APPEND, DELETE, or EXCLUS to be valid. APPEND is only valid when the file is being opened as read only. DELETE is only valid if the file is being created. EXCLUS is only valid if the file is not in use. EXCLUS is the default for new files and on IBM.

DID

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**modify**
mechanism:	**by reference**

Device ID to be used in subsequent calls to Jazelle IO routines for this device or if APPEND then this is an input pointer which must correspond to a direct access file already opened for read to which the file being opened is to be logically concatenated to.

RECL

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Record length of-file. For direct access files the default is 1024. For sequential files this is ignored if the file already exists.

OLDDID

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only (optional)**
mechanism:	**by reference**

Device ID to of another file. When a new file is created some of its header attributes (such as creation date and owner) will be copied from this devices header record. Useful when copying files.

RETURN VALUES

SLD$NORMAL	(S) Successful completion
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZIORD — Read record and link

This routine reads a Jazelle record from an IO device and links the banks into memory so they may be accessed using the normal methods.

FORMAT

JZIORD *DID, RNAME, PARAM [, OPTION]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

DID

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Device ID obtained from JZIOPN.

RNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**write only**
mechanism:	**by descriptor**

Record name (8 characters max) associated with record.

PARAM

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only**
mechanism:	**by reference**

Application defined parameters read from record header.

OPTION

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**

mechanism: **by reference**

See Description.

DESCRIPTION

All banks contained in record will be linked into Jazelle directory and can be accessed in normal fashion.

If record was written with JZIOWC then entire context will be replaced In this case the option argument is ignored.

If record was written with JZIOWR then action depends on OPTION.

Options:

ADD (Default) Read banks and add to existing banks. If bank already exists returns error condition.

REPLACE Read banks and add or replace existing banks.

APPEND Read banks and add to end of existing families. (ID of bank will not be the same as when written)

Note: **If an entire family of banks was written out then the entire family will be replaced (REPLACE) or an error will result if any banks exist in the family (ADD).**

RETURN VALUES

SLD$NORMAL	(S) Successful completion
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZIORW — Rewind Jazelle IO device

FORMAT

JZIORW *DID*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

DID

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Device ID obtained from JZIOPN.

RETURN VALUES

SLD$NORMAL	(S) Successful completion
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZIOWC — Write entire context to IO device

FORMAT

JZIOWC *DID, CNAME, RNAME, PARAM [, USERP]]*
]

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

DID

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Device ID obtained from JZIOPN.

CNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Name of context to be written.

RNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Record name (8 characters max) to be associated with record.

PARAM

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

Parameters to be written with record header.

USERP

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only (optional)**
mechanism:	**by reference**

User data to be written into the record header.

RETURN VALUES

SLD$NORMAL	(S) Successful completion
SLD$STATICON	(E) Can not write static banks.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZIOWR — Write Jazelle record to IO device

FORMAT **JZIOWR** *DID, LIST, LID, RNAME, PARAM [, USERP]]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

DID

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Device ID obtained from JZIOPN.

LIST

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Name of Jazelle List to be written.

LID

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

Bank ID of LIST.

RNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Record name (8 characters max) to be associated with record.

PARAM

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

Parameters to be written with record header.

USERP

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only (optional)**
mechanism:	**by reference**

User data to be written into record header.

RETURN VALUES

SLD$NORMAL	(S) Successful completion
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZIOWP — Write Jazelle record to IO device

FORMAT

JZIOWP *DID, LPTRX, RNAME, PARAM, USERP*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

DID

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Device ID obtained from JZIOPN.

LPTRX

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {UNKNOWN}**
access:	**read only**
mechanism:	**by reference**

Pointer to list to be written.

RNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Record name (8 characters max) to be associated with record.

PARAM

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

Parameters to be written with record header.

USERP

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only (optional)**
mechanism:	**by reference**

User data to be written into record header.

RETURN VALUES

SLD$NORMAL	(S) Successful completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZKGET — User interface to the JAZELLE constant system

Is used to load constants into a file and to inform the JAZELLE constants system that it should continue to process constants for the specified bank.

FORMAT

JZKGET *FILE, OPTION [, PTR [, ROUTNE]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

FILE

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Name of bank to be initialized.

OPTION

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Any options that need to be specified.

PTR

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**write only (optional)**
mechanism:	**by reference**

Pointer to bank created.

ROUTNE

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {EXTERNAL}**
access:	**read only (optional)**
mechanism:	**by reference**

Routine to be called when constants updated.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZKLNK — Routine to add a routine to the constant chain

Is used to define a routine which will be called each time a bank or set of banks is marked as changed either by the constants system or by a call to JZKPOK.

FORMAT

JZKLNK *ROUTNE, NAME [, ID]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

ROUTNE

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

Routine to be called.

NAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Name of bank to be added to chain.

ID

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

ID of bank to be added to chain (optional, default ALL).

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZKPOK — Routine to mark a bank as updated

Is called to mark a bank as having been updated. It will trigger any linked routines to be called nxet time JZKUPD is called.

FORMAT

JZKPOK *BANK*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

BANK

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Bank to be marked as updated.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZKUPD — Routine to update all banks in constant chain

Is called to call the initialization routines corresponding to any banks which have been updated since the last call to JZKUPD.

FORMAT

JZKUPD

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZLCRE — Create list bank

This routine creates a Jazelle symbolic or pointer list bank.

FORMAT

JZLCRE *NAME, CONTXT, KIND [, PTR [, IDOUT [, IDIN [, NPTRS]]]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

NAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Family name of list to be created.

CONTXT

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Context name in which list is to be created.

KIND

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

= 'SYMBOLIC' to create sympolic list;

= 'POINTER' to create pointer list;

(Only first character required).

PTR

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only (optional)**
mechanism:	**by reference**

Pointer to created list bank.

IDOUT

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only (optional)**
mechanism:	**by reference**

Actual ID of created list bank. (= 0 if error).

IDIN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Requested ID, or 'LAST' (default).

NPTRS

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Number of pointers for which space will be allocated.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$BADARGC	(E) Illegal specification of KIND.
JZL$BADKIND	(E) KIND does not match already existing list family
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZLINC — Include pointer in list

FORMAT **JZLINC** *LNAME, FNAME [, FID [, LID [, INDFLG]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

LNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Name of list to which pointer is to be added.

FNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Family name of bank(s) to be added.

FID

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Explicit ID of bank to be added, or 'ALL*' (default). If LNAME refers to a pointer list, and an explicit ID is used, the bank (FNAME, ID) must have been previously created. If LNAME refers to a pointer list and 'ALL*' is used, and the family FNAME has not been created, then the template bank and family block for that family will be created.

LID

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Explicit ID of list to which pointer is to be added, or 'FRST', or 'LAST' (default). The specified list bank must have been previously created.

INDFLG

VMS Usage:	**longword**
type:	**longword {LOGICAL}**
access:	**read only (optional)**
mechanism:	**by reference**

.TRUE. if FNAME is another list and should be used as another level of indirection; i.e. for list operations such as IO and wipe, the requested operation will be performed not on bank(s) (FNAME, FID), but on the banks specified therein. (default = .FALSE.).

DESCRIPTION

JZLINC adds a pointer to a JAZELLE symbolic or pointer list bank. If the list bank is full, a garbage collection is performed to squeeze out null entries left by previous calls to JZLREM. If this fails to create space, then the list bank is expanded. The new pointer is added as the last pointer in the list.

RETURN VALUES

JZL$NOLIST	(E) Requested list does not exist.
JZL$NOBANK1	(E) Requested bank does not exist.
JZL$STATIC	(E) Cannot include a static bank in a list.
JZL$BADALO	(F) Number of used pointers exceeds number allocated.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZLREM — Remove pointer from list

FORMAT

JZLREM *LNAME, FNAME [, FID [, LID]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

LNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Name of list from which pointer is to be removed.

FNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Family name of bank(s) to be removed.

FID

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Explicit ID of bank : Removes entry which was added to list with an explicit ID.

'ALL*': removes entry which was added to list with ID 'ALL*'.

'ANY*': (default) Removes any entry in the list belonging to the specified family.

LID

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**

mechanism: **by reference**

Explicit ID of list from which pointer is to be removed, or 'FRST' or 'LAST' (default).

RETURN VALUES

JZL$NOLIST	(E) Requested list bank does not exist.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZLWIP — Wipe (delete) banks in list

FORMAT

JZLWIP *LNAME [, LID]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

LNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Name of list.

LID

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Explicit ID of list, or 'FRST' or 'LAST' (default).

RETURN VALUES

JZL$NOLIST	(E) Requested list does not exist.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZMAP — Memory map of banks

Produces a memory map of all the JAZELLE banks.

FORMAT

JZMAP *[, LUN [, LEVEL [, WIDT]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

LUN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Logical unit number for output (Default 6).

LEVEL

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Output level (Default 0).

WIDTH

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Output width (Default 80).

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZPCMP — Compare Bank Contents

FORMAT **JZPCMP** *P1, P2, UOPTION [, UMATCH [, URPTCNT]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

P1

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Pointer to first bank.

P2

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Pointer to second bank.

UOPTION

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only**
mechanism:	**by reference**

Dergee of Match in the Data Area.

UMATCH

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only (optional)**
mechanism:	**by reference**

Match Flags.

URPTCNT

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Compare only the first RPTCNT variable blocks.

DESCRIPTION

JZPCMP compares the banks pointed to by p1 and p2.

It returns JZL$NORMAL or the error code JZL$MISMATCH, depending on whether the banks are equal with respect to the given criteria or not.

The following match options can be specified:

Options:

JZPCMP$SAME the pointers point to the same bank.

JZPCMP$FAMILY banks belong to the same family.

JZPCMP$TEMPLATE banks use the same template.

JZPCMP$COMPAT banks have compatible structures.

JZPCMP$DATA the data is identical (=JZPCMP$FIXDATA+VARDATA).

JZPCMP$FIXDATA the fixed part data is identical.

JZPCMP$VARDATA vrb data is identical (but one bank might be larger).

JZPCMP$SIZE same size.

JZPCMP$FIXSIZE fixed size parts have the same size.

JZPCMP$VARSIZE vrb's have the same size.

JZPCMP$VARALOC allocation counts for vrb's are identical.

JZPCMP$MATCH test for all of the above, return bits in MATCH.

The above are integer*4 values and the usual comparison operators can be used. The above table shows the values in order of size where JZPCMP$SAME is largest.

In the calling user routine, the above bit masks are used to

a) specify the type of comparison to be performed and

b) to interpret the result of the comparison (as returned in the optional parameter MATCH)

Users of these bitmask values need to include JZPCMP.PAR to obtain the definitions for these constants.

RETURN VALUES

SLD$NORMAL	(S) Banks match wrt the given criteria.
JZL$MISMATCH	(I) Banks dont match.
JZL$BADCMOPT	(E) Bad compare option specified.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZPCPY — Bank to Bank Copy

This routine copies the contents of one existing bank to another existing bank.

FORMAT

JZPCPY *FROMPTR, TOPTR*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

FROMPTR

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Pointer to bank holding the data to be copied (source).

TOPTR

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Pointer to bank to which the data is to be copied (target).

DESCRIPTION

JZPCPY copies the contents of one bank (FROMPTR–>) to another existing bank (TOPTR–>). Extra repeat blocks in the other bank will be initialized as specified in the template file.

If the other bank (TOPTR–>) is too small, the bank is expanded to match the size of the first bank.

RETURN VALUES

SLD$NORMAL	(S) Successful completion.
JZL$NOCOPY	(E) Failure.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZPDEL — Delete a bank

This routine deletes a Jazelle bank referenced by pointer.

FORMAT

JZPDEL *BANK*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

BANK

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Pointer to bank to be deleted.

RETURN VALUES

SLD$NORMAL	(S) if requested bank was successfully deleted.
JZL$NOBANK	(I) if requested bank was not found.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZPDMP — Dump user pointers

Is used to produce a dump of registered user pointers in one context or in all contexts. Flags may be specified to overide the default action.

FORMAT

JZPDMP *CONTXT [, FLAGS1 [, LUN [, LEVEL [, WIDTH]]]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

CONTXT

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Context to be dumped ('*' or ' ' dumps all).

FLAGS1

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Overide flags specified when pointers registered. The bits in FLAGS1 are defined in JZPFLG.PAR. (No useful bits are yet defined.)

LUN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Logical unit number for output (Default 6).

LEVEL

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Dump level (Default 0).

WIDTH

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Width of output (Default 80).

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZPIDX — Initialize JAZELLE indexed pointer array

FORMAT

JZPIDX *FNAME, PARRAY*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

FNAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Name of family to be associated with array.

PARRAY

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only**
mechanism:	**by reference**

Indexed pointer array.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZPM — Map banks

This routine produces a Jazelle post mortem dump.

FORMAT **JZPM** *[, LUN [, LEVEL [, WIDT]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

LUN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Logical unit number for output (Default 6).

LEVEL

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Output level (Default 2).

WIDTH

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Output width (Default 80).

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZPREG — Register a user pointer

Is used to register a user pointer. The pointer is inserted into a binary tree for the specified context along with the comment specifed.

FORMAT

JZPREG *COMMENT, CONTXT, PTR [, NPTR1 [, FLAGS1]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

COMMENT

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

The comment to be associated with pointer.

CONTXT

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

The context for this pointer.

PTR

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

The pointer to be registered.

NPTR1

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

The dimension of the pointer (default 1).

FLAGS1

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Flags associated with pointer (default 0) The bits in FLAGS are defined in JZPFLG.PAR. (No useful flags are yet defined.)

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZS — Character to (hollerith) string

This routine converts character-coded text to a Jazelle string (hollerith) data type.

FORMAT

JZS *N, IN, OUT*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

N

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

Length of JAZELLE string in bytes.

IN

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Character string to be converted.

OUT

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only**
mechanism:	**by reference**

Resultant JAZELLE string.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZS4 — 4-byte string to JAZELLE string

This routine converts a 4-byte character-coded text string to a 4-byte longword corresponding to a Jazelle string (hollerith) data type.

FORMAT

JZS4 *STRING*

RETURNS

VMS Usage:	**longword**
type:	**longword {INTEGER*4}**
access:	**write only**
mechanism:	**by value**

Jazelle string (hollerith) data. This longword corresponds to a 4-byte string representing character-coded data

ARGUMENTS

STRING

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

character string to be converted.

JZS8 — Character to 8-byte string

This routine converts an 8-byte character-coded text string to a Jazelle String*8 (hollerith) data type.

FORMAT

JZS8 *STRING*

RETURNS

VMS Usage:	**Quadword**
type:	**Quadword {REAL*8}**
access:	**write only**
mechanism:	**by value**

Jazelle string (hollerith) data. This quadword corresponds to an 8-byte hollerith string representing character-coded data. In Fortran, this corresponds to a double precision Real variable type.

ARGUMENTS

STRING

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Character string to be converted.

JZSTAT — Summary of memory usage

This routine produces a summary of Jazelle memory usage.

FORMAT

JZSTAT *[, LUN [, LEVEL [, WIDT]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

LUN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

LUN to which bank should be dumped (default 6).

LEVEL

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Dump level (default 0).

WIDTH

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Width of output (default 80).

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZSTRT — Initialize Jazelle

Initializes the JAZELLE Data Management System, and must be called once, before other JAZELLE routines can be used.

FORMAT

JZSTRT *[, LU]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

LUN

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

If specified JAZELLE banner will be written to LUN.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZTDEF — Add columns to tables

This is a user routine used to add new columns to tables and to create new tables.

FORMAT

JZTDEF *TABLEI, PATH, HEADER, FMT*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

TABLEI

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**modify**
mechanism:	**by reference**

Pointer to the table bank for the table If 0 table is created and pointer is set

Note: **If TABLEI is non-zero then PATH must belong to the same family as the table pointed to by TABLEI.**

PATH

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Path defining element to be added to table.

HEADER

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Title for this column. If blank element name is used.

FMT

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Format to be used. If blank default used.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZTDFL — Define default table for a bank

Allows users to specify a default table for a family.

FORMAT

JZTDFL *NAME, TABLEI [, TABLEO]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

NAME

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Family to which default is to be attached.

TABLEI

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Table to be used (0 default not changed).

TABLEO

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**write only (optional)**
mechanism:	**by reference**

Default table before any change.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZTMOD — Sets JAZELLE bank key values

Changes the key pointers in the specified bank to point to other banks (whose pointers are passed as KEY1..KEY5). If KEYn is 0, the corresponding key in the bank will not be changed.

FORMAT

JZTMOD *DATPTR [, KEY1 [, KEY2 [, KEY3 [, KEY4 [, KEY5]]]]]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

DATPTR

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Pointer to data region of bank containing keys.

KEY1

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Keys (if any) associated with this bank.

KEY2

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Keys (if any) associated with this bank.

KEY3

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Keys (if any) associated with this bank.

KEY4

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Keys (if any) associated with this bank.

KEY5

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only (optional)**
mechanism:	**by reference**

Keys (if any) associated with this bank.

RETURN VALUES

SLD$NORMAL	(S) Successful completion.
JZL$INCFAMLY	(E) bank pointed to belong to the wrong family.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZTSCN — Scan a table for key

This routine is used to scan a relational table finding all occurences of a specified key with a given value. The user supplies an array in which the pointers to the table entries are returned. If the array is not big enough for all the pointers an error occurs.

FORMAT

JZTSCN *TABLE, KEY, VALUE, MAXOUT, OUT, NOUT*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

TABLE

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Pointer to the family (table) as returned by JZBLOC.

KEY

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Name of the KEY to be scanned (as specified in the template for the table.

VALUE

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**read only**
mechanism:	**by reference**

Key value to be scanned for.

MAXOUT

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**read only**
mechanism:	**by reference**

Size of array OUT.

Note: **If MAXOUT is not big enough for all the entries found the routine will return an error condition and NOUT will be set to zero.**

OUT

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {POINTER}**
access:	**write only**
mechanism:	**by reference**

Array to receive pointers to found table entries. Must be dimensioned MAXOUT.

NOUT

VMS Usage:	**longword**
type:	**longword {INTEGER}**
access:	**write only**
mechanism:	**by reference**

Actual number of entries found.

Note: **If no entries are found with the specified value then NOUT will be set to zero and the routine will return an information level response.**

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZVERS — Jazelle Version numbers

This routine returns the current JAZELLE version number and, optionally, the IO system version number.

FORMAT

JZVERS *JVERS [, IOVERS]]*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

JVERS

VMS Usage:	**longword**
type:	**longword {REAL}**
access:	**write only**
mechanism:	**by reference**

The JAZELLE version number.

IOVERS

VMS Usage:	**longword**
type:	**longword {REAL}**
access:	**write only (optional)**
mechanism:	**by reference**

The IO system version number.

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

JZXWIP — Wipe an entire context

This routine deletes an entire context. This routine is much faster than deleting individual banks.

FORMAT

JZXWIP *CONTXT*

RETURNS

VMS Usage:	**longword_unsigned**
type:	**longword (unsigned) {INTEGER}**
access:	**write only**
mechanism:	**by value**

Longword condition value. Condition values that this function returns are listed under RETURN VALUES.

ARGUMENTS

CONTXT

VMS Usage:	**string**
type:	**character-coded text string {CHARACTER*(*)}**
access:	**read only**
mechanism:	**by descriptor**

Name of context to be wiped

RETURN VALUES

SLD$NORMAL	(S) Normal completion.
JZL$	Additional error conditions listed in the Jazelle Errors Appendix.

A Controlling JAZELLE's Global Parameters

Write-up not yet available.

B JAZELLE Utilties Programs

Write-up not yet available.

C Defining User Data Types

Write-up not yet available.

D Mapping Common Blocks to banks

Write-up not yet available.

E Error Conditions

Jazelle uses a standard error reporting mechanism to report all errors back to the user. See the SLD Error manual for more details on the error reporting system.

All the error conditions that can be returned are given below, along with an explanation of each error condition.

ABORT, Template processing aborted due to error

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-F-ABORT

ABORT, Constant processing aborted due to error

Facility: KONSTANT, Constant management system

Severity: KONSTANT-F-ABORT

ACCVIO, Access violation trapped by jazelle

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-ACCVIO

Explanation: An access violation (or 0C4 for you VMites) has occurred. JAZELLE has trapped it so that it can produce a post-mortem which may help someone figure out what has happened. Probably someone has called a routine (not necessarily a JAZELLE routine) with the wrong number of arguments, or someone is trying to use a pointer which has an invalid value. (Maybe someone is using an indexed pointer but the routine JZPIDX has not been called for the corresponding family). If not probably memory has been overwritten in some other way. There are two common ways of overwriting memory: The first is to write to an element of an array larger than the upper dimension, or smaller than the lower dimension (note that array here may mean either an array inside a JAZELLE bank, or an FORTRAN array). The second common way of overwriting memory is not to have allocated enough memory for a particular variable repeat block inside a bank. When a bank which contains a variable repeat block (or variable dimension) is created a certain number of blocks (or elements) are allocated. The user can then use as many or as few of these as he likes, however it is the users responsibility to ensure that he does not use more than are allocated. If more are required the bank must be expanded using the JZPEXP or JZBEXP routines.

ARGS, 'I' arguments passed to 'A' which expected 'I'-'I' arguments

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-ARGS

Explanation: A JAZELLE routine has been called with an inappropriate number of arguments. Read the manual!!

BADALIGN, JAZELL COMMON is incorrectly aligned

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADALIGN

Explanation: The common block JAZELL must be aligned in memory on an 8-byte boundary. On the VAX this error probably means that the line DUCSJAZELLE:JAZELLE/OPT was somehow missed out of your link command.

BADALO, Variable block count exceeds allocated count; bank 'A', Id '%JID'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADALO

BADARGC, Bad character argument 'A' = 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADARGC

BADBANKP, Inconsistent bank pointers in bank '%PTR'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADBANKP

Explanation: This error indicates that the memory managed by JAZELLE has become corrupted. While it is possible that this due to some internal JAZELLE bug it is more likely that the memory has been overwritten by some other program. There are two common ways of overwriting memory: The first is to write to an element of an array larger than the upper dimension, or smaller than the lower dimension (note that array here may mean either an array inside a JAZELLE bank, or an FORTRAN array). The second common way of overwriting memory is not to have allocated enough memory for a particular variable repeat block inside a bank. When a bank which contains a variable repeat block (or variable dimension) is created a certain number of blocks (or elements) are allocated. The user can then use as many or as few of these as he likes, however it is the users responsibility to ensure that he does not use more than are allocated. If more are required the bank must be expanded using the JZPEXP or JZBEXP routines.

BADBASE, Base must be 8, 10 or 16

Facility: KEY, Parsing utility

Severity: KEY-E-BADBASE

BADBLOCK, Bad BLOCK statment

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-BADBLOCK

BADBLOCK, Bad BLOCK statment

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-BADBLOCK

BADCHAR, JAZELLE name 'A' contains an illegal character ('A')

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADCHAR

Explanation: All JAZELLE names must be 1-8 characters, consisting only of uppercase letters, digits and the special symbols $ and _. All name must begin with a letter. JAZELLE names include bank names, element names, parameter names and context names.

BADCMOPT, Bad options specified for JZPCMP

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADCMOPT

Explanation: JZPCMP has been called with invalid option bits set in the OPTION parameter. DUCSJAZELLE:JZPCMP.PAR lists the valid options. These options can be logically or'ed together.

BADCNT, Illegal repeat count ('I'<0) given for JZBADD

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADCNT

Explanation: If specified, the repeat count for JZBADD must be >= 0.

BADCON, Bad context...cannot parse line

Facility: KEY, Parsing utility

Severity: KEY-F-BADCON

BADDEF, Default repeat count not valid in this location

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-BADDEF

BADDIMEN, A dimension has an illegal value

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-BADDIMEN

BADDNAME, Bad IO DDNAME = 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADDNAME

BADEBASE, Illegal base expression: 'A'

Facility: KEY, Parsing utility

Severity: KEY-E-BADEBASE

BADEND, End statement does not correspond to BANK or BLOCK statement

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-BADEND

BADEND, Constant file contains unclosed blocks

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-BADEND

BADEXPR, Error processing expression 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADEXPR

BADFAMP, Inconsistent Family pointers

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADFAMP

Explanation: This error indicates that the memory managed by JAZELLE has become corrupted. While it is possible that this due to some internal JAZELLE bug it is more likely that the memory has been overwritten by some other program. There are two common ways of overwriting memory: The first is to write to an element of an array larger than the upper dimension, or smaller than the lower dimension (note that array here may mean either an array inside a JAZELLE bank, or an FORTRAN array). The second common way of overwriting memory is not to have allocated enough memory for a particular variable repeat block inside a bank. When a bank which contains a variable repeat block (or variable dimension) is created a certain number of blocks (or elements) are allocated. The user can then use as many or as few of these as he likes, however it is the users responsibility to ensure that he does not use more than are allocated. If more are required the bank must be expanded using the JZPEXP or JZBEXP routines.

BADFILE, File 'A' has invalid format for JAZELLE IO

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADFILE

Explanation: An attempt has been made to read a file which JAZELLE does not recognize as containing valid JAZELLE data. Maybe it does not contain JAZELLE data, or alternately it may have been copied incorrectly, or it may have been somehow corrupted.

BADFMTP, Inconsistent format bank pointers in family 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADFMTP

Explanation: This error indicates that the memory managed by JAZELLE has become corrupted. While it is possible that this due to some internal JAZELLE bug it is more likely that the memory has been overwritten by some other program. There are two common ways of overwriting memory: The first is to write to an element of an array larger than the upper dimension, or smaller than the lower dimension (note that array here may mean either an array inside a JAZELLE bank, or an FORTRAN array). The second common way of overwriting memory is not to have allocated enough memory for a particular variable repeat block inside a bank. When a bank which contains a variable repeat block (or variable dimension) is created a certain number of blocks (or elements) are allocated. The user can then use as many or as few of these as he likes, however it is the users responsibility to ensure that he does not use more than are allocated. If more are required the bank must be expanded using the JZPEXP or JZBEXP routines.

BADFNUMB, number of banks in family 'A' inconsistent with family block

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADFNUMB

Explanation: The number of banks actually found in memory by traversing the linked list of banks does not match the number of banks as advertised in the family block. This is most likely due to an internal Jazelle error.

BADFORM, Record format 'A' unrecognised

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADFORM

BADHASH, Hash table corrupted

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADHASH

Explanation: This error indicates that the memory managed by JAZELLE has become corrupted. While it is possible that this due to some internal JAZELLE bug it is more likely that the memory has been overwritten by some other program. There are two common ways of overwriting memory: The first is to write to an element of an array larger than the upper dimension, or smaller than the lower dimension (note that array here may mean either an array inside a JAZELLE bank, or an FORTRAN array). The second common way of overwriting memory is not to have allocated enough memory for a particular variable repeat block inside a bank. When a bank which contains a variable repeat block (or variable dimension) is created a certain number of blocks (or elements) are allocated. The user can then use as many or as few of these as he likes, however it is the users responsibility to ensure that he does not use more than are allocated. If more are required the bank must be expanded using the JZPEXP or JZBEXP routines.

BADID, Illegal bank Id = '%JID'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADID

Explanation: Bank ID's must be in the range 0-32767. Sometime you can also use FIRST, LAST or ALL (or 'FRST','LAST','ALL*' as the argument to a routine.

BADIND, Bad index 'I' for element 'A'('I':'I') in '%JPTR'

Facility: JBCHK, JAZELLE bounds checking utility

Severity: JBCHK-E-BADIND

BADINDEX, Index 'I' out of range, 'A' dimensioned ('I':'I')

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADINDEX

Explanation: An attempt has been made to reference a dimension block or element with an index which is outside of the range of the dimension specified in the template. In the case of a variable dimension the dimension is outside the range of the currently used dimension.

BADKIND, Requested list 'A' with wrong kind of list

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADKIND

BADLASTP, Inconsistent last bank pointers in family 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADLASTP

Explanation: This error indicates that the memory managed by JAZELLE has become corrupted. While it is possible that this due to some internal JAZELLE bug it is more likely that the memory has been overwritten by some other program. There are two common ways of overwriting memory: The first is to write to an element of an array larger than the upper dimension, or smaller than the lower dimension (note that array here may mean either an array inside a JAZELLE bank, or an FORTRAN array). The second common way of overwriting memory is not to have allocated enough memory for a particular variable repeat block inside a bank. When a bank which contains a variable repeat block (or variable dimension) is created a certain number of blocks (or elements) are allocated. The user can then use as many or as few of these as he likes, however it is the users responsibility to ensure that he does not use more than are allocated. If more are required the bank must be expanded using the JZPEXP or JZBEXP routines.

BADLINE, 'A'

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-BADLINE

BADLINE, 'A'

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-BADLINE

BADLOGI, Invalid logical expression: 'A'

Facility: KEY, Parsing utility

Severity: KEY-E-BADLOGI

BADNODE, Corrupted jazelle balanced binary tree node. Addr= 'I'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADNODE

Explanation: This error indicates that the memory managed by JAZELLE has become corrupted. While it is possible that this due to some internal JAZELLE bug it is more likely that the memory has been overwritten by some other program. There are two common ways of overwriting memory: The first is to write to an element of an array larger than the upper dimension, or smaller than the lower dimension (note that array here may mean either an array inside a JAZELLE bank, or an FORTRAN array). The second common way of overwriting memory is not to have allocated enough memory for a particular variable repeat block inside a bank. When a bank which contains a variable repeat block (or variable dimension) is created a certain number of blocks (or elements) are allocated. The user can then use as many or as few of these as he likes, however it is the users responsibility to ensure that he does not use more than are allocated. If more are required the bank must be expanded using the JZPEXP or JZBEXP routines.

BADOPAPP, Bad use of APPEND option when opening file 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADOPAPP

Explanation: A file that is opened for non-direct access can not be opened with the append option, and a direct access file can not be opened with write and the append option. APPEND can not be used to append to a file not opened for direct access or to a file opened for write.

BADOPDEL, Bad use of DELETE option when opening file 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADOPDEL

Explanation: A file that is opened for non-direct access can not be opened with the delete option, and a direct access file can not be opened with the delete option if the file already exists.

BADOPEN, Open failure on IO device 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADOPEN

Explanation: An attempt to open a file or tape for JAZELLE IO has failed. Maybe the specified file or tape doesn't exist?

BADOPEXC, Bad use of EXCLUS option when opening file 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADOPEXC

Explanation: A file that is opened for non-direct access can not be opened with the exclusive option.

BADOPRW, Bad use of READ/WRITE when opening file 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADOPRW

Explanation: A sequential file can not be opened for both read and write.

BADOPT, Bad options specified for JZIORD

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADOPT

BADPARM, Undefined parameter

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADPARM

BADPATH, Bad path: 'A'

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-BADPATH

BADPATH, Bad pathname: 'A' ... 'A'...

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADPATH

BADPTR, Jazelle bounds checking detected bad pointer ('%JPTR')

Facility: JBCHK, JAZELLE bounds checking utility

Severity: JBCHK-E-BADPTR

BADPTR, Error in PTR value 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADPTR

BADRANGE, Second value in range must be greater than or equal to the first

Facility: PATH, JAZELLE path parsing utility

Severity: PATH-E-BADRANGE

BADRANGE, The first dimension must be less than or equal to the second

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-BADRANGE

BADRANGE, Index out of range: bank = 'A', pathname = 'A', index = 'I'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADRANGE

BADREAD, Error during read operation on file 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADREAD

BADRENM, Can not rename record to 'A'/'I':'I'. New name already in use.

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADRENM

Explanation: The record can not be renamed to the requested name because that record name is already in use by another record.

BADRWOPT, Bad RW option = 'A' in JZIOPN

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADRWOPT

BADSYNTX, Syntax error whilst parsing constant file

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-BADSYNTX

BADTABLE, Insufficient rows specified in table

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-BADTABLE

BADTBLPT, Bad table pointer specified for family 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADTBLPT

BADTEMPP, Inconsistent template pointers in family 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BADTEMPP

Explanation: This error indicates that the memory managed by JAZELLE has become corrupted. While it is possible that this due to some internal JAZELLE bug it is more likely that the memory has been overwritten by some other program. There are two common ways of overwriting memory: The first is to write to an element of an array larger than the upper dimension, or smaller than the lower dimension (note that array here may mean either an array inside a JAZELLE bank, or an FORTRAN array). The second common way of overwriting memory is not to have allocated enough memory for a particular variable repeat block inside a bank. When a bank which contains a variable repeat block (or variable dimension) is created a certain number of blocks (or elements) are allocated. The user can then use as many or as few of these as he likes, however it is the users responsibility to ensure that he does not use more than are allocated. If more are required the bank must be expanded using the JZPEXP or JZBEXP routines.

BADTINDX, Bank name in table definition cannot be qualified with index

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BADTINDX

BANKNEST, BANK statement cannot be nested

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-BANKNEST

BANKNEST, BANK statement cannot be nested

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-BANKNEST

BIGTOC, Table of contents too big for IO header record

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-BIGTOC

BTABERR, Inconsistency in table of static banks. Missing entry.

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BTABERR

Explanation: Somehow a bank has been flagged as static yet is is not in the JSTCTAB table.

BUGCHECK, Unexpected error returned from GETVM/FREEVM

Facility: GETVM, Virtual memory utility

Severity: GETVM-F-BUGCHECK

BUGCHECK, Jazelle internal coding error

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-BUGCHECK

Explanation: JAZELLE internal coding error. Report to JAZELLE expert.

BYTESWAP, File 'A' has bytes swapped...converting

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-BYTESWAP

Explanation: JAZELLE has detected that the byte-ordering in a file it is reading is not that which it expected. JAZELLE will translate each record as it is read in, but this is a time consuming process so if you intend to read this file many times, or if it is very long, you may wish to use the JIO command to convert it to the correct format first.

CONTEND, Template ends with a continuation line

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-CONTEND

CONTRACT, Bank '%JPTR' has repeat count 'I' > requested allocate count 'I'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-CONTRACT

Explanation: JZBEXP or JZPEXP has been called to contract a bank. However the allocation requested for the variable dimension or variable repeat count is less than is currently being used.

CTRLCABT, JAZELLE output aborted by CTRL_C interrupt

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-I-CTRLCABT

Explanation: You hit CTRL^C or its equivalent during JAZELLE output. JAZELLE was insulted and has decided to give you the silent treatment.

DECIMAL, Illegal character in decimal integer constant-'A'

Facility: KEY, Parsing utility

Severity: KEY-E-DECIMAL

DISKFULL, IO disk full

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-DISKFULL

Explanation: The message just about says it all.

DUPFAM, Family 'A' already exists

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-DUPFAM

DUPKEY, Duplicate key in jazelle balanced binary tree

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-I-DUPKEY

DUPNAME, The name 'A' has been defined twice in a bank

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-DUPNAME

ENDOFLIN, More characters expected in path

Facility: PATH, JAZELLE path parsing utility

Severity: PATH-E-ENDOFLIN

EOF, End of file

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-I-EOF

Explanation: End of file has been detected during a read operation on JAZELLE data. Probably nothing to worry about.

EOFLINE, End of line found too soon

Facility: KEY, Parsing utility

Severity: KEY-E-EOFLINE

EOREEL, Tape has run off reel

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-EOREEL

Explanation: Boy are you going to be popular with the tape operator. Probably you didn't stop writing after a previous routine reported you had reached the end of tape. Or maybe your events are just so hugh they don't fit. Better consult a JAZELLE expert.

EOT, End of tape

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-EOT

Explanation: End of tape detected during a read operation on JAZELLE data. If more than one tape was specified for input reading will continue with the next tape, otherwise the read will be terminated.

ERROR, Jazelle bounds checking detected error in routine 'A'

Facility: JBCHK, JAZELLE bounds checking utility

Severity: JBCHK-F-ERROR

EXISTS, Bank already initialized, no action taken

Facility: KONSTANT, Constant management system

Severity: KONSTANT-I-EXISTS

EXISTS, Input bank, 'A'('I'), already exists

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-EXISTS

Explanation: An attempt has been made to read a bank using JAZELLE IO, but a bank of the same name and ID already exists. JAZELLE IO supports three options for reading banks, ADD, REPLACE and APPEND. This error can only be generated with the ADD option specified.

EXTRA, Extra tokens

Facility: KEY, Parsing utility

Severity: KEY-E-EXTRA

FCPFAIL, File copy failed

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-FCPFAIL

FEXISTS, Input family, 'A', already exists

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-FEXISTS

Explanation: An attempt has been made to read a family of banks using JAZELLE IO, but at least one bank in the family already exists already exists. JAZELLE IO supports three options for reading data, ADD, REPLACE and APPEND. This error can only be generated with the ADD option specified.

FRSTBANK, First bank of family (could not locate first - 1)

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-I-FRSTBANK

HASHED, Eight letter name used with $USEBANK, 'A' shortened to 'A', check for conflicts

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-HASHED

Explanation: Due to IBM silliness common blocks are only allowed to have seven characters. Since using $USEBANK generates a common block of the same name as the family it is not advisable to use 8 character bank names. If you do JAZELLE will form a common block name by omitting one letter from the bank name, but this may cause clashes so beware.

HEADONLY, Names beginning with JB$ are reserved for use by JAZELLE

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-HEADONLY

HEX, Illegal character in hexadecimal integer constant-'A'

Facility: KEY, Parsing utility

Severity: KEY-E-HEX

IDEXISTS, Requested new Id already exists, bank 'A'('%JID')

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-IDEXISTS

Explanation: An attempt has been made to add a bank when a bank with the same name and ID already exists.

IDTOOBIG, Attempt to create ID='I'>MAXID='I' in family 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-IDTOOBIG

Explanation: An attempt has been made to create a bank with an ID bigger than the maximum ID (MAXID) specified in the template for that family. Maybe the program logic is incorrect or maybe you have found an event with more tracks or vertices than anyone imagined we would ever have. You can increase the MAXID in the template of the bank, but you must recompile at least one routine which references the template withe a $USEBANK statement or disaster will occur. (If there are no routines which reference this template with a $USEBANK you don't need to recompile anything...lucky you!)

ILCHRS, Illegal characters found

Facility: KEY, Parsing utility

Severity: KEY-E-ILCHRS

ILGLOPTN, Illegal option passed to JZKGET, call ignored

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-ILGLOPTN

ILLCHR, Illegal character 'A'

Facility: KEY, Parsing utility

Severity: KEY-E-ILLCHR

ILLGLFMT, Illegal FMT='A' specified in template, format ignored

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-ILLGLFMT

Explanation: An explicit format has been specified for an element in a template using the FMT= qualifier, but the value specified is not a legal FORTRAN format specifier. The qualifier has been ignored.

ILLOP, Operand 'A' has illegal value

Facility: KEY, Parsing utility

Severity: KEY-E-ILLOP

ILLRW, 'A' attempted on a file that was not opened for that action.

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-ILLRW

Explanation: A file that is not opened for read can not be read so this error occurs when someone attempts to perform a read on the file and the same goes for write.

ILLVAL, Operand 'A' has illegal value ('A'<=op<='A')

Facility: KEY, Parsing utility

Severity: KEY-E-ILLVAL

INCBANKS, Incompatible banks (must use the same template)

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-INCBANKS

Explanation: The operation failed because the two banks specified are not compatible. Banks are considered to be compatible iff they use the same template. In the current version of Jazelle that means that they have to belong to the same family.

INCNEST, Inconsistent nesting of Blocks

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-INCNEST

INCNEST, Inconsistent block nesting in constant file

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-INCNEST

INCONKEY, Key value '%JPTR' is inconsistent with binding 'A' in family 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-INCONKEY

Explanation: An attempt has been made to store a value into a key, however the value is not consistent with the type of pointer. More specifically the value given for the key is not a pointer to a bank of the kind declared for that key in the TEMPLATE for the bank containing the key.

INFILE, Error occurred in file 'A'

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-INFILE

INFILE, Error occurred in file 'A'

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-INFILE

INIT, Jazelle bounds checking initiated from routine 'A'

Facility: JBCHK, JAZELLE bounds checking utility

Severity: JBCHK-I-INIT

INITBAD, The number of initial values does not equal the number of elements

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-INITBAD

INITOOLN, An initial value string is too long

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-INITOOLN

INPATH, Error in path 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-INPATH

INSFSTOR, Insufficient virtual storage available

Facility: GETVM, Virtual memory utility

Severity: GETVM-F-INSFSTOR

INVDSIZE, Invalid virtual memory request (non-positive byte count)

Facility: GETVM, Virtual memory utility

Severity: GETVM-F-INVDSIZE

INVDWORD, Invalid word size request

Facility: GETVM, Virtual memory utility

Severity: GETVM-F-INVDWORD

INVTOKEN, Invalid token 'A' in path 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-INVTOKEN

IOCONV, File 'A' is being converted from 'A' format to 'A' format

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-IOCONV

Explanation: A file is being read which was written on a different type of computer from the one it is being read on. JAZELLE will translate each record as it is read in, but this is a time consuming process so if you intend to read this file many times, or if it is very long, you may wish to use the JIO command to convert it to the correct format first.

IOFAIL, IO failure

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-IOFAIL

Explanation: How can such a clear message be explained.

IOSYNCH, IO records out of synch

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-IOSYNCH

Explanation: JAZELLE has detected input records which are not in the order in which it expected them to be. Maybe the data is corrupted (disk/tape problem, incorrectly copied?) or maybe the program which wrote this data experienced some unexpected problems?

IOVCONV, File 'A' is being converted from JAZELLE V'F5.2' to V'F5.2' format

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-IOVCONV

Explanation: The input file just opened was created with an older version of Jazelle. Since it is of a different format, the data will be converted during the read operation. This is transparent to the user, but is more CPU intensive; also, current data formats may not be supported in future releases. Thus, it is a good idea to keep data files up to date at the current level of Jazelle. Utilities are provided with Jazelle to convert data files to the current data format.

IOVERS, IO Record format is not supported (anymore)

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-IOVERS

Explanation: The data file cannot be read because it was written by an ancient version of Jazelle which is not supported anymore or because it was written by a more recent version of Jazelle.

KEYNTFND, Key 'A' does not exist in table 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-KEYNTFND

Explanation: The name of a non-existent key has been specified for a table scan.

KEYREP, Keyword 'A' specified twice.

Facility: KEY, Parsing utility

Severity: KEY-E-KEYREP

LASTBANK, Last bank of family (could not locate last + 1)

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-I-LASTBANK

LTOOLONG, A line in the template is too long

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-LTOOLONG

LTOOLONG, A line in the template is too long

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-LTOOLONG

MAXCHAIN, Attempt to create too many constant chains

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-MAXCHAIN

Explanation: Consult a JAZELLE expert.

MAXIDMIS, Template for family 'A' should specify MAXID or NOMAXID

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-MAXIDMIS

Explanation: All templates must specify either MAXID=n or NOMAXID. If you specify NOMAXID then no-one will be able to use indexed pointers with banks in the family specified with the template. This is generally thought to be anti-social.

MISMAT, Mismatched

Facility: KEY, Parsing utility

Severity: KEY-E-MISMAT

MISMATCH, Banks dont match

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-I-MISMATCH

MISMATP, Mismatched parentheses ()

Facility: KEY, Parsing utility

Severity: KEY-E-MISMATP

MISSDEL, Missing delimiter in line.

Facility: KEY, Parsing utility

Severity: KEY-E-MISSDEL

MISSOP, Missing operand 'A'

Facility: KEY, Parsing utility

Severity: KEY-E-MISSOP

MTRANGE, Range ('I':'I') is empty

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-MTRANGE

MULTINIT, JZSTRT called more than once, multiple call ignored

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-MULTINIT

Explanation: The JAZELLE initialization routine JZSTRT has been called more than once in the same program. Calls after the first call are ignored.

MULTVARD, Only one variable dimension allowed per bank

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-MULTVARD

NEEDMXID, MAXID must be specified in template to use JZPIDX (bank 'A')

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-NEEDMXID

Explanation: In order to use indexed pointers with a family the TEMPLATE for that family must specify MAXID=n. Most banks do this but for some banks this is not possible or undesirable for some other reason. In this case indexed pointers cannot be used, and the POINTER macro should be used instead.

NESTBANK, Bank 'A' illegally referenced from bank 'A'

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-F-NESTBANK

NEVERALO, Attempt to free unallocated memory, Address 'z8', Size 'z8'

Facility: GETVM, Virtual memory utility

Severity: GETVM-E-NEVERALO

NOBANK, Template must start with BANK or BLOCK statement

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NOBANK

NOBANK, Constant file must start with BANK statement

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-NOBANK

NOBANK, Requested bank does not exist

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-I-NOBANK

NOBANK1, Bank 'A' Id '%JID' not found

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOBANK1

NOBIND, Only pointers and keys can be bound to a family

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NOBIND

NOBIND, Unable to bind pointer in family 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-NOBIND

Explanation: A pointer or key has been declared in a bank, but the template for the bank to which the pointer points to (–>) can not be found. The program will continue anyway.

NOCHANCE, Context JAZELLE cannot be wiped

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-NOCHANCE

Explanation: You have tried to wipe the JAZELLE context. JAZELLE is programmed to preserve itself at all costs, so it won't let you do that.

NOCOPY, Bank was not copied

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOCOPY

Explanation: A bank copy operation has failed. No banks have been created. Consult the messages issued in conjunction with this message for further clues as to why the operation failed. Most likely it is due to the source and target banks being of incompatible types.

NODEFTBL, No default table defined for family 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NODEFTBL

Explanation: An attempt has been made to tabulate a family for which no default table has been defined (either in the template or otherwise). You must explicitly define which elements of the family you wish to tabulate. You can do this by using the JZTYDEF routine, either directly or indirectly (for example using the IDA TABLE command).

NODEVICE, IO device, ID = 'I', has not been opened

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NODEVICE

Explanation: A read or write operation has been attempted on a file or tape which has never been opened. try opening it first.

NODOTS, Path must contain at least one period (.)

Facility: PATH, JAZELLE path parsing utility

Severity: PATH-E-NODOTS

NOEQUAL, Equals sign missing or misplaced in POKE command

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOEQUAL

NOEXPAND, Cannot expand fixed length bank: '%JPTR'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOEXPAND

Explanation: JZBEXP or JZPEXP has been called to expand a bank which doesn't have a variable dimension. Try reading the manual again.

NOFAMILY, JZFDEF has been called for a non-existent family

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOFAMILY

NOFORMAT, A FMT= qualifier is only allowed with JAZELLE intrinsic types

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NOFORMAT

NOFRECON, No more free contexts...cannot parse line

Facility: KEY, Parsing utility

Severity: KEY-F-NOFRECON

NOIBMOCT, Octal numbers are not implemented on VM

Facility: KEY, Parsing utility

Severity: KEY-E-NOIBMOCT

NOINIT, Blocks must have a POKE routine defined to be initialized

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NOINIT

NOINIT, JAZELLE has not been initialized (c.f. JZSTRT)

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-NOINIT

Explanation: An attempt has been made to call a JAZELLE routine before the JAZELLE initialization routine JZSTRT has been called. JZSTRT must be the first JAZELLE routine called in any program.

NOKEY, Jazelle balanced binary tree key not found

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-NOKEY

Explanation: This error indicates that the memory managed by JAZELLE has become corrupted. While it is possible that this due to some internal JAZELLE bug it is more likely that the memory has been overwritten by some other program. There are two common ways of overwriting memory: The first is to write to an element of an array larger than the upper dimension, or smaller than the lower dimension (note that array here may mean either an array inside a JAZELLE bank, or an FORTRAN array). The second common way of overwriting memory is not to have allocated enough memory for a particular variable repeat block inside a bank. When a bank which contains a variable repeat block (or variable dimension) is created a certain number of blocks (or elements) are allocated. The user can then use as many or as few of these as he likes, however it is the users responsibility to ensure that he does not use more than are allocated. If more are required the bank must be expanded using the JZPEXP or JZBEXP routines.

NOKEYBLK, Keys can not occur in blocks (as opposed to banks)

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NOKEYBLK

NOKEYDIM, Keys cannot be dimensioned

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NOKEYDIM

NOKEYSTC, Keys can not occur in static banks

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NOKEYSTC

NOKEY1, Jazelle balanced binary tree key not found

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-I-NOKEY1

NOLETTER, JAZELLE name 'A' does not begin with a letter

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOLETTER

Explanation: All JAZELLE names must be 1-8 characters, consisting only of uppercase letters, digits and the special symbols $ and _. All name must begin with a letter. JAZELLE names include bank names, element names, parameter names and context names.

NOLIST, List 'A' Id '%JID' not found

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-NOLIST

NOMAXID, MAXID must be specified in template to use $USEBANK (bank 'A')

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOMAXID

Explanation: In order to use a $USEBANK statement for a family the TEMPLATE for that family must specify MAXID=n. Most banks do this but for some banks this is not possible or undesirable for some other reason. In this case $USEBANK cannot be used, and the POINTER macro should be used instead.

NOMEMORY, Unable to allocate virtual memory

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-NOMEMORY

NONINTGR, Variable dimension must be an INTEGER*4

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NONINTGR

NOPADSTC, JAZELLE tried to add padding to a static bank

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NOPADSTC

Explanation: Jazelle aligns elements in banks on multiples of four bytes and then pads the in between space (if necessary). Common blocks do not have any padding so a static bank which is the copy of a common block can not have any either. In other words, all common blocks that are mapped to a static bank must consist of elements that are of a size that is a multiple of four bytes.

NOPARM, Undefined parameter

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-I-NOPARM

NORANGE, Cannot specify range with bank name in constant file

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-NORANGE

NORANGE, Cannot handle range of indices

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NORANGE

Explanation: A range of ID's or indices (ie 1:3) has been used in a place where its use is not supported. Use an explicit ID (eg 2) instead.

NORMAL, Normal successful completion

Facility: KONSTANT, Constant management system

Severity: KONSTANT-S-NORMAL

NORMAL, Normal successful completion

Facility: KEY, Parsing utility

Severity: KEY-S-NORMAL

NOROUTNE, JAZELLE has unexpectedly been unable to find the JROUTINE bank

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-NOROUTNE

Explanation: Consult a JAZELLE expert.

NOSCALER, Variable dimension must be a scaler

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NOSCALER

NOSUPORT, Setting values for this type of variable is not supported

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOSUPORT

NOSYMBOL, Symbol does not appear in JAZELLE family table

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-I-NOSYMBOL

NOTABCRE, Cannot create table for table insertion

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOTABCRE

NOTABLE, ENDTABLE occurs outside a table definition

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-NOTABLE

NOTALLOC, Attempt to free memory that was not allocated

Facility: GETVM, Virtual memory utility

Severity: GETVM-E-NOTALLOC

NOTDIR, Cannot get a directory for a sequential file.

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOTDIR

Explanation: An attempt was made to get a directory for a file that has been opened for sequential access and for sequential files there are no directories, only direct access files have them.

NOTENTRY, Cannot delete table entry–not found in table

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-NOTENTRY

Explanation: Either the pointer used in the operation does not point to a proper Jazelle memory allocation table, or the table entry is already flagged as deleted. This error is most likely to occur during the expansion or deletion of banks when the pointer to a bank has been modified by incrementing or decrementing it or by assigning a random value to it, or when an attempt is made to delete the same bank more than once. If not, probably memory has been overwritten in some other way. There are two common ways of overwriting memory: The first is to write to an element of an array larger than the upper dimension, or smaller than the lower dimension (note that array here may mean either an array inside a JAZELLE bank, or an FORTRAN array). The second common way of overwriting memory is not to have allocated enough memory for a particular variable repeat block inside a bank. When a bank which contains a variable repeat block (or variable dimension) is created a certain number of blocks (or elements) are allocated. The user can then use as many or as few of these as he likes, however it is the users responsibility to ensure that he does not use more than are allocated. If more are required the bank must be expanded using the JZPEXP or JZBEXP routines.

NOTFOUND, Template for family 'A' could not be found

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NOTFOUND

NOTFOUND, Constant file 'A' could not be found

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-NOTFOUND

NOTIMPL, 'A' not yet implemented

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOTIMPL

Explanation: Of course if you would like to volunteer your assistance...

NOTLIST, Bank 'A', Id '%JID' is not a list

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-NOTLIST

NOTSTIC, JZBMAP cannot be used to add nonstatic banks

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOTSTIC

Explanation: An attempt was made to use the JZBMAP routine on nonSTATIC banks.

NOTUSED, Parameter 'A' not used.

Facility: KEY, Parsing utility

Severity: KEY-E-NOTUSED

NOVARBLK, BLOCKs cannot contain variable dimensions

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NOVARBLK

NOVARSTC, Static banks cannot contain variable dimensions

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-NOVARSTC

Explanation: Since static banks are by definition static they can not have variable dimensions.

NOVCONV, File 'A' written with JAZELLE V'F5.2' cannot be read by V'F5.2'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-NOVCONV

Explanation: In general old versions of JAZELLE cannot read files written by more recent versions of JAZELLE. Relink reading program with most recent version of JAZELLE.

NOWILD, Cannot specify wild cards with bank name in constant file

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-NOWILD

NOWILD, Cannot handle wild cards

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOWILD

Explanation: The * character, or the word ALL, has been used as a wild card in a position for which its use is not supported. Replace it with an explicit ID or index.

NOWMTCTX, Context 'A' is empty and therefore cannot be written out

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NOWMTCTX

Explanation: JAZELLE is currently unable to write contexts which contain no banks. I know you REALLY wanted to write an empty context...if you like I could explain how to upgrade the IO routines...it will probably only take you a month or so.

NTENGHRM, Scan for key 'A'='%JPTR' results in too many matches

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-NTENGHRM

Explanation: JZTSCN has been called to scan a relational table, but it has found more entries matching the specified search criterian than it is able to fir into the output buffer provided. There are better ways to scan a table than tio use JZTSCn...maybe you will want to consult a JAZELLE expert, or maybe you just want to increase the size of the array passed to JZTSCN.

NUCSDEST, Nucleus storage pointers destroyed

Facility: GETVM, Virtual memory utility

Severity: GETVM-F-NUCSDEST

OBSOLETE, Q% macro is now obsolete, use PTR% macro instead

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-OBSOLETE

OCTAL, Illegal character in octal integer constant-'A'

Facility: KEY, Parsing utility

Severity: KEY-E-OCTAL

OLDRTNE, Routine 'A' is obsolete, use 'A' instead

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-OLDRTNE

Explanation: Some very old JAZELLE routines have been discontinued. Consult the manual for details on the new usage.

OPENBANK, Template contains a bank or block with no END statement

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-F-OPENBANK

OPENBANK, Constant file contains a bank or block with no END statement

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-OPENBANK

OVERFLOW, Error translating 'A' (probably overflow)

Facility: KEY, Parsing utility

Severity: KEY-E-OVERFLOW

PARSE, Error parsing line

Facility: KEY, Parsing utility

Severity: KEY-E-PARSE

PATHINV, Path contains invalid token

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-PATHINV

PMOPENF, Unable to open post mortem file specified by JPM

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-PMOPENF

Explanation: You (or someone else) has requested that the JAZELLE post-mortem that is generated automatically when a program abends be put into a file. file instead of being dumped to the normal output stream. JAZELLE was however unable to open the specified file for some reason...too bad it would probably have made interesting reading.

PMSPRSD, Post-mortem suppressed by JPM

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-PMSPRSD

Explanation: You (or someone else) has requested that the JAZELLE post-mortem that is normally generated when a program abends be suppressed. We are happy to oblige...just don't ask us to figure out what the problem is.

PNTRDEST, User storage pointers destroyed

Facility: GETVM, Virtual memory utility

Severity: GETVM-F-PNTRDEST

PSHOVER, Pushdown overflow

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-PSHOVER

Explanation: JAZELLE maintains a stack where it stores temporary information. Normally the stack amply large to contain all the temporary information but somehow it has overflowed. Unless you have an extremely large number of banks this probably indicates some internal JAZELLE error. It is possible to increase the maximum stack size allowed by changing the value of the STACKMAX element in the JAZELLE bank.

PSHUNDER, Pushdown underflow

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-PSHUNDER

Explanation: This error probably indicates an internal JAZELLE error. Contact a JAZELLE expert.

PSYNTAX, Syntax error in parameter statement

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-PSYNTAX

PTHEXTRA, Extra tokens following legal path: 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-PTHEXTRA

PTOOLONG, Path name is too long

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-PTOOLONG

RDFAIL, IO read fail

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-RDFAIL

Explanation: An IO failure has occurred during a read operation on JAZELLE data. Maybe your tape is dirty or otherwise corrupted (you probably shouldn't have left it by that magnet!

REAL, Illegal character in real constant-'A'

Facility: KEY, Parsing utility

Severity: KEY-E-REAL

RECCOPY, 'I' records successfully copied

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-I-RECCOPY

RECNF, Could not find requested record name 'A' with param 'I':'I'.

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-RECNF

Explanation: The record requested for reading was not found.

RECURSE, JZKGET has been called recursively

Facility: KONSTANT, Constant management system

Severity: KONSTANT-F-RECURSE

RESERVED, The family name 'A' is reserved for use by JAZELLE

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-RESERVED

RESRVD, 'A,A,A' is a reserved record specification. Rename record.

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-RESRVD

Explanation: The requested record name is reserved for a direct access file's directory. This record should not be written to file by anyone, only internal code can play around with it.

STATIC, STATIC banks cannot be 'A'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-STATIC

Explanation: The only thing that can be done to static banks is peeking and poking. They are added via the JZBMAP routine and so cannot be added, deleted, wiped, expanded, included in lists, and other such stuff that would contradict the very meaning of the word static.

STATICON, Cannot 'A' STATIC context

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-STATICON

Explanation: This is really the same error as JZL%STATIC except that the operation was based on the context rather than just on the bank.

STCARG, Requested template already mapped with different parameters.

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-STCARG

Explanation: An attempt was made to use the JZBMAP routine to map an already mapped template to an address different from the one it is already mapped to and/or to map it with a different I/O specification.

SYNTAX, Syntax error in path: 'A'

Facility: PATH, JAZELLE path parsing utility

Severity: PATH-E-SYNTAX

SYNTAX, Syntax error whilst parsing template

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-SYNTAX

TABALIGN, Table entry pointer alignment error

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-F-TABALIGN

Explanation: The pointer used in the operation points to a proper Jazelle memory allocation table, but it does not coincide with the starting address of any of its entries. This error is most likely to occur during the deletion of banks when the pointer to a bank has been modified by incrementing or decrementing it or by assigning a random value to it. If not, probably memory has been overwritten in some other way. There are two common ways of overwriting memory: The first is to write to an element of an array larger than the upper dimension, or smaller than the lower dimension (note that array here may mean either an array inside a JAZELLE bank, or an FORTRAN array). The second common way of overwriting memory is not to have allocated enough memory for a particular variable repeat block inside a bank. When a bank which contains a variable repeat block (or variable dimension) is created a certain number of blocks (or elements) are allocated. The user can then use as many or as few of these as he likes, however it is the users responsibility to ensure that he does not use more than are allocated. If more are required the bank must be expanded using the JZPEXP or JZBEXP routines.

TABLPATH, Error creating table entry

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-TABLPATH

TBTOOBIG, Too many rows specified in table

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-TBTOOBIG

TBTOOWID, Too many columns in table (Maximum allowed 'I')

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-TBTOOWID

TBWRNGE, Wrong number of values specified for table row, 'I' given, 'I' expected

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-TBWRNGE

TMNYNAME, Too many names have been defined in one bank

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-TMNYNAME

TMYDEL, Too many delimiters in line (max.'I')

Facility: KEY, Parsing utility

Severity: KEY-E-TMYDEL

TMYKEY, Too many keywords specified (max. 'I')

Facility: KEY, Parsing utility

Severity: KEY-E-TMYKEY

TNDRNGE, An explicit range musty be given for the TABLE control variable

Facility: KONSTANT, Constant management system

Severity: KONSTANT-E-TNDRNGE

TOMNYCTX, Attempt to create too many contexts (VM zones)

Facility: GETVM, Virtual memory utility

Severity: GETVM-F-TOMNYCTX

TOODEEP, Too many nested blocks in block 'A'

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-F-TOODEEP

TOODEEP, Path contains too many tokens

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-TOODEEP

TOOLGE, No number where expected or number too large

Facility: KEY, Parsing utility

Severity: KEY-E-TOOLGE

TOOLONG, A block or bank name is too long

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-TOOLONG

TOOLONG, Operand 'A' too long (max. length 'I')

Facility: KEY, Parsing utility

Severity: KEY-E-TOOLONG

TOOLONG, JAZELLE name 'A' is longer than 8 characters

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-TOOLONG

Explanation: All JAZELLE names must be 1-8 characters, consisting only of uppercase letters, digits and the special symbols $ and _. All name must begin with a letter. JAZELLE names include bank names, element names, parameter names and context names.

TOOMNYDF, JZFDEF has been called for too many families

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-TOOMNYDF

TRUNCATE, Token 'A' truncated

Facility: KEY, Parsing utility

Severity: KEY-W-TRUNCATE

TRUNCATE, String 'A' was too long and has been truncated

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-TRUNCATE

TRUNLINE, Line too long, truncated

Facility: KEY, Parsing utility

Severity: KEY-W-TRUNLINE

TUFFLUCK, PEEK/POKE to variables of this type is not allowed

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-TUFFLUCK

UEXPCHAR, Unexpected characters: 'A'

Facility: PATH, JAZELLE path parsing utility

Severity: PATH-E-UEXPCHAR

UEXPDELM, Unexpected delimiter: 'A'

Facility: PATH, JAZELLE path parsing utility

Severity: PATH-E-UEXPDELM

UNDEFINE, Undefined block 'A' referenced in block 'A'

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-F-UNDEFINE

VALUE, Illegal value

Facility: KEY, Parsing utility

Severity: KEY-E-VALUE

VARDLAST, Variable dimension element must be last in bank

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-E-VARDLAST

VARINIT, Variable dimension cannot be initialized, initial value ignored

Facility: TEMPLATE, Template parsing system

Severity: TEMPLATE-I-VARINIT

WRNGNKEY, 'I' key values specified but template 'A' defines 'I' keys

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-WRNGNKEY

Explanation: An attempt has been made to ADD a bank, but the number of keys specified does not correspond to the number of keys declared in the TEMPLATE for that bank.

WRONGBNK, Path 'A' specifies element in wrong family for table '%JPTR'

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-E-WRONGBNK

Explanation: All columns in a table must represent elements in the same bank. You have attempted to mix different banks within one table.

WRONGSIZ, Attempt to deallocate wrong size, Address 'z8', Size 'z8'

Facility: GETVM, Virtual memory utility

Severity: GETVM-E-WRONGSIZ

WROTEPM, Jazelle wrote post mortem to disk file

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-WROTEPM

Explanation: You (or someone else) has requested that the JAZELLE post-mortem that is generated automatically when a program abends be put into a file instead of being dumped to the normal output stream. JAZELLE has obliged.

WRTFAIL, IO write fail

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-WRTFAIL

Explanation: IO failure during a write operation of JAZELLE data. Probably means your tape is dirty of something of that ilk.

WTRUNC, Output was too long and was truncated

Facility: JAZELLE, JAZELLE data manager

Severity: JAZELLE-W-WTRUNC

Index