Multimission Image Processing Laboratory

VICAR Run-Time Library Reference Manual

R. Deen
L. Bolef

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL D-4311 Rev B

Contact: Robert Deen, Robert.G.Deen@jpl.nasa.gov.

JPL

1. Introduction

This manual is an introduction to writing software using the VICAR image processing executive and a reference manual for the experienced VICAR programmer. Coding examples are in FORTRAN and C; a knowledge of at least one of these languages is assumed. This manual should be used with "The VICAR User's Guide": http://www-mipl.jpl.nasa.gov/PAG/public/vug/vugfinal.html and the "VICAR Addendum to the TAE User's Reference Manual".

Before changing or editing this manual, please see 13 Appendix E: About This Document (page 146).

1.1 Document Organization

The VICAR RTL (Run-Time Library) is a collection of subroutines for writing image processing software. We divide these subroutines into six packages, each of which is described in its own section:

- Image I/O**:** A set of subroutines to allow the input and output of image data to image files (page 41).
- Label I/O**:** A set of subroutines allowing access to information stored in the "VICAR label". The VICAR label contains both information about the file and user supplied information (page 65).
- Parameter I/O**:** A set of subroutines which allow access to user supplied parameters, from both the command line and from files (page 81).
- Translation Routines**:** Subroutines to translate pixel formats (page 93).
- FORTRAN String Conversion Routines**:** Subroutines to convert from and to differing string representations (page 102).
- Utility Routines**:** Miscellaneous subroutines for terminal I/O, program termination, etc. (page 107).

VICAR data formats are first discussed in detail to provide the basic knowledge needed. Next is an introduction to programming under VICAR. This includes general information, argument passing conventions and how to write your first program under VICAR.

Succeeding sections cover the six subroutine packages listed above, describing each subroutine call in detail. Subroutine descriptions are followed by notes on using the VICAR run-time library and examples of use. An appendix contains detailed descriptions of all the error messages and status codes from run-time library calls.

The VICAR Virtual Raster Display Interface (VRDI) provides device independent access to display devices. Use of the VRDI is deprecated and should not be used for new code It is described in a separate manual: "MIPL Virtual Raster Display Interface User's Reference Guide, JPL D-5100", available in TeX dvi form: http://www-mipl.jpl.nasa.gov/vrdi/vrdi.dvi.

1.2 Acronym List

ANSI	American National Standards Institute
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AST	Asynchronous System Trap

BIL	Band Interleaved by Line
BIP	Band Interleaved by Pixel
BSQ	Band SeQuential
CMS	Code Management System
DEC	Digital Equipment Corporation
FEI	File Exchange Interface
FITS	Flexible Image Transport System
FORTRAN	FORmula TRANslator
FPS	Floating Point Systems
GUI	Graphical User Interface
HP	Hewlett-Packard
HRSC	High Resolution Stereo Camera
HTML	HyperText Markup Language
HW	HRSC/WAOSS
IBIS	Image Based Information System
IBM	International Business Machines
IEEE	Institute for Electrical and Electronic Engineers
I/O	Input/Output
JPEG	Joint Photographic Experts Group
JPL	Jet Propulsion Laboratory
K&R	Kernighan and Ritchie
MDMS	Multimission Data Management Subsystem
MIDR	Mosaicked Image Data Record
MIPL	Multimission Image Processing Laboratory
MIPS	Multimission Image Processing Subsystem
MSTP	Multimission Software Transition Project
NBB	Number of Bytes of Binary (prefix)
NFS	Network File System
NIMS	Near Infrared Mapping Spectrometer
NLB	Number of Lines of binary (header)
PCA	Performance and Coverage Analyzer
PDS	Planetary Data System
PVM	Parallel Virtual Machine

QIO	Queue Input/Output
RDM	Report Display Manager
RISC	Reduced Instruction Set Computer
RMS	Record Management Services
RPC	Remote Procedure Call
RTL	Run-Time Library (specific to VICAR in this document)
SAGE	Science Analysis Graphical Environment
SPARC	Scaleable Processor ARChitecture. Informally, a RISC microprocessor chip.
SPICE	Spacecraft, Planet, Instrument, C-matric, Events, a navigation tool, see: http://pds.jpl.nasa.gov/naif.html
SUBLIB	SUBroutine LIBrary (of VICAR)
TAE	Transportable Applications Environment
TBD	To Be Determined
TCL	TAE Command Language
VAX/VMS	Virtual Address eXtension/Virtual Memory System, a minicomputer operating system.
VFC	Vector Function Chainer
VICAR	Video Information Communication And Retrieval
VIDS	VICAR Interactive Display Subsystem
VRDI	Virtual Raster Display Interface
WAOSS	Wide Angle Optoelectronic Stereo Scanner
WWW	World-Wide Web

1.3 Data Types and Host Representations

Different host computers have different ways of representing data internally. Some machines are “big-endian”, meaning the high-order byte of an integer is stored first in memory, while others are “little-endian”, meaning the low-order byte is stored first .

Data that are to be transferred between these machines must be byte-swapped. Most machines use the IEEE floating point standard, but DIGITAL VAXes and Alphas running the VMS operating system have their own standard. Some of the IEEE-format machines are byte-swapped relative to each other. Data transferred between these machines must be converted as well.

1.3.1 VICAR File Representations

Conversion among hosts would be greatly simplified if all data were stored in ASCII instead of binary. However, that is inefficient in both time and space for image data. Image data must be stored in a binary representation. The question is, which one?

A standard, canonical representation could be chosen, such as Sun format: big-endian, IEEE floating point. That would simplify the file format, but would lead to inefficient operation on other

machines with different formats. Doing processing locally on a VAX, every pixel would be converted to Sun format every time it got read in or written out for every processing step. There wouldn't be enough coffee in the world to keep you awake while waiting. Due to the huge quantity of existing images written in VAX format, the canonical representation would have to be VAX format, which is not desirable in the long run.

Since most processing is done locally on one machine, and transfers between machine architectures are comparatively less frequent, the solution is to use the native format of whatever machine you are running on, and to identify that machine in the image label. That way, local operations are done efficiently, and conversion is done only when switching machines.

Applications *must* be able to do data format translations automatically. In order to ease the burden, the following conventions have been adopted:

- Applications shall be able to read files from any host representation.
- Applications shall normally write files in the native host representation of the machine on which they are currently running.

Placing the burden only on reading greatly simplifies the writing, while still insuring that the translations will take place in all cases. Some special-purpose applications may choose to write in a non-native format on occasion; however, *all applications must be able to read all formats, without exception*.

The Run-Time Library relieves most of this burden. When the standard I/O routines are called (**x/zvread** and **x/zvwrit**), the translations as stated above are performed automatically for the image data. The application merely calls **x/zvread** and it receives the data in the native format, ready for processing. It calls **x/zvwrit**, and the data is written out in the native format (which is what the buffer is in).

There are three cases where applications will have to do their own conversion:

- Binary labels: both headers and prefixes must be converted. See Using Binary Labels (page 14).
- Array I/O: Any program using Array I/O will get the data as it exists in the file, without any translation. Applications using Array I/O are responsible for doing their own data format translations on the data they read.
- Convert OFF: It is possible for an application to turn off the RTL's automatic conversion. This should not normally be done, but is available for special cases. If this option is selected, the application must do its own translation.

The **x/zvtrans** family of RTL routines are used to translate. Do *not* attempt to write your own data format conversion routines, even if you think it's only byte-swapping. Although at the present time byte-swapping is the only integer conversion, this may not always be the case. Other integer representations exist, such as one's-complement and sign-magnitude, that can not be translated by a simple byte swap. By having only one set of conversion routines, porting to a new platform with a different data format is easier. **x/zvtrans** translations are standardized, and thoroughly debugged. They are coded to be efficient, especially for simple byte-swapping.

1.4 Data Type Labels

VICAR uses system label items to keep track of the machine type the file (both image and binary label) was written on. These label items are summarized below.

- **HOST, string**: The type of computer used to generate the image. It is used only for documentation; the RTL uses the INTFMT and REALFMT label items to determine the presentation of the pixels in the file. Nevertheless, HOST should be kept consistent with INTFMT and REALFMT.
  See Table 1: Valid VICAR HOST Labels and Machine Types for a list of the currently valid host labels and the machine types they represent. New values for HOST will appear every time the RTL is ported to a new machine, so the table is not necessarily a complete list. Programs should not be surprised by values other than those in Table 1: Valid VICAR HOST Labels and Machine Types appearing in the label.

HOST label	Description of host machine
ALLIANT *	Alliant FX series computer
AXP-LINUX *	DIGITAL Alpha running Linux
AXP-UNIX *	DIGITAL Alpha running Unix (OSF/1)
AXP-VMS	DIGITAL Alpha running VMS
CRAY *	Cray (port is incomplete)
DECSTATN *	DECstation (any DEC MIPS-based RISC machine) running Ultrix
HP-700	HP 9000 Series 700 workstation
MAC-AUX *	Macintosh running A/UX
MAC-MPW *	Macintosh running native mode with Mac Programmers Workbench
SGI	Silicon Graphics workstation
SUN-SOLR	Sun SPARC machine running Solaris 2
SUN-3 *	Sun 3, any model
SUN-4 †	Sun 4, SPARCstation, or Sun clone running SunOS
TEK *	Tektronix workstation
VAX-VMS †	DIGITAL VAX running VMS
X86-LINUX	Intel x86 machine running Linux
X86-SOLR *	Intel x86 machine running Solaris 2

* Host machine is not officially supported
† No longer officially supported

Table 1: Valid VICAR HOST Labels and Machine Types

- **INTFMT, string**: The format used to represent integers in the file. INTFMT, REALFMT, and HOST should all match. If you change one change all three.
  The valid values of INTFMT may change as the RTL is ported to new machines. However, the currently valid values are listed. See Table 2: Valid VICAR Integer Formats.

INTFMT label	Description
LOW	Low byte first, “little endian”. Used for hosts VAX-VMS, AXP-VMS, X86-SOLR, DECSTATN, AXP-UNIX, AXP-LINUX, X86-LI
HIGH	High byte first, “big endian”. Used for all other hosts (except for CRAY, which has not been implemented).

Table 2: Valid VICAR Integer Formats

- **REALFMT, string**: The format used to represent floating point numbers in the file. INTFMT, REALFMT, and HOST should all match, so if you change one please change all three. Note: Compaq’s Alpha supports three floating point formats: D, G, and IEEE. D is used for compatibility with VAX, though G is the default. IEEE is used for Alphas running Unix, but not for VMS. The valid values of REALFMT may change as the RTL is ported to new machines. However, the currently valid values are listed. See Table 3: Valid VICAR Real Number Formats.

REALFMT label	Description
VAX	VAX format. Single precision is VAX F format, double precision is VAX D format. Used on hosts VAX-VMS, AXP-VM X86-SOLR, DECSTATN, AXP-UNIX, AXP-LINUX, X86-LINUX.
RIEEE	Reverse IEEE format. Just like IEEE, except the bytes ar reversed,with the exponent last. Used on hosts DECSTATN X86-SOLR only
IEEE	IEEE 754 format, with the high-order bytes (containing t exponent) first. Used for all other hosts (except for th CRAY, which has not been implemented)

Table 3: Valid VICAR Real Number Formats

- **BHOST, string** :The type of computer used to generate the binary label. It is used only for documentation; the RTL uses the BINTFMT and INTFMT label items to determine the representation of the binary labels in the file. Nevertheless, it should be kept consistent with BINTFMT and BREALFMT.
  The valid values of INTFMT are exactly the same as for the HOST label item.
- **BINTFMT, string**: The format used to represent integers in the binary label. BINTFMT, INTFMT, and INTFMT should all match, so if you change one please change all three.
  The valid values of BINTFMT are exactly the same as for the INTFMT item above.
- **BREALFMT, string**: The format used to represent floating point numbers in the binary label. BINTFMT, INTFMT, and INTFMT should all match, so if you change one please change all three.
  The valid values of BREALFMT are exactly the same as for the REALFMT label item above.
- **BLTYPE, string:** The type of the binary label. This is not type in the sense of data type, but a string identifying the kind of binary label in the file. The RTL does not do any interpretation or checking of BLTYPE. It is intended for documentation, so people looking at the image will know what kind of

binary label is present. It may also be used by application programs to make sure they can process the given type of binary label, or to make sure it is processed correctly.
Valid BLTYPE values are maintained in a name registry, so that all possible kinds of binary labels can be documented in one place. Only names that are registered should be used in BLTYPE. See 1.7.2 Programming and Binary Labels (page 16) for more details.

- **FORMAT**, string**:** The data type of the pixels in the file:
  - **BYTE**: Single byte, unsigned integer pixel type.
  - **HALF**: Signed short integer pixel type (often 2 bytes).
  - **FULL**: Standard size signed integer pixel type (often 4 bytes).
  - **REAL**: Single precision floating-point pixel type.
  - **DOUB**: Double precision floating-point pixel type.
  - **COMP**: Two single precision floating-point numbers representing a complex pixel type, in the order (real, imaginary).

The label item **FORMAT** is used to specify the data type. It is unfortunate that the name FORMAT is used for the data *type*, rather than for the host representation (which would better be called *format*), but the names cannot be changed for historical reasons.

1.5 Pixel Type Declarations

All pixel buffers in an application must be declared using the standard pixel type declarations. See Table 6: FORTRAN declarations for pixel types (page 25) for the standard FORTRAN declarations, and See Table 4: C Declarations for Pixel Types (page 23) for the standard C declarations.

1.5.1 Pixel Sizes

Don't assume that FULL (integer) is 4 bytes, DOUB is 8 bytes, HALF is 2 bytes, etc.. A given machine may have data types of different sizes, such as 64-bit (8 byte) integers.

Pixel sizes must be determined both for files and internal program buffers (arrays). A file could be written on a machine with pixel sizes different from the machine it will be read on. Use one of the RTL pixel size routines, **x/zvpixsize**, **x/zvpixsizeu**, or **x/zvpixsizeb** to determine the size as written. These routines return the size of a pixel in bytes, given the data type and machine formats. **x/zvpixsizeu** and **x/zvpixsizeb** retrieve the machine formats from an image label.

Buffers used within a program must also be sized correctly. Since internal buffers are almost always in native format for the machine you are running on, you can use the C function `sizeof()` to get the size of an element in bytes. Use the proper data type for the pixel. See Table 4: C Declarations for Pixel Types (page 23). FORTRAN has no `sizeof()` operator or equivalent; use **x/zvpixsize** with machine formats of "NATIVE" or "LOCAL" instead.

1.6 Converting Data Types & Hosts

This section describes how to convert data between different data types and hosts, when the RTL does not do it for you. Most of the time, the RTL will take care of any data type and host conversions automatically. There are times, however, when you will need to do your own conversions.

The translation routines have two parts: the setup routines, and the actual translation routine. The

setup routines must be called first, to create a user-supplied buffer that describes the translation. The translation routine may then be called as many times as necessary to do the actual translation.

It is important to have as many translation buffers active at the same time as needed. They can be created first, then used when needed in the program. This is illustrated in the example below.

There is only one setup routine internally, which requires the data type, integer format, and real format for both the source and the destination. Since it is unwieldy to specify all six parameters every time something needs translating, there are five setup routines that provide part of the information for you. They are all simply syntactic sugar for the internal setup routine. These five are **x/zvtrans_set**, **x/zvtrans_in**, **x/zvtrans_inu**, **x/zvtrans_inb**, and **x/zvtrans_out**. See Translation Routines (page 93).

Where do these six parameters come from? Either the source or the destination will be in the native host representation, so only the foreign machine need be specified. For **x/ztvtrans_set**, both source and destination are the local machine. The information for the foreign machine will usually come from the label of the file being read. This is made easier by the **x/zvtrans_inu** routine. For binary labels (which may be from a different host type than the image), **x/zvtrans_inb** can be used to get the binary label host information.

You may make use of the **x/zvhost** routine to get the INTFMT and REALFMT for a machine, given the type of machine. This allows the user to specify that the file be written in a foreign format. Although this is not the usual mode of operation for VICAR, it is allowed. An example should help to clarify things. A file is being read which contains a structure of mixed data type. The structure needs to be converted to native format so it can be processed. The structure could come from the binary label of the file, or it could be the data in an old-style IBIS file. It doesn't even have to be a VICAR file the translation routines don't care. The code below is partially C code and partially pseudocode:

```
/* Structure format:  An int followed by two reals and an array of 8 shorts */

struct {
   int type;
   float coord[2];
   short int values[8];
} data;
unsigned char *old_ptr;
static int short_conv[12], int_conv[12], real_conv[12]; /* Translation bufs */
static int short_size, int_size, real_size;    /* Pixel sizes for each type */

{    /* Begin here */

   char intfmt[20], realfmt[20];
   unsigned char input_buf[100];

   zveaction("sa", "");                        /* abort on error */

   /*** Determine INTFMT and REALFMT, possibly via zvhost. ***/
   /*** For a VICAR file, zvtrans_inu may be used instead. ***/
   /*** For binary labels, zvtrans_inb may be used.        ***/

/* Now set up the translation buffers. */

   zvtrans_in(short_conv, "HALF", "HALF", intfmt, realfmt);
   zvtrans_in(int_conv,   "FULL", "FULL", intfmt, realfmt);
   zvtrans_in(real_conv,  "REAL", "REAL", intfmt, realfmt);

/* Get pixel sizes in the input file for each type */

   zvpixsize(&short_size, "HALF", intfmt, realfmt);
   zvpixsize(&int_size,   "FULL", intfmt, realfmt);
   zvpixsize(&real_size,  "REAL", intfmt, realfmt);

/* The following could be in a read loop if desired. */

   /*** Now the buffers are set up.  Read the data into input_buf. ***/

   old_ptr = input_buf;

   zvtrans(int_conv, old_ptr, &data.type, 1);           /* One integer */
   old_ptr += int_size;
   zvtrans(real_conv, old_ptr, data.coord, 2);          /* Two reals */
   old_ptr += real_size*2;
   zvtrans(short_conv, old_ptr, data.values, 8);        /* 8 shorts */
   old_ptr += short_size*8;

}    /* That's all!  The data has been translated. */
```

1.7 Using Binary Labels

The use of binary labels are not portable across computer architectures. Property labels should be

used instead. They serve the same function as binary headers, though possibly not binary prefixes. Binary labels are allowed, but before creating a new format, stop to consider if property labels might be a better approach. See 4.1.2.1 Using Property Labels (page 67), for details.

Binary labels are application-defined extensions to a VICAR image used to store information about the image. They have two parts: binary headers: extra records at the beginning of the image, and binary prefixes: extra bytes at the beginning of each image record. Binary labels are not part of image data. An application will never see the binary label data unless it specifically asks for it via a COND BINARY optional to **x/zvopen**.

Binary headers are extra records that appear at the beginning of the file, between the standard label and the image. The number of records is specified by the NLB (Number of Lines of Binary) system label item. Binary headers are often thought of as extra "lines" of data, but depending on the file organization they can actually be extra lines, samples, or bands. The size of each binary header record is exactly the same as the size of each image record. The headers specified by NLB occur exactly once in the file, not once per band (BSQ organization) or once per line (BIL or BIP organization).

Binary prefixes are extra bytes that appear at the beginning of each image record. The number of bytes is specified by the NBB (Number of Bytes of Binary) system label item. The image record consists of NBB bytes of binary prefix data, followed by the samples that make up one line (for BSQ organization). Other file organizations label the units differently: a record for BIL is NBB plus the samples that make up one band, while a record for BIP is NBB plus the bands that make up one sample. NBB is specified in terms of bytes, *not* in terms of pixels, even if the pixels are larger than one byte.

Binary labels (both headers and prefixes) pose a problem: the data stored in them is application-defined. The RTL can't determine what types of data are stored in a binary label, and therefore cannot automatically convert the data to the native host format as it does for image data. The application program has sole responsibility for converting binary data to the native host format when it is read.

Few application programs understand any given kind of binary label. For example, the Voyager project has a definition for what goes in the binary label of its images. The Voyager-specific processing programs know how to interpret these labels and make use of them. The Galileo project also has a definition for its binary labels. Galileo-specific programs can interpret the Galileo binary labels. However, the two kinds of binary labels are different, and programs written for one cannot make use of the other. The Magellan project has its own binary labels in yet another format.

The variety of binary labels poses a problem for a general-purpose application. They could certainly be ignored, but then the information would be lost. Therefore, many general-purpose programs copy the binary labels from the input to the output, thereby preserving the information. As long as the information is only copied, not used, the application need not know any details.

But what happens when you mix machine types? A file with binary labels was written on a VAX. You want to run COPY on the file from a Sun. COPY reads the input image, and writes the output image. The RTL automatically converts the image data from VAX to Sun format on input, and so the file gets written in Sun format. The system labels also say it is in Sun format. COPY also reads the binary labels, and writes them to the output file. The binary labels cannot be converted, as neither the RTL nor the COPY program know what data types are in the binary labels. Therefore, the binary labels get written out in the only way possible: in VAX format.

The problem is that the system labels say the file is in Sun format, but the binary labels are still in VAX format.

1.7.1 Separate Host Types

The solution to the binary label problem is *separate* host formats for the image and the binary label. The system labels HOST, INTFMT, and REALFMT describe the host formats for the image host type, while the system labels BHOST, BINTFMT, and BREALFMT describe the host formats for the binary .

It is possible to generate files that have data in two different host formats: one for the image itself and one for the binary labels. This is not particularly desirable, but there is no practical alternative. As long as applications make sure they use the binary label host formats while accessing the binary labels, there's no problem. However, this does place a burden on application programmers to make sure they access binary labels correctly.

UPDATE mode changes a file in place (rather than by copying) but does not convert it to native format. The application must write out any binary label updates in the format of the file, or it must read and re-write the entire binary label in a native format. The RTL handles the conversion automatically for image data.

A set of subroutines should be written for each type of binary label to read/write/update that label. If all applications used this set of subroutines, it wouldn't matter what format the binary labels were kept in. The subroutines would be able to adapt and hide the details from the application program. If changes were made to the binary labels, or even if they were converted to property labels, the only code that would need to change would be the subroutines that access them.

1.7.2 Programming and Binary Labels

This Section shows how application programs make use of the binary label support features provided by the RTL.

The use of binary labels is discouraged, due to portability problems. Where possible, property labels should be used instead. They usually serve the same function as binary headers, though possibly not binary prefixes. Binary labels are allowed, but before creating a new format, stop to consider if property labels might be a better approach. See 4.1.2.1 Using Property Labels (page 67), for details.

The first problem is to open the file. Opening a file for input, update, and output will be covered separately, as the behavior is slightly different. All modes make use of **x/zvadd** and **x/zvopen** to open the files. The optional arguments mentioned may be used with either routine.

For an input file, binary label host formats are obtained from the BHOST, BINTFMT, and BREALFMT system labels in the input file. The host formats thus obtained are used for the **x/zvpixsizeb** and **x/zvtrans_in** routines, as well as for **x/zvget**. The labels in the file are assumed to be correct, so there is no real override. If you need an override, you can always use the host formats directly through **x/zvpixsize** or **x/zvtrans_in**. If the input file does not have the BHOST, etc. labels (for older images), then VAX format is assumed. If the input file has no label at all (COND NOLABELS), then the binary host formats are obtained from the BHOST, BINTFMT, and BREALFMT optional arguments to **x/zvadd** or **x/zvopen**.

For an update file, binary label host formats are also obtained from the file. The only difference is if you wish to change the binary label type of the file. You would need to re-write all the binary labels in the new format (be careful because the size might change), but you would also need to change the binary label format system label items (BHOST, BINTFMT, and BREALFMT) via **x/zladd**. If you use **x/zladd** to change these items, then do *not* use **x/zvget** on them, or use **x/zvpixsizeb** or **x/zvtrans_in** at all, as these routines may still use the original binary label format. The **x/zlget** routine would get the

new values.

The BLTYPE optional argument is not used for input or update files. You may change the BLTYPE system label on an update file via **x/zladd**, however. See the next section for a description of BLTYPE.

Output files are more complex. There are two basic cases: either you convert the binary labels to native format, or you leave them alone. Which option you choose is controlled by the BIN_CVT optional argument.

If you set BIN_CVT to ON, write the binary labels in the native host format. The BHOST, BINTFMT, and BREALFMT system label items are set automatically to the native host formats. There is no override; the corresponding optional arguments are disabled when BIN_CVT is ON.

Turn BIN_CVT ON if you are writing a new file with binary labels in native format. Although the name implies "convert", it applies to any binary labels written in native format. If BIN_CVT is ON, then the application must know the kind of binary label being written. To do this, set the BLTYPE label using the BLTYPE optional argument. See the next section for a description of BLTYPE.

If BIN_CVT is OFF (default), then the assumption is that you are copying the binary label without converting it. The binary label host formats used are those of the primary input file. If the primary input is not available, the default is VAX format. The primary input may be changed with **x/zvselpi**.

BIN_CVT OFF is the appropriate mode to use for general-purpose applications that do not know the format of the binary label. The BHOST, BINTFMT, and BREALFMT optional arguments to **x/zvadd** and **x/zvopen** will override any other settings, so you could write binary labels in an arbitrary format by setting to OFF and setting the three optional arguments to appropriate values.

If you need to write specifically in VAX format, then do not depend on the default being VAX format (as mentioned above). That may not be reliable, depending on the primary input. If you want to write in a specific format, specify that format in the optional arguments. Do not use **x/zladd** to change the binary label system label items on an output file. Specify them when you open the file.

The BLTYPE optional argument is active if is OFF. Set it if you know the kind of binary label being written. BLTYPE comes from the primary input if you do not specify it. See the next section for a description of BLTYPE.

Once a file is open, translate the binary label data you read and/or write to/from native format. The actual translations are performed in the same way as described in the rest of . Section Data Types and Host Representations (page 8), except using the BHOST, BINTFMT, and BREALFMT labels and the **x/zvpixsizeb** and **x/zvtrans_inb** routines.

As mentioned previously, *all* binary label data read in must be converted to native format before being used. There are no exceptions to this rule.

Binary labels written out to a file may be in either native or foreign formats, as discussed previously.

1.7.3 Binary Label Types

BLTYPE improves the documentation of files and makes the VICAR system more robust. Everyone using binary labels is encouraged to make use of BLTYPE.

BLTYPE is a string that describes the kind of binary label in the file. It may be set with **x/zvadd** or **x/zvopen**, or by calling **x/zladd** on an already open file. No error checking or range of valid values

are enforced by the RTL.

The BLTYPE should be a short (maximum 32 characters) string that names the binary label type. It does not attempt to describe the fields in the binary label at all, but merely provides a pointer to how the fields could be determined. Some of the currently registered names are “GLL_SSI_EDR”, “IBIS”, and “M94_HRSC”.

To maintain a consistent set of names, a name registry (similar to the one for properties) has been established for BLTYPE. Possible values for BLTYPE must be entered into this registry, with a pointer to documentation describing the format of the binary label. To create a new kind of binary label, tell the keeper of the registry what name you want to use and the format, either explicitly or by stating that it is in document *X*. The registrar will check for duplicates, approve your request and enter your name into the registry.

The keeper of this registry is the VICAR system programmer.

The RTL makes no checks on the validity of the names used. It is the responsibility of each individual programmer to make sure that they use this system. Failure to do so may result in your program not being accepted for delivery.

Application programs that expect a certain kind of binary label should check the value of BLTYPE to make sure that they have been given the correct type, and issue an error message if it is incorrect. If BLTYPE is not present or blank, assume that the binary label is of the correct type for backwards compatibility.

More sophisticated application programs could use the BLTYPE field to read several different kinds of binary labels. These could be different versions of the same basic binary label (e. g. a type of “X V2. 0” for version 2 of the X label type), or labels from different projects. Widespread use of BLTYPE will allow a general-purpose program to be written that understands most binary label types.

2. Programming Practice

2.1 General VICAR conventions

- **Portability**. All VICAR software runs on all platforms that MIPL supports.
- **Separate FORTRAN and C Interfaces**. All subroutines have separate FORTRAN and C calling sequences, with different names. Subroutines call only the appropriate language interface.
- **Status return.** The status code is the function return. If automatic error checking is on (via **x/zveaction** or one of the "*_ACT" key words), then status return may be ignored.
- **Pass by value.** All input values to the RTL are passed by value as in standard C, rather than by reference. Output variables or arrays are passed by reference. Check to remove "`&` "'s before input arguments.
- **Data Formats and I/O**.
  - Applications read files written in any host representation. See 1.3.1 VICAR File Representations (page 8).
  - Applications write files in the native host representations of the machine on which they are currently running. See 1.3.1 VICAR File Representations (page 8).
  - Programs reading binary labels read any host format, and convert to native format before using. See 1.3 Data Types and Host Representations (page 8) and 1.7 Using Binary Labels (page 14).
- **Data Format Translation.** Do not write your own pixel data type or host representation conversion routines. The routine x/zvtrans is the standard way to translate data between different host representations and pixel data types in VICAR. See VICAR File Representations (page 8).
- **Do not use EQUIVALENCE** for data type conversion in FORTRAN. Use the `INT2BYTE` or `BYTE2INT` routines. See 2.7.2 No EQUIVALENCE for Type Conversion (page 34).
- **Pixel Sizes.** Do not assume the size in bytes of a pixel or other data; it may be different on other machines. Use one of the **x/zvpixsize** routines, or `size of` in C, to determine the size of a data element. See 1.5 Pixel Type Declarations (page 12).
- **Name Registries.** Every property name or BLTYPE name used is entered into the appropriate name registry. See 4.1.2.1 Using Property Labels (page 67) and 1.7.2 Programming and Binary Labels (page 16).
- **Operating System Calls.** All OS-specific code is eliminated or isolated, and should be written in C if possible. See 2.7.6 VMS-Specific Code (page 36).
- **Conditional Compilation.** Conditional compilation statements to handle machine dependencies use names of specific features that are defined in standard include files based on the machine type. Feature dependencies are defined only in `x/zvmaininc.h` or `vmachdep.h`. Symbols VMS_OS and UNIX_OS distinguish between VMS and UNIX operating systems.
- **No Terminal I/O.** VICAR applications may not write directly to the terminal (e. g. stdout). Use

x/zvmessage instead.

- **ANSI C.** Function prototype include files use `#ifndef` wrappers, _NO_PROTO, and extern "C" declarations. FORTRAN bridge routines do not use prototypes or the prototype form of declaration.
- **FORTRAN 77 standard.** All FORTRAN code conforms to the ANSI FORTRAN -77 standard, with the exception of the allowed extensions listed in the reference. See 2.7.5 VMS FORTRAN Extensions (page 36).
- **FORTRAN Strings.** All FORTRAN-callable subroutines written in C use the RTL string conversion routines (**sfor2c** *et al*) to handle CHARACTER*n arguments. See 2.6.4 Passing Strings (page 29).
- **vimake.** VICAR applications use imakefiles compatible with vimake to create their build files.
- **vpack.** VICAR applications are packed into a COM file using the vpack command.
- **Optional Arguments**

There are two types of RTL routines: those that take a constant number of arguments, and those that take optional arguments of the form "key word", "value", "key word"," value", etc.

All routines that accept keyword-value pairs, whether or not the pairs are used in any particular call, must have an argument list terminator. The terminator is a language-dependent argument at the end of the argument list, where the next key word would be if it existed.

In FORTRAN, the terminator is a single blank in quotes at the end of the argument list:

```
call xvopen(unit, status, 'op', 'write','open_act','sa',' ')
call xvread(unit, buffer, status, 'line', 1, ' ')
call xvclose(unit, status, '')
```

In C, the terminator is a 0 or NULL argument (*not* a blank):

```
status = zvopen(unit, "op", "write", "open_act", "sa", 0);
status = zvread(unit, buffer, "line", 1, 0);
status = zvclose(unit, 0);
```

Certain RTL routines used to take pure optional arguments (not keyword/value), but that practice is not portable and is now obsolete.

2.2 ANSI C

ANSI C is used for all VICAR programs. This section describes what needs to be done within the VICAR environment in order to successfully use ANSI C. Read it before using ANSI C with VICAR.

Function prototypes allow the compiler to check the number and types of arguments to subroutines. If they don't match, a warning is issued. This feature will catch many common programming errors. You will undoubtedly find many more compiler warnings if you use prototypes, but that is a good thing. Most of those warnings are errors in the code, or questionable coding practices. Take heed and your code quality should improve.

A function with a prototype does not suffer default argument promotion; it can, for example, pass a float as a float, rather than promoting it to a double. This can be more efficient in some cases. This is also a danger, however; if a subroutine is intended to be called from non-ANSI code (which includes almost all SUBLIB routines), then be careful to use only argument types that won't be promoted in the absence of a prototype in order to remain compatible. Any ANSI C reference should be able to provide

more details.

Program writers should be careful to include the appropriate include files, in order to get the function prototypes for each subroutine package. The prototype file for the RTL is called “zvproto.h”. You should include this file in any routines that make use of the RTL.

Subroutine writers should provide an include file that defines the prototypes for the functions in your subroutine package. This include file may be very short if you have only one function, or quite long for a big package. This include file should generally be the same as that containing public definitions, although it can be separate if need be. For example, the RTL include file, ”zvproto.h”, has only prototypes.

When you declare prototypes, there are three things you need to do: provide a wrapper, use _NO_PROT O, and support C++. The first thing is to protect your file against multiple inclusions. A “wrapper” define and #ifndef should be put around the entire file. This way, if the file is included more than once, it will not be compiled the second time.

Second, your include file may be used in a non-ANSI environment. For this reason, it is important to protect the prototypes with the _NO_PROTO symbol. The _NO_PROTO symbol is declared in `xvmaininc.h`. If _NO_PROTO is not defined, use prototypes. If it is, use the old-style declarations instead.

Third, allow support of C++. C++ uses the same prototype mechanism as ANSI C, but prototypes are required. In order to cross languages and link to a C module, however, the C declaration must be enclosed in “`extern “C” {...}` “. By putting this wrapper around your entire include file, and providing prototypes for every function, you can make your subroutines available from C++.

Here is an example which demonstrates all three techniques. Use it as a template for prototype includes.

```
/* xyz.h: include file for the XYZ subroutine package. */
#ifndef _XYZ_H
#define _XYZ_H

#include "xvmaininc.h"     /* not needed if you can guarantee the
caller */
                           /* has already included it */

#ifdef __cplusplus
extern "C" {
#endif

#define ONE_OF_MY_FLAGS 10            /* your common definitions go
here */

#ifndef _NO_PROTO

double xyz(int a, char *b, double d); /* your prototypes go here */

#else /* _NO_PROTO */

double xyz();                         /* non-prototype decls go here */

#endif /* _NO_PROTO */

#ifdef __cplusplus
}        /* end extern "C" */
#endif

#endif /* XYZ_H */
```

There are some pitfalls using ANSI C prototypes. First, do *not* put prototypes on any FORTRAN bridge routines. FORTRAN doesn't understand them, and the prototypes interfere with the FORTRAN string passing mechanisms. Use the old-style declarations instead. Prototypes can and should exist for the C function called *by* the bridge.

Second, variadic functions (those with variable arguments) are a problem if you are intending to have both ANSI and non-ANSI callers. The ANSI C standard states that variadic functions must use "`...`" in the prototype and function header. The implication is that compiler is free to use a different argument-passing mechanism for these functions (although current compilers don't seem to do this).

The old form of using varying arguments, <varargs.h> doesn't have the risk of a different argument-passing mechanism. Plus, if a variadic function is going to be called by non-ANSI C code, the..." form is not available. For this reason, and to be safe, you should use <varargs.h> instead of <stdarg.h>, and not the ..." form. See the comments in zvproto.h for more details. This should not cause much of a problem in VICAR code because varying arguments may only be used in certain limited situations.

2.3 C Calling Sequence

All RTL routines have a C language binding: every routine has an entry point that uses C-style arguments and calling sequences. All portable C language programs use the C language interface to the RTL.

The C-language routines are named similarly to the old (now FORTRAN-style) routines, but start with a "z" instead of "x". For example, instead of calling **xvwrit** or **xlget**, a C program would now call **zvwrit** or **zlget**.

2.3.1 C Data Types

The standard C definitions for each of the VICAR pixel data types are listed. See Table 4: C Declarations for Pixel Types (page 23). The data types that may be passed into the RTL, and the C declarations for each type, are listed in Table 5: C Declarations for Run-Time Library Arguments.(below)

Pixel Type	C Declaration
BYTE	unsigned char
HALF	short int
FULL	int
REAL	float
DOUB	double
COMP	struct complex {float r, i; \};

Table 4: C Declarations for Pixel Types

RTL Argument Type	C Declaration
integer	int
string	char x[n]
value	int, char x[n], float, or double
integer array	int x[m]
string array	char x[m][n]
value array	array of any " value" type above
pixel buffer	pointer to any pixel type. See Table 4: C Declarations for Pixel Types (page 23)
FORTRAN string	char *x
FORTRAN string array	char *x
void pointer	void *x (actually, a pointer to anything)
pixel pointer	address of pointer to any pixel type. See Table 4: C Declarations for Pixel Types (page 23)
pointer to string array	char **x

Table 5: C Declarations for Run-Time Library Arguments

Because labels and parameters can have more than one value, the routines that deal with them must be able to accept (or output) arrays of strings. The RTL expects all arrays of strings to be two-dimensional arrays of "char". This does not mean an array of pointers to strings, but a pointer to an area of memory n*m bytes long, where n is the maximum size of each string and m is the number of strings. The inner dimension, n, must be passed in to the routine so it knows how to access the string array.

All RTL routines that use string arrays have a way to specify the length. The length is the size of the inner dimension; it is not the length of any particular string, but must include space for the null terminator at the end of a string. For example, if a string has at most 10 characters, make sure the length is at least 11 to allow room for the null terminator.

2.4 FORTRAN Calling Sequence

The Run-Time Library has two entry points (called language bindings) for every routine. One is designed to be called from FORTRAN only, and the other is designed to be called from C only. The FORTRAN routines start with "x", as in **xvread**, **xladd**, etc.

2.4.1 Character Strings

Most RTL routines take characters strings as arguments. They appear in parameter names, key words, messages, and many other places. Under the old RTL, strings could be passed in either of two ways: as FORTRAN CHARACTER*n variables, or as BYTE or LOGICAL*1 arrays. The array method will no longer work.

All FORTRAN-callable RTL routines accept character strings *only* in FORTRAN CHARACTER*n format. CHARACTER*n variables and constants have a length associated with them (the "n"), which the RTL uses to find the end of the string, and to make sure that buffer overflow does not occur on output strings. Byte arrays can't be used, since in FORTRAN there is no way for the RTL to find the end of the string. Arrays of CHARACTER*1 will not work; the RTL will think the string being passed in is one character long.

2.4.2 FORTRAN Data Types

The standard FORTRAN definitions for each of the VICAR pixel data types are listed below in Table 6: FORTRAN declarations for pixel types. The data types that may be passed in to the RTL, and the FORTRAN declarations for each type, are listed below in Table 7: FORTRAN declarations for Run-Time Library arguments.

Pixel Type	FORTRAN Declaration
BYTE	BYTE
HALF	INTEGER*2
FULL	INTEGER*4
REAL	REAL*4
DOUB	REAL*8
COMP	COMPLEX*8

Note the use of the "*n" form, even where not strictly necessary. This is intentional; please use the form shown for all pixel data.

Table 6: FORTRAN declarations for pixel types

RTL Argument Type	FORTRAN Declaration
integer	INTEGER
string	CHARACTER*n
value	INTEGER, CHARACTER*n, REAL, or DOUBLE PRECISION
integer array	INTEGER x(m)
string array	CHARACTER*n x(m)
value array	array of any "value" type above
pixel buffer	array of any pixel type. See Table 6: FORTRAN declarations for pixel types (page 25)
pixel pointer	N/A

The "*n" is not used for arguments, except for strings. This helps to distinguish pixel data (which uses "*n") from other types of data.

Table 7: FORTRAN declarations for Run-Time Library arguments

2.5 Include Files

There are five major classes of include files: system, VICAR main, VICAR system, VICAR subroutine, and local.

Include only those files needed. Some includes require that other files be included first. Extra includes in your program don't gain you anything, but slow down your compiles. They also make system maintenance difficult, since the source code must occasionally be searched to see what programs use an include, and all programs that use an include must be recompiled when it changes (even if it's not really used).

System include files are provided by the operating system. Many are OS-specific, but there are a

few that are standard across platforms. Make sure the ones you need are available in both operating systems, or isolate them in machine-dependent code. System include files should be enclosed in angle brackets and have the ".h" extension. Some known portable includes are:

```
#include <varargs.h>
#include <math.h>
#include <ctype.h>
#include <stdio.h>
```

One known *non*-portable file is `<unistd.h>`. If you need symbolic constants for the lseek() arguments(which are often contained in `unistd.h`), use the definitions provided in xvmaininc.h.

The VICAR main include file, which all VICAR programs should start with, must be in lower case, with double quotes. The name of the main include is "vicmain_c", with no directory specifiers and no ".h" extension.

Any modules other than the main program module that need VICAR definitions should include `xvmaininc.h`, in lower case, with double quotes and the ".h" extension xvmaininc.h is automatically included with vicmain_c.

```
#include "vicmain_c"
```

—or —

```
#include "xvmaininc.h"
```

VICAR system includes are provided as part of the RTL. Their names should be in lower case, with quotes and the ".h" extension. The valid ones are listed below. Only a few of these should ever be used in application code, notably zvproto.h, ftnbridge.h, errdefs.h, and applic.h. The rest should only be needed in very unusual circumstances. They should generally be in the order listed below.

Include *only* those needed. Extraneous include files will only slow down your compile and make system maintenance harder. TAE includes are handled as specified in the TAE documentation. They should be in double quotes, with the ".inc" or ".inp" extensions as appropriate. The first four listed below are TAE includes:

```
#include "taeconf.inp"
#include "symtab.inc"
#include "parblk.inc"
#include "pgminc.inc"
#include "zvproto.h"
#include "defines.h"
#include "declares.h"
#include "externs.h"
#include "applic.h"
#include "errdefs.h"
#include "ftnbridge.h"
#include "xviodefs.h"
```

VICAR subroutine include files (class 2 and 3 SUBLIB, VRDI, etc.) should be in double quotes, with no path names, and should include the ".h" extension, like VICAR system includes. For these includes, the directories containing the includes must be available to the compiler. This is normally handled transparently, but some libraries will require a LIB_* macro in the imakefile to access the includes.

```
#include "vmachdep.h"
#include "gll_ssi_edr.h"
#include "xderrors.h"
```

Local includes, which are delivered as part of the COM file and are used only by that application, should be in double quotes, with no path names. Include the ".h" extension, just as for VICAR system and subroutine includes. Local includes must be listed in the INCLUDE_LIST macro in the imakefile so they can be cleaned up properly.

```
#include "my_inc.h"
```

2.6 Mixing FORTRAN and C

This section describes how to mix code written in both FORTRAN and C. It covers both internal code (used only within one program), and SUBLIB subroutines that must be called from either language.

A subroutine must have a separate interface for each language. A FORTRAN program can only call the FORTRAN interface, while a C program can only call the C interface, because the necessary calling conventions are so different. If you are rewriting a subroutine which is used only internally to your program, you may need an interface for only one language. Some SUBLIB routines are useful only for one language, but most subroutines will have two interfaces.

2.6.1 Bridge Routines

The calling interface for the language of subroutine are straightforward. Follow the rules for that language. The interface for the other language will be implemented via a "bridge" routine, which must be written in C.

The bridge routine accepts arguments in the format of the opposite language, converts them to the form of the subroutine's language and calls the main subroutine. For example, a subroutine written in C would have a bridge also written in C that accepts arguments in the FORTRAN style, reformat them, and call the main C subroutine. A subroutine written in FORTRAN would have a C bridge that accepts C-style arguments, and call the FORTRAN subroutine in the proper manner.

If you are using ANSI C, do not use function prototypes on FORTRAN bridges (routines written in C that are intended to be called from FORTRAN).

2.6.2 Naming Subroutines

The subroutine names for the FORTRAN and C interfaces *must* be different. It is not sufficient to add an underscore to the end of a name; the letters themselves must be different. The name difference is forced by the fact that some FORTRAN compilers put a trailing underscore after the name, and others do not.

In order for a C routine to be called from FORTRAN, it must be named in a manner the FORTRAN compiler will recognize. This is handled via the FTN_NAME macro. This macro is used any where the subroutine name would be, including in the declaration of the subroutine. The subroutine name is the argument for the macro; "ftnbridge.h" must be included:

```
#include "xvmaininc.h"
#include "ftnbridge.h"

void FTN_NAME(mysub)(param1, param2)
int *param1, *param2;                          /* inputs */
{
   zmysub(*param1, *param2);
}
```

would be called by FORTRAN:

```
      integer a, b
      call mysub(a, b)
```

A FORTRAN routine that wishes to be called by C must have a bridge written in C. Otherwise, the C caller would have to use the FTN_NAME macro to call the routine, which would cause problems if the routine were ever converted to C. The bridge routine takes care of this:

```
subroutine fsub(param1, param2)
      integer param1, param2                   ! inputs
C     do something
      return
```

The C bridge looks like the following. Note the use of FTN_NAME when calling the FORTRAN program:

```
#include "xvmaininc.h"
#include "ftnbridge.h"

void csub(param1, param2)
int param1, param2;                            /* inputs */
{
   FTN_NAME(fsub)(&param1, &param2);
}
```

It would be called from a C program:

```
int a, b;
csub(a, b);
```

2.6.3 Passing Numeric Arguments

Passing numeric arguments and arrays between FORTRAN and C is straightforward. The data type equivalencies are listed. See Table 7: FORTRAN declarations for Run-Time Library arguments (page 25). A routine that receives data of one type must be passed the equivalent type of data if called from the other language. The bridge may change the datatype between the caller and the routine if desired, but the subroutine interface that actually spans the language change (caller-bridge for FORTRAN to C, bridge-routine for C to FORTRAN) must follow this table.

For numeric arguments FORTRAN passes arguments by reference, while C normally passes input arguments by value. Such arguments must be converted in the bridge routine. A FORTRAN to C bridge declares all arguments as pointers, then puts asterisks (*) in front of the appropriate arguments to dereference them when calling the main routine. A C to FORTRAN bridge declares the appropriate arguments as values (not pointers), and uses an ampersand (&) in the call to the FORTRAN routine to convert them to pointers. There are examples in 2.6.2 Naming Subroutines (page 27).

Not all C arguments are passed by value. If a pointer comes in, such as for an output variable or an array, then the pointer can normally be passed unchanged to the subroutine.

FORTRAN Data Type	C Data Type
BYTE	unsigned char
INTEGER*2	short int
INTEGER*4	int
REAL*4	float
REAL*8	double
COMPLEX*8	struct complex {float r, i; }
INTEGER	int
CHARACTER*n	char x[n] *
INTEGER x(m)	int x[m]
REAL x(m)	float x[m]
CHARACTER*n x(m)	char x[m][n] *

* Characters require special handling, see 2.6.4 Passing Strings (page 29)

Table 8: FORTRAN and C Argument Data Type Equivalence

Two-dimensional (or higher) arrays are handled differently in each language. FORTRAN accesses them in column-major order, while C uses row-major order. The order of the subscripts is reversed. One can either copy the array and reshuffling it in the bridge (which can be inefficient), or simply document the need to reverse the subscripts in the other language when calling the routine.

2.6.4 Passing Strings

Passing FORTRAN strings is not standardized among compilers. The RTL currently supports six different methods of passing FORTRAN strings in arguments. The details of the six methods are hidden in the RTL FORTRAN String Conversion Routines, so they all look the same to the application programmer.

Code that passes a string to or from FORTRAN uses the RTL string conversion routines (**sfor2c** and **sc2for** family). Code that calls the conversion routines must set the FTN_STRING flag in the imakefile.

2.6.4.1 Accepting FORTRAN Strings in C

Writing a C routine that accepts FORTRAN strings as arguments is easy. Use **sfor2c** for input from a FORTRAN program and **sc2for** for output to a FORTRAN program.

The **sfor2c** and **sc2for** family of routines are described, with examples, in

Typically, a FORTRAN to C bridge routine will first call some of the **sfor2c** routines to convert all the input strings to C format, then it will call the main C subroutine, and finally it will convert any output strings back to FORTRAN format via the **sc2for** family of routines.

There is no way currently implemented to return CHARACTER*n variables as the function return of a C subroutine. Any output strings should be included as arguments instead.

2.6.4.2 Accepting C Strings in FORTRAN

The easiest way to accept C strings in a FORTRAN subroutine is to write the subroutine in C instead. It is possible to accept C strings by doing a two-stage bridge routine. The first stage, written in C, handles the FTN_NAME part of the transfer, and passes the string and its length as arguments to the second bridge.

The second bridge, written in FORTRAN, accepts the strings as BYTE arrays and uses the SUBLIB routine `mvlc` and the passed-in length to convert them to FORTRAN strings, which may then be passed to the FORTRAN main subroutine. To get strings out, the reverse procedure is used. The FORTRAN bridge uses `mvcl` to write a BYTE array, then passes that and the length back to the C bridge, which puts a null terminator on the string and returns it to the C caller.

Examples of this type of bridge are the SPICE bridge routines, in the file SPBRI.COM. One of the bridges, for `bodvar`, is presented below. The C-callable first bridge follows:

```
void zbodvar(body, item, dim, values)

int body;
char *item;
int *dim;
double *values;
{
   int i;
   i=strlen(item);

   FTN_NAME(xbodvar) (&body, item, &i, dim, values);
}
```

It takes the length string parameter and passes it with the string pointer to the second-stage bridge. It converts scalar and array arguments, and uses the FTN_NAME macro. The second bridge called by code above follows:

```
subroutine xbodvar(body, item, i, dim, values)

      integer body
      byte item(1)
      integer i
      integer dim
      double precision values(*)
      character*80 text

      text='
      if (i.gt.80) call xvmessage('xbodvar, string too long',' ')

C     Transformation to Fortran-string
      call mvlc(item, text, i)

      call bodvar(body, text, dim, values)

      return
```

It converts the C string (passed as a BYTE array) and length to a FORTRAN CHARACTER*n variable, which is then passed into the `bodvar` routine, the FORTRAN -callable version.

2.6.4.3 Machine Dependencies

Machine dependencies can't always be avoided. Code may be dependent on the operating system (VMS or UNIX), the computer hardware or variety of UNIX. There are significant differences among UNIX implementations.

Machine dependencies may be isolated into separate source files, and compiled only on the relevant machine. Or machine-dependent code can be written in-line, and the C preprocessor used to select the appropriate version.

The first solution, separate source files, is appropriate when there are entire routines that are implemented differently under various operating systems. This option is normally used for differences between VMS and UNIX; only rarely will you have separate source files for variants of UNIX. Filenames end in an underscore and the OS name, for example, "open_input_file_vms.c" and "open_input_file_unix.c".

Once you have separate working source files, get the appropriate one to compile during a build by modifying the imakefile.

The MODULE_LIST (or other appropriate macro) is defined based on the machine type. Since vimake uses the C preprocessor, the rules listed below for machine-dependent preprocessor symbols should be used. For example, a program named "prog.c" calls a routine that is OS-dependent, named "sub_vms.c" and "sub_unix.c":

```
#if VMS_OS
#define MODULE_LIST prog.c sub_vms.c
#define CLEAN_OTHER_LIST sub_unix.c
#else
#define MODULE_LIST prog.c sub_unix.c
#define CLEAN_OTHER_LIST sub_vms.c
#endif
```

The CLEAN_OTHER_LIST macro deletes the source code for the module not compiled during a clean-source operation.

The second solution to the machine dependency problem uses the C preprocessor to conditionally compile parts of the code. This method is appropriate for small differences and handling differences between various flavors of UNIX. The necessary preprocessor symbols are defined in `xvmaininc.h`.

Use the symbols "VMS_OS" and "UNIX_OS" to select VMS or UNIX. Use "#if " (not "#ifdef"). "#else" may be used to select the other operating system, but the #else clause must apply to UNIX. Put "VMS_OS" in the #if statement.

To open a standard text file with the `fopen()` routine, VMS requires that you give RMS file type arguments to `fopen()`. Otherwise, EDT will not be able to access the file properly. For example:

```
/* Open an output text file */
#if VMS_OS
file = fopen(filename, “w”,”rat=cr”, “rfm=var”);
#else
file = fopen(filename, “w”);
#endif
```

The file xvmaininc.h defines symbols for all the types of machines VICAR runs on, such as: “VAX_ARCH”, “SUN4_ARCH”, “MAC_AUX_ARCH”, etc., but these are never used in program code. For example, the `fstat()` system routine returns the optimal blocksize on some machines, but not on others. The Sun 4 and DECstation have it, but Silicon Graphics and HP do not. The `mmap()` command, is available on Sun 4 and Silicon Graphics, but not on DECstation or HP; the groupings are different.

Machine dependencies based on *features*, rather than *machines* is the proper way to solve the problem. To port to a new system, we determine whether or not it has the feature in question, and modify the include files. For example, the symbol “FSTAT_BLKSIZE_OS” defines whether or not `fstat()` returns the optimal blocksize. “MMAP_AVAIL_OS” defines whether or not the `mmap()` routine is present. These and other *feature defines*, which all end in “_OS” by convention, handle machine dependencies. All such definitions end in “_OS”, so searches through the code to find the machine dependencies are easy.

The rule is: conditional compilation statements to handle machine dependencies use names of specific features that are required in the code. These names are defined in standard include files based on the machine type.

Examine xvmaininc.h for current machine dependencies, and check again; it will change. There are machine dependencies that aren't listed in xvmaininc.h. Contact the VICAR system programmer to add the dependency to xvmaininc.h. This is the only include file an imakefile can use.

Since xvmaininc.h is under the control of the VICAR system programmer, it is not always possible for an application programmer to have it modified on short notice. “vmachdep.h” is provided in the portable includes (p2$inc on VMS and $P2INC on UNIX). It contains dependencies that are needed only by application programs. If your definition is in vmachdep.h, it must be included in your source.

Use the style of xvmaininc.h when adding new dependencies to “vmachdep.h”, and make sure you determine the correct setting for every architecture VICAR supports. If you can't find out, contact the system programmer, and if you still can't, make note of the unknowns in the include file. Reserve “vmachdep.h”, add your definition, and redeliver it, even if your application isn't ready for delivery. Then other programmers can modify the file as well. Don’t sit on “vmachdep.h”! Following are examples:

```
#if FTRUNCATE_AVAIL_OS
   if (ftruncate(devstate->dev.disk.channel, j) != 0)
      status = errno;
   else
#endif
      status = SUCCESS;

...

#if OPEN_PROTECT_OS
   file = open(filename, O_RDWR|O_CREAT, 0666);
#else
   file = open(filename, O_RDWR|O_CREAT);
#endif

...

/* This example shows finding the EOF with and without fstat */

#if FSTAT_AVAIL_OS
#if VMS_OS
#include <stat.h>
#else
#include <sys/types.h>
#include <sys/stat.h>
#endif
#else
#include seek_include
#endif
...
#if FSTAT_AVAIL_OS
   struct stat statbuf;
#endif
...
#if FSTAT_AVAIL_OS
   status = fstat(file, &statbuf);
   if (status != 0) return errno;
   eof = statbuf.st_size;
#else                                        /* fstat not available */
   status = lseek(file, 0, SEEK_END);
   if (status == -1) return errno;
   eof = status;
#endif
```

2.7 Writing Portable FORTRAN

This Section discusses writing portable FORTRAN code for VICAR applications and describes how to deal with machine dependencies in VICAR code. It does not explain the rules for writing portable FORTRAN; see a reference book on FORTRAN. There are a few allowed VMS extensions to FORTRAN, which are described below.

2.7.1 RTL Issues

Portable FORTRAN applications use the rules below when calling the RTL:

- **Terminator**. RTL routines with keyword-value pairs of optional arguments have a terminator in the argument list, even if none of the optional arguments are used.
- **Character strings.** Strings input to the RTL are CHARACTER*n variables or constants. Both for string arguments and token words in keyword-value pairs. See 2.7.4 READ & WRITE to Strings (page 35).
- **Optional arguments.** Pure optional arguments are not allowed. Most routines require all arguments to be specified.

2.7.2 No EQUIVALENCE for Type Conversion

The use of EQUIVALENCE to convert among datatypes, especially byte to integer, is a common trick. This practice is not portable. EQUIVALENCE must not be used to convert any data types. EQUIVALENCE is used only to conserve memory. Access the data using the same data type you stored it with.

We recommend the **x/zvtrans** family of routines for data conversion. See 1.6 Converting Data Types & Hosts (page 8). An alternative efficient method for converting between byte and integer uses the functions `INT2BYTE` and `BYTE2INT`. Include the file "fortport" in your subroutine. "fortport" is a SUBLIB include; add it to FTNINC_LIST in the imakefile.

`INT2BYTE` and `BYTE2INT` assume that the data is in the 0 to 255 range. No bounds checking is performed. The functions are implemented either as array lookups (rather than functions), or as statement functions, so the include file must be present to get the appropriate definition. Example:

```
SUBROUTINE do_something(b, i)
        BYTE b
        INTEGER i
        INCLUDE 'fortport'
        b = INT2BYTE(i)
        i = BYTE2INT(b)
        RETURN
```

2.7.3 CHARACTER*n for Strings

The FORTRAN standard does not support the use of BYTE arrays as character strings. They must be declared as CHARACTER*n variables. Moreover, descriptors are not used on most machines, so there is no way to tell the difference between a BYTE and a CHARACTER*n variable. Since CHARACTER*n variables have a declared length associated with them, and BYTE arrays do not, any routine expecting a length will get garbage if a BYTE variable is passed in.

The RTL expects that *all* strings will be CHARACTER*n variables. This applies to all RTL routines, but is especially important for the output routines **qprint** and **x/zvmessage** (**x/zvmessage** is preferred over **qprint** for new code). All strings destined for output must be CHARACTER*n variables. READ & WRITE to Strings (page 35) discusses how to do output conversion to create the strings.

While changing output strings to CHARACTER*n is possible, there are many areas in the code, in files and data structures, where strings are stored in BYTE arrays, and it is not practical to change.

For that reason, character strings in BYTE arrays may be used if it is embedded too deeply to change. The SUBLIB routines `mvcl` and `mvlc` have been provided to convert between BYTE arrays and CHARACTER*n variables. Do *not* copy data directly to a CHARACTER*n variable from a BYTE array, or vice-versa, since that will fail if there are descriptors.

Subroutines should use CHARACTER*n string arguments. In cases where it is impractical to change, BYTE arrays may be passed. Passing a CHARACTER*n argument where a BYTE array is expected is just as bad as the opposite. If possible, include separate entry points to the routine that accept both types. For example, the routine `vic1lab` returns data in a character string, while `vic1labx` returns data in a BYTE array.

2.7.4 READ & WRITE to Strings

To perform input and output conversion, use the FORTRAN I/O package on an "internal file". An internal file to FORTRAN is simply a CHARACTER*n variable. Give the name of the variable (it may be a substring) instead of the unit number in a FORTRAN READ or WRITE statement. Write to a string and send the string to **qprint** or **x/zvmessage**. Do not attempt to write directly to the terminal. Any messages written directly to the terminal will not appear in the session log and maybe printed in unexpected ways due to interaction with VICAR I/O or SAGE.

Here is one way to code the above example (the leading blank is not required in **xvmessage**):

```
REAL*8 VRES, VRAD, VTAN
      CHARACTER*80 MS1
      WRITE (MS1,1001) VRES, VRAD, VTAN
 1001 FORMAT ('VEL=', F9.2, ' (VRAD,VTAN)=(', F7.2, ',', F9.2, ')
M/SEC')
      CALL XVMESSAGE(MS1, ')
```

There are three things to note about this example:

- The DATA statement is separate from the declaration. Putting initialization data on the declaration line is not portable.
- The format specifier may be put in-line in the WRITE statement if you wish.
- The numbers changed from the `outcon` call because the leading blank was eliminated.

The **xvmessage** routine always prints the first character, with no carriage control. So, all the character position numbers are one less than in the `outcon` example.

Conversion of data from string to numeric form using READ is similar. Use standard FORTRAN I/O using the CHARACTER*n variable as the internal file. Substrings are much more useful on READ to read only the portion you are interested in. For example, the following call form is typical:

```
CALL INCON(NPAR, %REF(ALABEL(OFF+I+5:)),PAR,20)
      BUF(2) = PAR(1)                  !Frame number
```

where ALABEL is the input string (already a CHARACTER*n variable), BUF(2) is an integer receiving the value, and NPAR is ignored. The value is an integer in the range 0 to 99 (obtained from the documentation), so this call could be converted to:

```
READ (ALABEL(OFF+I+5:),'BN,I2') BUF(2)
```

You may want to make use of the ERR=n clause on reads in order to trap errors such as a decimal point being present in an integer.

2.7.5 VMS FORTRAN Extensions

VMS has many non-standard extensions to its FORTRAN compiler, which are used frequently in VICAR code. Some of these extensions are widely available in other vendors' FORTRAN compilers, while others are not. Only standard FORTRAN77 code should be used in a portable program.

To find out what is standard and what isn't, refer to the ANSI FORTRAN standard, or the *VAX FORTRAN Language Reference Manual*. Anything printed in blue in that manual is non-standard FORTRAN and should not be used, except as noted below.

Some of the FORTRAN extensions are so useful that it would be impractical to write VICAR code without them. Fortunately, these extensions are common industry-wide and are available in the FORTRAN compilers for every machine MIPL is interested in. Therefore some extensions to standard FORTRAN77 are allowed. These extensions are listed below.

You should not use any non-standard statements or features that are not mentioned below. If you absolutely have to, then make sure it is isolated in a machine-specific section of code, and provide a means for performing the same function on machines that don't have that extension.

Following are the *only* permit VMS FORTRAN extensions:

- BYTE and INTEGER*2 data types. (Not LOGICAL*1).
- Data type length specifiers in general (i.e. The *2 in INTEGER*2). They should only be used for pixel declarations, as listed. See Table 6: FORTRAN declarations for pixel types (page 25).
- DO-WHILE loops.
- DO-END DO loops.
- INCLUDE statement (see above for usage).
- Symbolic names up to 31 characters, with at least 14 significant in external names.
- Lowercase letters and underscore (_) are allowed in symbolic names. Names are not case sensitive.
- Exclamation point () starts a comment anywhere in the source line.
- Tab-format source lines (source lines may start with a TAB character).
- IMPLICIT NONE statement.

2.7.6 VMS-Specific Code

Many current VICAR applications and subroutines have VMS-specific code embedded in them. Some of it is obvious, like a system service call. Some is insidiously difficult to find, like using a double precision floating point value as a single. This works on VMS because the first half of a double value looks like a float. This is not the case on any other machine.

All VMS-specific code must be eliminated or isolated. If the same thing can be done in a portable way, do it that way. If not, then isolate the VMS-specific code and write UNIX code to perform the same function. If the function is useful as a general-purpose subroutine, then put it in SUBLIB. If not, include it with your code. See 2.6.4.3 Machine Dependencies (page 31) for methods of dealing with machine-dependent code.

An attempt is made below to list the types of VMS-specific code you will run into. This is not and can not be an exhaustive list, as there are far too many potential problem areas that will only be

uncovered with more experience. Use this list as examples of what to look out for. You should be familiar with writing portable FORTRAN code; if not then see a standard FORTRAN manual (the black type in the *VAX FORTRAN Language Reference Manual* is a good source).

- The order of bytes in an integer (or any other data type) is reversed on VMS with respect to most other machines. Any code that depends on byte order must be changed to not do so. Byte-order dependencies typically come from converting between BYTE and other data types, where the first byte of an integer is set to the value of a byte, then used as an integer. This is typically accomplished via EQUIVALENCE. Use the **x/zvtrans** routines or the `BYTE2INT` and `INT2BYTE` method described above instead.

- Integers and double precision floating point values must not be used as INTEGER*2s or REALs without conversion. On VMS, the address of a double could be treated as the address of a real, or they could be equivalenced to each other. This worked because the bit pattern for the first half of a double is the same as for a real. The same is true for short vs. normal integers. A pointer to an INTEGER could be treated as a pointer to an INTEGER*2 or even a BYTE. This is not valid on non-VMS systems. The IEEE floating point standard uses different bit patterns for doubles and for single-precision reals. For integers, most other machines use the “big-endian” byte order. The first bytes of an INTEGER are actually the higher-order bytes, unlike VMS, so the value cannot be treated as an INTEGER*2.

This practice is most common (and hardest to find) in subroutine calls, where a program is sloppy about passing data of the correct type. If the subroutine expects a REAL, the value passed in better be a REAL and not a DOUBLE PRECISION. The same is true for INTEGER, INTEGER*2, and BYTE. Be *very* careful to watch out for this, as it is difficult to find.

- VMS System Calls must be eliminated or isolated. These include a whole range of things, like system services (SYS$), VMS Run-Time Library calls (LIB$, etc., not to be confused with the VICAR RTL), RMS calls or structures, QIO calls for I/O, AST's (asynchronous system traps), and anything else that is VMS specific. Any subroutine or structure with a dollar sign ($) is suspect, as most VMS system routines have $ in the name.

There are two choices for dealing with these. The first is to dispense with system-specific calls and do things in a standard way, when possible. This could be achieved by using standard UNIX-style calls that do the same thing (many of which are implemented under VMS), which may imply calling C code that calls the UNIX routines. The second is to isolate the VMS-specific code, and write corresponding code that works on a UNIX system. Both versions could either be included as separate modules in the program itself, or they could be put in a SUBLIB subroutine if it's a capability that could be generally useful. See 2.6.4.3 Machine Dependencies (page 31) for ways of handling machine-dependent code. In general, system routines should not be called from FORTRAN, as there is little standardization.

- Assembly language (called Macro on the VAX) included in the program will have to be rewritten. You could write assembler versions for every machine supported, but this is *highly* discouraged. The best solution is to write the subroutine in a high-level language, preferably C. FORTRAN could be used, but C is generally a better match to assembly language.

- Character strings must be CHARACTER*n variables instead of BYTE, LOGICAL*1, or INTEGER arrays. This topic is discussed in depth above.

- FORTRAN -C interfaces must be checked. There are several rules for mixing FORTRAN and C, including having separate names for subroutines that are to be called from both FORTRAN and C. Make sure you don't call a subroutine from FORTRAN by its C interface, or vice-versa, as that is not portable. String handling is a particular problem. See 2.6 Mixing FORTRAN and C (page 27), for

details.

- Make sure that *all* input operations from files can accept files written on a foreign machine. This is often handled automatically, but there are cases where it is not. See 1.3 Data Types and Host Representations (page 8) for details.
- Do not assume that you know the size of a pixel in an inputfile. An integer may not be four bytes on every machine that VICAR runs on. Use the RTL routines **x/zvpixsize**, **x/zvpixsizeu**, or **x/zvpixsizeb** to get the size of a pixel.
- Knowledge of the bit patterns used to store data is not portable, especially floating-point data. This is not very common, but any code that does bit-twiddling is likely to have problems. Byte-order dependencies often creep into this sort of code.
- Use of EQUIVALENCE to access the same bit pattern in different ways is highly unportable. If EQUIVALENCEs are used, make sure you use only one of the definitions for any given piece of data. If you use the other definition, put a new value of that type in to the EQUIVALENCE. Be careful when mixing INTEGER and REAL data types in a single array (the FORTRAN way of doing structures). While that can be legitimate if done properly, care has to be taken since the data sizes may not be the same on all machines.
- Optional arguments are not allowed in subroutines. Supply all the arguments the subroutine requires. If you are porting a subroutine that accepts optional arguments, you have the choice of eliminating the extra arguments, forcing them to be there, or creating different subroutine names with different numbers of arguments.
- Do not use numeric constants for things like error numbers. VICAR constants could change in the future, and "CANNOT_FIND_KEY" is much more understandable than "-38". Use the VICAR include files when needed.
- Variable and subroutine names must not conflict with UNIX system routines or the RTL internal routines. Most UNIX machines do not have the concept of a shareable image like VMS does. Shareable images allowed subroutine libraries such as the RTL to hide their internals from the program. Without them, the entire RTL is linked with your program. This greatly increases the chance of name collisions, as the internal RTL subroutine names (and TAE and SPICE and X-windows and ...) are suddenly visible, and some of the names are fairly common. Be very careful with subroutine and global variable names.

The ultimate solution to this problem is to have a consistent naming scheme for RTL internal names that won't conflict with applications. Until this is implemented, however, watch out for conflicts. Even this fix won't help other subroutine packages that MIPL does not have direct control over, such as TAE, SPICE, and X-windows.

- There are many differences in the file system and filename structure between VMS and UNIX. The VMS path name of "`disk:[dir.subdir]file.ext;version`" is quite different from the UNIX path name of " `/dir/subdir/subdir/filename`". Logical names do not exist under UNIX. The analog to logical names, environment variables, act quite differently in many respects (for example, the system `open()` call doesn't know how to deal with environment variables, although **x/zvopen** and **x/zvfilename** do).

Filenames and path names that are embedded in the program should be removed and made available as arguments, both to handle architecture differences and differences in directory structure on other machines. Any code that parses filenames from user input must be aware of the differences and have

code to handle each system. Such code should be rare as the RTL does most of the filename parsing; however, it does exist.

- Filenames and path names tend to be longer under UNIX than VMS. Many current programs arbitrarily limit filename parameters to 40 or even 20 characters in the PDF and in the code. These limits need to be lifted. Most of the time, the program never sees the filename (it lets **x/zvunit** take care of it), so the solution is simply to not specify the maximum string length in the PDF. Instead of "(STRING,40)", use "STRING". If you need a buffer in the program code to handle the filename, allow at least 255 characters (which is slightly more than the maximum TAE string size).
- The filenames of the program units themselves will need changing. FORTRAN language modules must end in a ".f", *not* ".for ", to be compatible with vimake. UNIX likes ".f", and it is more picky about filename extensions than VMS is.
- Many VAX FORTRAN language extensions are used in current VICAR code. A small subset of extensions is allowed (they are listed above), since they are available on all machines of interest. All other extensions will have to be removed, as they are not portable. Check the *VAX FORTRAN Language Reference Manual* for the feature in question. If it is printed in blue ink, then it is not portable and may not be used, unless it is in the above list.

2.8 Portable TAE Command Language (TCL)

The dcl command allows you to put VMS DCL commands in your procedure, which are not portable. The corresponding command under UNIX is `ush`, which allows you to execute shell commands inside the procedure.

In order to use the `dcl` or `ush` commands, there has to be some way of determining which operating system you are on. A global variable called "$syschar" holds the type of operating system. Index 1 in this variable is either "UNIX" or "VAX_VMS". For example:

```
procedure
parm file string
refgbl $syschar
body
if ($syschar(1) = "UNIX")
  ush rm &file
else
  dcl delete &file
end-if
end-proc
```

Unfortunately, there appears to be no straightforward way to determine the particular version of UNIX, so use common shell commands.

Another major trouble area for TCL is filenames. Filenames and path names look much different under VMS and UNIX. There is a lot of TCL code in VICAR that parses filenames, appends filenames to directories, etc. Many test scripts use hardcoded VMS directories and filenames to find test files. All of these will have to change. The same "$syschar" variable can be used to do different things with the filename, or pick different testfiles, based on which system you are using.

The "$syschar" variable may also be tested inside help files, help within a PDF, and menus via special conditional commands. These commands are part of TAE, but they are not documented by TAE. The conditionals test to see if the given string is in any element of the "$syschar" variable. Like other PDF/MDF directives, they should appear on a line by themselves with period "." in the first

column:

- **.if *string***: Prints the following lines (up to .elseif or .ifend) if *string* is in ”$syschar”.
- **.ifn *string***: Prints the following lines (up to .elseif or .ifend) if *string* is not in ”$syschar”.
- **elseif *string***: Prints the following lines of text (up to .ifend or another .elseif) if the previous condition was not met, and *string* is in “$syschar”.
- **.ifend**: Ends a conditional clause.
- **.if1 *string***: Single-line conditional. Just like .if, but only the next line is printed, and no .ifend is required.
- **.ifn1 *string***: Single-line negative conditional. Just like .ifn, but only the next line is printed, and no .ifend is required.

The following example is taken from the TAE command mode help file (commode.hlp):

```
                CONT*INUE
.if VAX_VMS
                DCL               any-VMS/DCL-command
                DCL-NOINTERRUPT   any-VMS/DCL-command
.ifend
                DEFC*MD COMMAND=command-name STRING=replacement-string

...

                T*UTOR-NOSCREEN PROC=proc-subcmd  parameters
.if1 UNIX
                USH any-UNIX/shell-command
                WAIT-ASYNC JOB=job-name-list

...

                        RUNTYPE Command Qualifier (continued)

           If the command qualifier is set to BATCH, the following TAE
           message is displayed:

.if VAX_VMS
                Job (nnn) submitted to queue (que)

           where "nnn" is the assigned job number and "que" is
           the name of the queue the job was submitted to.

.elseif UNIX
                Batch job file "filename" submitted successfully.

           where "filename" is the batch job file name, defined
           as "proc".job.

.ifend
```

Refer to the TAE documentation if you have more questions on writing portable TCL.

3. Image I/O

3.1 Introduction

This Section describes the image I/O to be used with the MIPL VICAR image processing executive.

Note that a x/zvread or x/zwrite operation, without specifying the location (line/sample/band), automatically reads the "next" record. In BSQ this is the next line. See · UPD_HIST, string: The valid values for UPD_HIST are ON and OFF. Default is OFF. UPD_HIST controls whether or not to write a history label when the file is opened in UPDATE mode. If it is ON, a label is written; if OFF (the default), no label is written. UPD_HIST is only used in UPDATE mode, not in READ or WRITE. In WRITE mode, a history label is always created).

x/zvread—Read a line (page 62) for more information.

3.1.1 Unix filename expansion

Under Unix, filenames accepted by the RTL (via x/zvunit), either the U_NAME optional or the INP or OUT parameters) now can contain environment variables and usernames. A reference of the form *$var* will be replaced with the value of the environment variable *var*. The name of the environment variable (but not the dollar sign) may optionally be enclosed in curly braces (${*var*}). A tilde (~) followed by a username will be replaced with the home directory of that user. A tilde without a username will be replaced with the home directory of the current account. Both of these expansions exactly mimic the behavior of the C-shell, so they should be familiar to most Unix users.

3.1.2 Temporary files

Filenames that begin with a plus sign (+) are treated as *temporary* files in both Unix and VMS. The old VMS VICAR style of specifying temporary filenames was to leave off the filename extension, which was replaced with a .Z*xx* extension (where *xx* is based on the process ID). This approach still works under VMS, but it is not supported under Unix. A Unix filename can have no extension and still be perfectly legal. Plus, it is infeasible to search all the user's directories to delete temporary files when the user logs off (which is what happens under VMS).

Instead of scattering temporary files all over the place (distinguised by their name), the new style is to collect them all in one directory. Prepending a plus sign (+) to the name tells the VICAR RTL to put the files in the temporary directory. This directory is pointed at by the $VTMP environment variable in Unix, and the VTMP: rooted logical name in VMS. VTMP is set up by vicset2 for both systems (it normally points at /tmp/*username* for Unix and a scratch directory for VMS). Because VTMP: is a rooted directory, accessing the top-level directory outside of VICAR is a little more difficult; you must use “vtmp:[000000]”.

Subdirectories of VTMP are allowed. Under Unix, they look like “+sub/dir/file”, while under VMS, they look like “+[sub.dir]file”. The subdirectories are not automatically created; use “mkdir $VTMP/sub/dir” under Unix and “cre/dir vtmp:[sub.dir]” under VMS. Because of these differences, the use of subdirectories is not portable between systems.

Currently, all processes using the same login id share the same temporary directory. This may be changed in the future so concurrent independent jobs will have separate directories. In the meantime, a workaound is to redefine VTMP to use a process-specific directory name.

3.1.3 Filename Expansions

Any time the RTL accepts a filename (x/zvunit or x/zvfilename), there are a number of expansions that can occur, most of which will be familiar to C-shell programmers. Also, a special syntax for temporary files has been provided.

For UNIX, the expansions are as follows:

- **`$var`**:Expand environment variable "`var`".
- **`${var}`**:Expand environment variable "`var`".
- **`~user`**: Expand to home directory of user"`user`".
- **`~`**: Expand to home directory of current user (`$HOME`).
- **`$$`**: Insert a single `$` (no environment variable).
- **`+`**: Expand to translation of `"$VTMP/`" for temporary files.

For VMS, the expansions are as follows:

- **`+`**: Expand to "`vtmp:`" (if subdirectory present) or "`vtmp:[000000]`"(if no subdirectory) for temporary files.
- **no + and no suffix**: Append ". Z *xx*" (where *xx* comes from `v2$pidcode`) for old-style temporary names.

Under VMS, both expansions may occur on the same name.

The temporary filename locations (`$VTMP` and `vtmp`) are set up in vicset2.

3.2 Image I/O API

The nine routines for image I/O (line I/O and array I/O) are described below. A description of the optional arguments follows each subroutine description.

3.2.1 x/zvadd—Add information to control block

```
call xvadd(unit, status, <optionals>, ' ')
status = zvadd(unit, <optionals>, 0);
```

x/zvadd updates the internal control block information describing the file associated with UNIT. Each

item of information is identified by a key word and is followed by a value which holds the update information. The keyword-value pairs are optional and may be entered in any order. To process the file, **x/zvadd** must be called before opening (with **x/zvopen**) the file.

Arguments:

- **UNIT**: input, integer
  Unit number of file from **x/zvunit**.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 128). See also the optional arguments OPEN_ACT, IO_ACT, LAB_ACT, and ERR_ACT.

Optional arguments:

- **U_NL:** input, integer
  Indicates that the Executive is to create an output file capable of holding NL lines. If the file associated with the unit number already exists but is too small, it will be enlarged.
- **U_NS:** input, integer
  As in NL but for samples.
- **U_NB:** input, integer
  As in NL but for bands.
- **U_NBB:** input, integer
  For output files, indicates the number of bytes of binary prefix that will precede each image line. The binary prefix will be stored in an area before each image line. It will be hidden from subsequent programs unless the program opens the file with a COND value that includes BINARY. Subsequent reads of a file with COND including BINARY will return the binary prefix to each line ahead of the line in the read buffer. An output file with a non-zero U_NBB will expect the binary prefix to precede the image line in the **x/zvwrit** buffer. For input files U_NBB is ignored. Defaults to 0.
- **U_NLB:** input, integer
  For output files, indicates the number of lines of binary header information that is to precede the image lines in an image file. The image I/O subsystem will expect these lines to be written with **x/zvwrit** prior to the writing of the image lines. The image header will remain hidden from subsequent programs unless those programs open the file with a value of COND that includes the substring, BINARY. For input files, **U_NLB** is ignored. Defaults to 0.
- **U_N1, U_N2, U_N3**: input, integers
  Indicate that the Executive is to ensure that the output file will have the dimensions specified by the values associated with the respective arguments. U_N1 refers to the first file dimension, U_N2 the second, and U_N3 the third. These arguments are useful for the program which does not care about the file organization, and simply wants to deal with records and pixels. The relation between these arguments and the normal NL, NS, NB arguments are as follows:

OR G	N 1	N 2	N 3
BS Q	N S	N L	N B

BI L	N S	N B	N L
BI P	N B	N S	N L

Table 9: File Organization

- **U_ORG:** input, string
  Indicates the file organization that the programmer expects or desires for the file. The acceptable organizations are band sequential, band interleaved by pixel, and band interleaved by line. The respective values that represent these states are, 'BSQ', 'BI P', and 'BIL'. If the file is being created by the **x/zvopen**, then the indicated organization will become the new file organization. If the file is an existing file that is not to be deleted and created again, then the Executive will check the indicated organization against that of the existing file. If they are not the same then an error condition will be raised. If the image file has less than three dimensions, this argument will be ignored.

- U_DIM: input, integer
  Number of dimensions of dimensions the file has. Current default if U_DIM not specified is 3. 4 is permitted but not supported.

- **METHOD**: input, string
  Indicates the access method for the file.
  Valid values are 'SEQ' (sequential) or 'RANDOM'. The default is 'SEQ'. 'SEQ' and 'RANDOM' only effect the internal buffering used by the executive; if the method is 'SEQ', then a larger buffer will be used for I/O, resulting in more efficient processing. If 'RANDOM' is given, a smaller buffer will be used to avoid reading unnecessary data.

If the file is a tape file, an error condition will be raised if 'RANDOM' is specified. The specification of 'RANDOM' for a file with OP equal to 'WRITE' is equivalent to 'SEQ'.

- **OP**: input, string
  Indicates the intended operation. Valid values are 'READ', 'WRITE', and 'UPDATE'. Default is 'READ'. **Output files must have OP equal to WRITE.**

- **OPEN_ACT:** input, string
  Indicates the action to be taken by the Executive when an error condition is raised as a result of the open. A valid string may contain one or more of three characters in any order, 'A', 'U', 'S'. The presence of each character has the following meanings:

  - **A**: an error will cause an immediate program abort. Otherwise, the routine will return an error number in STATUS.

  - **U**: an error will cause the string that the user has specified with the OPEN_MES optional to be printed.

  - **S**: an error will cause a system message to be printed.

The three values are independent. Thus 'SA' will cause a system message to be printed and the program to abort if an error occurs. If OPEN_ACT consists only of blanks, an error will cause the **x/zvopen** to return an error number in STATUS. Default is the action specified with **x/zveaction**.

- **IO_ACT:** input, string

As in OPEN_ACT but for **x/zvread** and **x/zvwrit**. Default is specified with **x/zveaction**.

- **LAB_ACT:** input, string
  Specified as is OPEN_ACT, LAB_ACT defines a default error action to be taken by the XL routines. It can be overridden in a call to an XL routine with the ERR_ACT optional. The message given if a 'U' is given is specified by LAB_MESS. The default for LAB_ACT is given by **x/zveaction**.
- **CLOS_ACT:** input, string
  Indicates an action to be taken when the file is closed. If the string contains one of the substrings listed below, the corresponding action will be taken. More than one action may be specified.
  - DELETE: Causes the file to be deleted. Default is to retain the file.
  - FREE: Frees up the unit number so it can be used again. If this action is specified, then the file can no longer be accessed using this unit number. If you want to access the file again, x/zvunit must be called again. Since there are a limited number of unit slots available, FREE is most useful when processing large numbers of files. It allows VICAR to re-use this unit slot. If FREE is not specified (the default), then the file may be opened again via x/zvopen without calling x/zvunit, but VICAR cannot assign this unit number to another file.
- **FORMAT**: output, string, maximum length 32
  The format of the pixel in an existing file. Possibilities are BYTE, HALF, FULL, REAL, DOUB, COMP. WORD and COMPLEX are equivalent to HALF and COMP, but are non-standard.
- **O_FORMAT:** input, string
  For input files this argument is ignored. For output files, O_FORMAT becomes the format of the output file. Valid values are 'BYTE', 'HALF', 'FULL', 'REAL', 'DOUB', and 'COMP'. 'WORD' and 'COMPLEX' are equivalent to 'HALF' and 'COMP', respectively, but for consistency are no longer to be used. The Executive will ensure that the output pixels have the indicated FORMAT by converting if necessary. A related argument is U_FORMAT which is the format of the pixels returned or passed to **x/zvread** or **x/zvwrit**. Default is the primary input FORMAT, that is, the format of the pixels in the input file.
- **U_FORMAT:** input, string
  Indicates the format of pixels in the buffer returned by **x/zvread** or the format of the pixels the program will pass to **x/zvwrit**. If for input files U_FORMAT is not equal to the input file format, then VICAR will convert. On output, if U_FORMAT is not equal to O_FORMAT then VICAR will convert. Default for input files is the input file format; for output files, it is the primary input U_FORMAT.
- **I_FORMAT:** input, string, maximum length 8
  If an input file is unlabeled and does not have the default pixel format of 'BYTE', then this optional may be used to set the proper format; valid values are ' BYTE', 'HALF', 'FULL','REAL', 'DOUB', and 'COMP'. 'WORD' and 'COMPLEX' are equivalent to 'HALF' and 'COMP', respectively, but for consistency should no longer be used.
- **OPEN_MES:** input, string, maximum length 132
  OPEN_MES holds the user message that is issued when the OPEN_ACT argument contains the substring 'U' and an error occurs. Default is specified by **x/zveaction**.
- **IO_MESS:** input, string, maximum length 132
  Holds the user message that is issued under certain settings of the IO_ACT argument when I/O

errors occur. Default is specified by **x/zveaction**.

- **LAB_MESS:** input, string, maximum length 132
  Holds the user message that is issued when the ERR_ACT optional is not given to an XL routine, and the LAB_ACT optional contains the substring 'U'. Default is specified by **x/zveaction**.

- **TYPE**: input, string, maximum length 8
  With this argument, the programmer can designate the type of the output file desired. If the file already exists the Executive will alter any file name and label items that indicate the file type. On input, the file will be checked for type. If a mismatch occurs between the TYPE argument and the actual file type, an error status condition will be raised. Currently supported values are:

  - IMAGE: The file contains image data.
  - PARM: The file contains parameters for input to a program via the PARAMS parameter on the command line.
  - PARAM: Same as PARM, but an older file type.
  - GRAPH1: The file is an IBIS Graphics-1 file.
  - GRAPH2: The file is an IBIS Graphics-2 file.
  - GRAPH3: The file is an IBIS Graphics-3 file.
  - TABULAR: The file is an IBIS Tabular file.

- **COND**: input, string, maximum length 132
  If the string is null or contains only blanks, this argument will have no effect. If the string contains one of the substrings listed below, the indicated condition will be set for the specified file.

  - **NOLABELS**: Indicates that no label processing will be done for the file.
  - **NOBLOCK**: Indicates that if the file is an output file (OP=WRITE), the file will not be blocked (RECSIZE=BUFSIZ).
  - **BINARY:** indicates that the binary prefix and binary header, if they exist, will be unhidden to the program (see U_NLB and U_NBB)
  - **VARREC**: Indicates that the file is a tape containing variable length records. NOLABELS and NOBLOCK must also be specified with this argument.

The string may contain more than one substring, such as 'COND', 'NOBLOCK NOLABELS'. Default is to label and block files and hide the binary areas of the file.

- **U_FILE:** input, integer
  Used to specify the file number to be accessed on a tape. If U_FILE = 0, the tape advances to the next file (the default). If U_FILE > 0, the tape is moved to the absolute file number given. U_FILE may not be less than zero. The first file on a tape is file number one; thus if U_FILE = 1, the tape will be rewound.

- **HOST, string:** The type of computer used to generate the image. The value for HOST appears in the system image label. It is used only for documentation; the RTL uses the INTFMT and REALFMT label items to determine the representation of the pixels in the file. HOST will default to the type of computer the program is running on, so it normally will not be needed. It is used to write a file in a

host format other than the native one. While this is not recommended (the general rule is read anything, write native format), it is allowed. See 1.3 Data Types and Host Representations (page 8), for more information. New values for HOST will appear every time the RTL is ported to a new machine, so no checking is done on the string. However, the currently accepted values and what they represent, are listed. See Table 1: Valid VICAR HOST Labels and Machine Types (page 10). In addition, the following values are also accepted:

- NATIVE: The host type of the currently running machine
- LOCAL: Same as NATIVE

- **INTFMT, string**: The format used to represent integers in the file. INTFMT, REALFMT, and HOST should all match, so if you change one please change all three. INTFMT is a system label item in the image used by the RTL to automatically convert data read in, via **x/zvread,** to the native integer representation. It defaults to the representation of the machine the program is running on, so it only needs to be set when writing in a non-native format. If you are reading a file in such a way that the RTL does *not* automatically convert datatypes, such as Array I/O or CONVERT OFF, pay attention to the INTFMT label (via **x/zvget**) and use the **x/zvtrans** representation (for binary labels see BINTFMT below). See 1.3 Data Types and Host Representations (page 8), for more information. The valid values of INTFMT may change as the RTL is ported to new machines. However, the currently valid values are listed. See Table 2: Valid VICAR Integer Formats (page 11). In addition, the following values are also accepted:
  - NATIVE: The host type of the currently running machine
  - LOCAL: Same as NATIVE

- **REALFMT, string** :The format used to represent floating point numbers in the file. INTFMT, REALFMT, and HOST should all match, so if you change one please change all three. REALFMT is a system label item in the image used by the RTL to automatically convert data read in, via **x/zvread**, to the native floating point representation. It defaults to the representation of the machine the program is running on, so it only need s to be set when writing in a non-native format. If you are reading a file in such a way that the RTL does *not a*utomatically convert data types, such as Array I/O or CONVERT OFF, pay attention to the REALFMT label (via **x/zvget**) and use the **x/zvtrans** family of routines to translate to the native representation (for binary labels see INTFMT below). See 1.3 Data Types and Host Representations (page 8), for more information. The valid values of REALFMT may change as the RTL is ported to new machines. However, the currently valid values are listed. See Table 3: Valid VICAR Real Number Formats (page 11). In addition, the following values are also accepted:
  - **NATIVE**: The host type of the currently running machine
  - **LOCAL**: Same as NATIVE

- **BHOST, string**: The type of computer used to generate the binary labels. The value for BHOST appears in the system image label. It is used only for documentation; the RTL uses the BINTFMT and INTFMT label items to determine the representation of the binary labels in the file. For input (or update) files, the BHOST optional is used if the INTFMT label is not present in the file (if neither

are present it defaults to VAX-VMS). For output files, the BHOST optional is used if is OFF (if not present the file's BHOST comes from the primary input, or defaults to VAX-VMS). If is ON, the BHOST optional is ignored since the file's INTFMT gets set to the native host. See 1.3 Data Types and Host Representations (page 8), for more information. The valid values for BHOST are exactly the same as for HOST above (including NATIVE and LOCAL).

- **BINTFMT, string**: The format used to represent integers in the binary label. BINTFMT, BREALFMT, and BHOST should all match, so if you change one change all three. BINTFMT is a system label item in the image made available by the RTL to help applications to convert binary labels read or written to the file. For input (or update) files, the BINTFMT optional is used if the BINTFMT label is not present in the file (if neither are present it defaults to the integer format for VAX-VMS). For output files, the BINTFMT optional is used if is OFF (if not present the file's BINTFMT comes from the primary input, or defaults to the integer format for VAX-VMS). If is ON, the BINTFMT optional is ignored since the file's BINTFMT gets set to the native host integer format. See 1.3 Data Types and Host Representations (page 8), for more information. Applications using binary labels should pay close attention to BINTFMT, as they are responsible for doing their own data format conversions.

The valid values for BINTFMT are exactly the same as for INTFMT above (including NATIVE and LOCAL).

- **BREALFMT, string**: The format used to represent real numbers in the binary label. BREALFMT, BINTFMT, and BHOST should all match, so if you change one change all three. INTFMT is a system label item in the image made available by the RTL to help applications to convert binary labels read or written to the file. For input (or update) files, the BREALFMT optional is used if the BREALFMT label is not present in the file (if neither are present it defaults to the floating-point format for VAX-VMS). For output files, the BREALFMT optional is used if is OFF (if not present the file's INTFMT comes from the primary input, or defaults to the floating-point format for VAX-VMS). If is ON, the BREALFMT optional is ignored since the file's BREALFMT is set to the native host floating-point format. See 1.3 Data Types and Host Representations (page 8), for more information. Applications using binary labels should pay close attention to BREALFMT, as they are responsible for doing their own data format conversions.
  The valid values for BREALFMT are exactly the same as for REALFMT above (including NATIVE and LOCAL).

- **BLTYPE, string:** The type of the binary label**.** The value for BLTYPE appears in the system image label. This is not type in the sense of datatype, but is a string identifying the kind of binary label in the file. The RTL does not do any interpretation or checking of BLTYPE. It is intended mainly for documentation, so people looking at the image will know what kind of binary label is present. It may also be used by application programs to make sure they can process the given type of binary label, or to make sure it is processed correctly. For input (or update) files, the BLTYPE optional is used if the BLTYPE label is not present in the file (not that “used” in this sense means only being made available to **x/zvget**). For output files, the BLTYPE optional is used for the BLTYPE system label of the file (regardless of the setting). If the optional is not present, the file's BLTYPE comes from the primary input. See 1.3 Data Types and Host Representations (page 8), for more information. The valid values of BLTYPE are maintained in a name registry, so that all possible kinds of binary labels can be documented in one place. Although the RTL does no checking on the given name, only names that are registered should be used in BLTYPE. See 1.7.2 Programming and Binary Labels (page 16), for more details.

- **CONVERT**, string: The valid values for CONVERT are ON and OFF. The default is ON.

Normally, the RTL will automatically convert pixels that are read from a file to the native host representation, and to the data type specified in U_FORMAT.

Setting CONVERT to OFF turns off both of these conversions, giving you the bit patterns that are in the file directly. When writing a file, host conversion is possible (although not normally used), but the U_FORMAT data type conversion is normally performed. Setting CONVERT to OFF turns off both of these conversions as well, writing the bit patterns that are in memory directly to the file.

Note: Be careful! If you turn CONVERT OFF, you are responsible for doing your own data type conversions. Any given program should be able to read files written on any machine, so if you turn off CONVERT you should make use of the **x/ztvtrans** family of routines. See 1.3 Data Types and Host Representations (page 8), for details.

- **BIN_CVT**, string: The valid values for BIN_CVT are ON and OFF. The default is OFF. BIN_CVT is used to inform the RTL whether or not you will be converting binary labels to the native host representation. If BIN_CVT is ON, then the host formats in the output file (BHOST, BINTFMT, and BREALFMT) will be set to the native machine's host formats. This will be the standard case for applications that know how to interpret the binary label. Since the binary label type is known, the application can convert the data to the native format before writing the file.

If the application does not know the binary label type, however (such as a general-purpose application), then it cannot convert the host format. In this case, set BIN_CVT to OFF. This means the output file will receive the binary label host types of the primary input file, defaulting to VAX-VMS if not available. The general-purpose application may then simply transfer the binary labels over to the output file without re-formatting them. The BHOST, BINTFMT, and BREALFMT optional arguments will override this setting, but if BIN_CVT is OFF (they are ignored if BIN_CVT is on). BIN_CVT is ignored for input files. See 1.7.2 Programming and Binary Labels (page 16), for more details.

- **UPD_HIST**, string: The valid values for UPD_HIST are ON and OFF. Default is OFF. UPD_HIST controls whether or not to write a history label when the file is opened in UPDATE mode. If it is ON, a label is written; if OFF (the default), no label is written. UPD_HIST is only used in UPDATE mode, not in READ or WRITE. In WRITE mode, a history label is always created).

3.2.2 x/zvclose Close a file

```
call xvclose(unit, status, <optionals>, ' ')
status = zvclose(unit, <optionals>, 0);
```

x/zvclose will close an open image file. The use of **x/zvclose** is required in an application program.

Arguments:

- **UNIT**: input, integer
  Unit number of file from **x/zvunit**.

- **STATUS**: output, integer
  Error status code. A value of one indicates success. See the optional arguments OPEN_ACT, and IO_ACT under **x/zvopen**.

Optional arguments:

- **CLOS_ACT:** input, string, maximum length 132
  Indicates an action to be taken when the file is closed. If the string contains one of the substrings listed below, the corresponding action will be taken. More than one action may be specified.

- **DELETE**: Causes the file to be deleted. Default is to retain the file.
- **FREE**: Frees up the unit number so it can be used again. If this action is specified, then the file can no longer be accessed using this unit number. If you want to access the file again, x/zvunit must be called again. Since there are a limited number of unit slots available, FREE is most useful when processing large numbers of files. It allows VICAR to re-use this unit slot. If FREE is not specified (the default), then the file may be opened again via **x/zvopen** without calling **x/zvunit**, but VICAR cannot assign this unit number to another file.

3.2.3 x/zveaction—Set the default error handling action

call xveaction (action, message)
status = zveaction (action, message)

x/zveaction sets the default error handling action for the entire VICAR job. It is used as a default for all the error handling optionals such as OPEN_ACT, IO_ACT, LAB_ACT, and ERR_ACT, and it is used directly by routines that do not have their own independent error action arguments.

The default if **x/zveaction** is not called is a null string for both the action and the message.

Arguments:

- **ACTION**: input, string, maximum length 8
  Indicates the action to be taken by the Executive when an error condition is raised. A valid string may contain one or more of three characters in any order, 'A', 'U', 'S'. The presence of each character has the following meanings:
  - **A:** an error will cause an immediate program abort. Otherwise, the routine will return an error number in a status variable (if available).
  - **U**: an error will cause the string that is specified with the MESSAGE argument to be printed.
  - **S**: an error will cause a system message to be printed.

  The three values are independent of each other. Thus 'SA' will cause a system message to be printed and the program to abort if an error occurs. If ACTION consists only of blanks, or if **x/zveaction** has never been called, an error will cause the Executive to return an error number in a status variable (if one is available on the particular subroutine that had the error).
- **MESSAGE**: input, string, maximum length 132
  MESSAGE holds the user message that is issued when the ACTION argument contains the substring 'U' and an error occurs. Even if ACTION is overridden by a action optional (such as OPEN_ACT), the message specified by this argument may still be used if the optional contains the substring 'U', unless,, it is overridden by the corresponding message optional (such as OPEN_MES).
- **STATUS**: output, integer
  Error status code. A value of one indicates success. See the optional arguments OPEN_ACT, and IO_ACT under **x/zvopen**.

3.2.4 x/zvget—Retrieve control block information

```
call xvget(unit, status, <optionals>, ' ')
status = zvget(unit, <optionals>, 0);
```

x/zvget permits the programmer to request file information stored in the executive's internal control block associated with a unit number. The item of information is identified by an input key word

field. The information is returned by **x/zvget** in the variable following the key word argument. The keyword-value pairs are optional and may be entered pairwise in any order.

Arguments:

- **UNIT**: input, integer
  Unit number of file from **x/zvunit**.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. See the optional arguments OPEN_ACT and IO_ACT under **x/zvopen**.

Optional arguments:

- **FORMAT**: output, string, maximum length 32
  The format of the pixel in an existing file. Possibilities are BYTE, HALF, FULL, REAL, DOUB, COMP. WORD and COMPLEX are equivalent to HALF and COMP, but are non-standard.
- **NL**: output, integer
  The number of lines in the image
- **NS**: output, integer
  The number of samples in the image.
- **NB**: output, integer
  The number of bands in the image.
- **N1, N2, or N3**: output, integer
  The number of pixels in the first, second and third dimensions of the image. The values returned refer to the physical dimensions of the file and are independent of file organization.
- **DIM**: output, integer
  The dimension of the image. Valid values are 2 or 3. Currently, all new files created by the executive have a DIM of 3. If DIM is 3 and NB is 1, then the file effectively has a DIM of 2.
- **ORG**: output, string, maximum length 32
  The file organization; organizations may be: BSQ (band sequential), BIP (band interleaved by pixel), or BIL (band interleaved by line).
- **NAME**: output, string, maximum length 132.
  The host system file name. For a display, the display id is returned.
- **TYPE**: output, string, maximum length 32
  The type of file. Currently supported values are:
  - **IMAGE**: The file contains image data.
  - **PARM**: The file contains parameters for input to a program via the PARAMS parameter on the command line.
  - **PARAM**: Same as PARM, but an older file type.
  - **GRAPH1**: The file is an IBIS Graphics-1 file.
  - **GRAPH2**: The file is an IBIS Graphics-2 file.
  - **GRAPH3**: The file is an IBIS Graphics-3 file.

- **TABULAR**: The file is an IBIS Tabular file.

- **PIX_SIZE:** output, integer
  The size of a pixel in bytes.
- **BUFSIZ**: output, integer
  The size of the internal buffer being used for the I/O operations. If the file is a tape, BUFSIZ will be equivalent to the block size of the tape in bytes.
- **RECSIZE**: output, integer
  The record size in bytes. Equal to PIX_SIZE x N1.
- **VARSIZE**: output, integer
  The record size of the last read for variable length records. VARSIZE is only valid if COND VARREC was given in **x/zvopen** or **x/zvadd**. If VARREC was not specified, the value of VARSIZE is undefined.
- **FLAGS:** output, integer
  FLAGS is an integer whose bits represent various attributes of the unit in question. To test a value in FLAGS, perform a bit-wise AND with it. For example, in the C language you could test to see if a file is open with the code

```
int flags;

xvget(&unit, &stat, "FLAGS", &flags);
if (flags & OPEN) {. . .
```

  The various flags listed here are predefined for applications written in C. Only the names should be used, to protect against changes in future versions of the executive.

  - **OPEN** indicates that the unit has already been opened by **x/zvopen**.
  - **NO_LABELS** indicates that the file was opened with the COND, NOLABELS option.
  - **LABELS_MODIFIED** indicates that the label I/O routines have been called, modifying the image label.
  - **UNIT_TAPE** indicates that the unit is a tape file.
  - **DATA_WRITTEN** indicates that the first logical data record was written. For internal use only.
  - **NOBLOCK** indicates that COND, NOBLOCK was specified at open time, and that tapes will be unblocked (one logical record per physical record).
  - **BINARY** indicates that COND, BINARY was specified at open time, causing binary labels, if present, to be available to the application.
  - **VARREC** indicates that COND, VARREC was specified at open time, allowing variable length records on a tape.
  - **SEQUENTIAL_DEVICE** indicates that the device only allows sequential access, such as a magnetic tape.
  - **HOST, string:** The type of computer used to generate the image. The value for HOST appears in the system image label. It is used only for documentation; the RTL uses the INTFMT and REALFMT

label items to determine the representation of the pixels in the file. HOST will default to the type of computer the program is running on, so it normally will not be needed. It is used to write a file in a host format other than the native one. While this is not recommended (the general rule is read anything, write native format), it is allowed. See 1.3 Data Types and Host Representations (page 8), for more information. New values for HOST will appear every time the RTL is ported to a new machine, so no checking is done on the string. However, the currently accepted values, and what they represent, are listed. See Table 1: Valid VICAR HOST Labels and Machine Types (page 10). In addition, the following values are also accepted:

- **NATIVE**: The host type of the currently running machine
- **LOCAL**: Same as NATIVE

- **INTFMT, string**: The format used to represent integers in the file. INTFMT, REALFMT, and HOST should all match, so if you change one please change all three. INTFMT is a system label item in the image used by the RTL to automatically convert data read in, via **x/zvread,** to the native integer representation. It defaults to the representation of the machine the program is running on, so it only needs to be set when writing in a non-native format. If you are reading a file in such a way that the RTL does *not* automatically convert datatypes, such as Array I/O or CONVERT OFF, pay attention to the INTFMT label (via **x/zvget**) and use the **x/zvtrans** representation (for binary labels see BINTFMT below). See Data Types and Host Representations (page 8), for more information. The valid values of INTFMT may change as the RTL is ported to new machines. However, the currently valid values are listed. See Table 2: Valid VICAR Integer Formats (page 11). In addition, the following values are also accepted:
  - **NATIVE**: The host type of the currently running machine
  - **LOCAL**: Same as NATIVE

- **REALFMT, string** :The format used to represent floating point numbers in the file. INTFMT, REALFMT, and HOST should all match, so if you change one please change all three. REALFMT is a system label item in the image used by the RTL to automatically convert data read in, via **x/zvread**, to the native floating point representation. It defaults to the representation of the machine the program is running on, so it only need s to be set when writing in a non-native format. If you are reading a file in such a way that the RTL does *not a*utomatically convert data types, such as Array I/O or CONVERT OFF, pay attention to the REALFMT label (via **x/zvget**) and use the **x/zvtrans** family of routines to translate to the native representation (for binary labels see INTFMT below). See Data Types and Host Representations (page 8), for more information. The valid values of REALFMT may change as the RTL is ported to new machines. However, the currently valid values are listed. See Table 2: Valid VICAR Integer Formats (page 11). In addition, the following values are also accepted:
  - **NATIVE**: The host type of the currently running machine
  - **LOCAL**: Same as NATIVE

- **BHOST, string**: The type of computer used to generate the binary labels. The value for BHOST appears in the system image label. It is used only for documentation; the RTL uses the BINTFMT

and INTFMT label items to determine the representation of the binary labels in the file. For input (or update) files, the BHOST optional is used if the INTFMT label is not present in the file (if neither are present it defaults to VAX-VMS). For output files, the BHOST optional is used if is OFF (if not present the file's BHOST comes from the primary input, or defaults to VAX-VMS). If is ON, the BHOST optional is ignored since the file's INTFMT gets set to the native host. See 1.3 Data Types and Host Representations (page 8), for more information. The valid values for BHOST are exactly the same as for HOST above (including NATIVE and LOCAL).

- **BINTFMT, string**: The format used to represent integers in the binary label. BINTFMT, BREALFMT, and BHOST should all match, so if you change one change all three. BINTFMT is a system label item in the image made available by the RTL to help applications to convert binary labels read or written to the file. For input (or update) files, the BINTFMT optional is used if the BINTFMT label is not present in the file (if neither are present it defaults to the integer format for VAX-VMS). For output files, the BINTFMT optional is used if is OFF (if not present the file's BINTFMT comes from the primary input, or defaults to the integer format for VAX-VMS). If is ON, the BINTFMT optional is ignored since the file's BINTFMT gets set to the native host integer format. See 1.3 Data Types and Host Representations (page 8), for more information. Applications using binary labels should pay close attention to BINTFMT, as they are responsible for doing their own data format conversions.

The valid values for BINTFMT are exactly the same as for INTFMT (including NATIVE and LOCAL).

- **BLTYPE, string:** The type of the binary label**.** The value for BLTYPE appears in the system image label. This is not type in the sense of datatype, but is a string identifying the kind of binary label in the file. The RTL does not do any interpretation or checking of BLTYPE. It is intended mainly for documentation, so people looking at the image will know what kind of binary label is present. It may also be used by application programs to make sure they can process the given type of binary label, or to make sure it is processed correctly. For input (or update) files, the BLTYPE optional is used if the BLTYPE label is not present in the file (not that “used” in this sense means only being made available to **x/zvget**). For output files, the BLTYPE optional is used for the BLTYPE system label of the file (regardless of the setting). If the optional is not present, the file's BLTYPE comes from the primary input. See 1.3 Data Types and Host Representations (page 8), for more information. The valid values of BLTYPE are maintained in a name registry, so that all possible kinds of binary labels can be documented in one place. Although the RTL does no checking on the given name, only names that are registered should be used in BLTYPE. See 1.7.2 Programming and Binary Labels (page 16), for more details.

- **BREALFMT, string**: The format used to represent real numbers in the binary label. BREALFMT, BINTFMT, and BHOST should all match, so if you change one change all three. INTFMT is a system label item in the image made available by the RTL to help applications to convert binary labels read or written to the file. For input (or update) files, the BREALFMT optional is used if the BREALFMT label is not present in the file (if neither are present it defaults to the floating-point format for VAX-VMS). For output files, the BREALFMT optional is used if is OFF (if not present the file's INTFMT comes from the primary input, or defaults to the floating-point format for VAX-VMS). If is ON, the BREALFMT optional is ignored since the file's BREALFMT is set to the native host floating-point format. See 1.3 Data Types and Host Representations (page 8), for more information. Applications using binary labels should pay close attention to BREALFMT, as they are responsible for doing their own data format conversions.
  The valid values for BREALFMT are exactly the same as for REALFMT(including NATIVE and LOCAL).

- **IMG_REC: output, integer**
  The number of the last record read or written using x/zvread and x/zvwrit. In BSQ this is the last line.
- **UPD_HIST**, string: The valid values for UPD_HIST are ON and OFF. Default is OFF. UPD_HIST controls whether or not to write a history label when the file is opened in UPDATE mode. If it is ON, a label is written; if OFF (the default), no label is written. UPD_HIST is only used in UPDATE mode, not in READ or WRITE. In WRITE mode, a history label is always created).

3.2.5 x/zvopen—Open a file

```
call xvopen(unit, status, <optionals>, ' ')
status = zvopen(unit, <optionals>, 0);
```

x/zvopen prepares the file for I/O processing. If the file is an input file, it verifies the file's existence and the existence and form of its label.

For output files, **x/zvopen** will ensure that the file is the right size, that being either the size indicated by the programmer on the call to **x/zvopen**, the size field for the file, or the input file, in that order. If the file does not exist, **x/zvopen** will create it. If the file exists but is too small, **x/zvopen** will expand it. If the file is a tape file, it will be positioned to the desired file.

For all output files, **x/zvopen** will create, unless the programmer negates this default function, an output label for the file.

A call to **x/zvopen** is required before I/O processing can take place on the file.

Arguments:

- **UNIT**: input, integer
  Unit number of file from **x/zvunit**.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107). See also the optional arguments OPEN_ACT, IO_ACT, LAB_ACT, and ERR_ACT.

Optional arguments:

- **U_NL:** input, integer
  Indicates that the Executive is to create an output file capable of holding NL lines. If the file associated with the unit number already exists but is too small, it will be enlarged.
- **U_NS:** input, integer
  As in NL but for samples.
- **U_NB:** input, integer
  As in NL but for bands.
- **U_NBB:** input, integer
  For output files, indicates the number of bytes of binary prefix that will precede each image line. The binary prefix will be stored in an area before each image line. It will be hidden from subsequent programs unless the program opens the file with a COND value that includes BINARY. Subsequent reads of a file with COND including BINARY will return the binary prefix to each line ahead of the line in the read buffer. An output file with a non-zero U_NBB will expect the binary prefix to precede the image line in the **x/zvwrit** buffer. For input files U_NBB is ignored. Defaults to 0.

- **U_NLB:** input, integer
  For output files, indicates the number of lines of binary header information that is to precede the image lines in an image file. The image I/O subsystem will expect these lines to be written with **x/zvwrit** prior to the writing of the image lines. The image header will remain hidden from subsequent programs unless those programs open the file with a value of COND that includes the substring, BINARY. For input files, **U_NLB** is ignored. Defaults to 0.

- **U_N1, U_N2, U_N3**: input, integers
  Indicate that the Executive is to ensure that the output file will have the dimensions specified by the values associated with the respective arguments. U_N1 refers to the first file dimension, U_N2 the second, and U_N3 the third. These arguments are useful for the program which does not care about the file organization, and simply wants to deal with records and pixels. The relation between these arguments and the normal NL, NS, NB arguments are as follows:

OR G	N 1	N 2	N 3
BS Q	N S	N L	N B
BI L	N S	N B	N L
BI P	N B	N S	N L

Table 10: File Organization

- **ADDRESS** output, pointer (integer for FORTRAN)
  ADDRESS is a special argument which instructs the executive to cause the image file to be treated as an array of memory, and returns to the program the address of the beginning of the image data. The program may then use this address to manipulate the image data. No calls to **x/zvread** or **x/zvwrit** are needed, although they will still function. See the examples Section for details of how to use this argument in FORTRAN and C. **Note:** There is a VMS memory management flaw that occasionally surfaces when using the ADDRESS optional. If you are opening and closing very large array I/O files, you might get a VASFULL (Virtual Address Space Full) error message, even though there should be plenty of virtual memory left. If this happens (and you're programming in C), try doing a large malloc () immediately followed by a free () call for the same memory area before opening any array I/O files. It is best to put the calls at the top of the program. This solution sounds like a no-operation, and from the programmer's point of view it is, but it causes VMS to do things that get around the problem. The amount of memory you allocate should enough to cover all dynamic memory allocated by your program, plus about 100K or so. Try a megabyte to start with, and adjust it upwards or downwards as needed.
  Example:

```
free (malloc (1024*1024)); /* Fix Virtual Address Space Full error */
```

- **U_ORG:** input, string
  Indicates the file organization that the programmer expects or desires for the file. The acceptable organizations are band sequential, band interleaved by pixel, and band interleaved by line. The respective values that represent these states are, 'BSQ', 'BI P', and 'BIL'. If the file is being created by the **x/zvopen**, then the indicated organization will become the new file organization. If the file is an

existing file that is not to be deleted and created again, then the Executive will check the indicated organization against that of the existing file. If they are not the same then an error condition will be raised. If the image file has less than three dimensions, this argument will be ignored.

- U_DIM: input, integer
  Number of dimensions of dimensions the file has. Current default if U_DIM not specified is 3. 4 is permitted but not supported.

- **METHOD**: input, string

Indicates the access method for the file. Valid values are 'SEQ' (sequential) or 'RANDOM'. The default is 'SEQ'. 'SEQ' and 'RANDOM' only effect the internal buffering used by the executive; if the method is 'SEQ', then a larger buffer will be used for I/O, resulting in more efficient processing. If 'RANDOM' is given, a smaller buffer will be used to avoid reading unnecessary data.

If the file is a tape file, an error condition will be raised if 'RANDOM' is specified. The specification of 'RANDOM' for a file with OP equal to 'WRITE' is equivalent to 'SEQ'.

- **OP**: input, string
  Indicates the intended operation. Valid values are 'READ', 'WRITE', and 'UPDATE'. Default is 'READ'. **Output files must have OP equal to WRITE.**

- **OPEN_ACT:** input, string
  Indicates the action to be taken by the Executive when an error condition is raised as a result of the open. A valid string may contain one or more of three characters in any order, 'A', 'U', 'S'. The presence of each character has the following meanings:

  - **A**: an error will cause an immediate program abort. Otherwise, the routine will return an error number in STATUS.

  - **U**: an error will cause the string that the user has specified with the OPEN_MES optional to be printed.

  - **S**: an error will cause a system message to be printed.

The three values are independent. Thus 'SA' will cause a system message to be printed and the program to abort if an error occurs. If OPEN_ACT consists only of blanks, an error will cause the **x/zvopen** to return an error number in STATUS. Default is the action specified with **x/zveaction**.

- **IO_ACT:** input, string
  As in OPEN_ACT but for **x/zvread** and **x/zvwrit**. Default is specified with **x/zveaction**.

- **LAB_ACT:** input, string
  Specified as is OPEN_ACT, LAB_ACT defines a default error action to be taken by the XL routines. It can be overridden in a call to an XL routine with the ERR_ACT optional. The message given if a 'U' is given is specified by LAB_MESS. The default for LAB_ACT is given by **x/zveaction**.

- **CLOS_ACT:** input, string
  Indicates an action to be taken when the file is closed. If the string contains one of the substrings listed below, the corresponding action will be taken. More than one action may be specified.

  - DELETE: Causes the file to be deleted. Default is to retain the file.

  - FREE: Frees up the unit number so it can be used again. If this action is specified, then the file can no longer be accessed using this unit number. If you want to access the file again,

x/zvunit must be called again. Since there are a limited number of unit slots available, FREE is most useful when processing large numbers of files. It allows VICAR to re-use this unit slot. If FREE is not specified (the default), then the file may be opened again via x/zvopen without calling x/zvunit, but VICAR cannot assign this unit number to another file.

- **O_FORMAT:** input, string
  For input files this argument is ignored. For output files, O_FORMAT becomes the format of the output file. Valid values are 'BYTE', 'HALF', 'FULL', 'REAL', 'DOUB', and 'COMP'. 'WORD' and 'COMPLEX' are equivalent to 'HALF' and 'COMP', respectively, but for consistency are no longer to be used. The Executive will ensure that the output pixels have the indicated FORMAT by converting if necessary. A related argument is U_FORMAT which is the format of the pixels returned or passed to **x/zvread** or **x/zvwrit**. Default is the primary input FORMAT, that is, the format of the pixels in the input file.
- **U_FORMAT:** input, string
  Indicates the format of pixels in the buffer returned by **x/zvread** or the format of the pixels the program will pass to **x/zvwrit**. If for input files U_FORMAT is not equal to the input file format, then VICAR will convert. On output, if U_FORMAT is not equal to O_FORMAT then VICAR will convert. Default for input files is the input file format; for output files, it is the primary input U_FORMAT.
- **I_FORMAT:** input, string, maximum length 8
  If an input file is unlabeled and does not have the default pixel format of 'BYTE', then this optional may be used to set the proper format; valid values are ' BYTE', 'HALF', 'FULL','REAL', 'DOUB', and 'COMP'. 'WORD' and 'COMPLEX' are equivalent to 'HALF' and 'COMP', respectively, but for consistency should no longer be used.
- **OPEN_MES:** input, string, maximum length 132
  OPEN_MES holds the user message that is issued when the OPEN_ACT argument contains the substring 'U' and an error occurs. Default is specified by **x/zveaction**.
- **IO_MESS:** input, string, maximum length 132
  Holds the user message that is issued under certain settings of the IO_ACT argument when I/O errors occur. Default is specified by **x/zveaction**.
- **LAB_MESS:** input, string, maximum length 132
  Holds the user message that is issued when the ERR_ACT optional is not given to an XL routine, and the LAB_ACT optional contains the substring 'U'. Default is specified by **x/zveaction**.
- **TYPE**: input, string, maximum length 8
  With this argument, the programmer can designate the type of the output file desired. If the file already exists the Executive will alter any file name and label items that indicate the file type. On input, the file will be checked for type. If a mismatch occurs between the TYPE argument and the actual file type, an error status condition will be raised. Currently supported values are:
  - IMAGE: The file contains image data.
  - PARM: The file contains parameters for input to a program via the PARAMS parameter on the command line.
  - PARAM: Same as PARM, but an older file type.

- GRAPH1: The file is an IBIS Graphics-1 file.
- GRAPH2: The file is an IBIS Graphics-2 file.
- GRAPH3: The file is an IBIS Graphics-3 file.
- TABULAR: The file is an IBIS Tabular file.

- **COND**: input, string, maximum length 132
  If the string is null or contains only blanks, this argument will have no effect. If the string contains one of the substrings listed below, the indicated condition will be set for the specified file.
  - **NOLABELS**: Indicates that no label processing will be done for the file.
  - **NOBLOCK**: Indicates that if the file is an output file (OP=WRITE), the file will not be blocked (RECSIZE=BUFSIZ).
  - **BINARY:** indicates that the binary prefix and binary header, if they exist, will be unhidden to the program (see U_NLB and U_NBB)
  - **VARREC**: Indicates that the file is a tape containing variable length records. NOLABELS and NOBLOCK must also be specified with this argument.

  The string may contain more than one substring, such as 'COND', 'NOBLOCK NOLABELS'. Default is to label and block files and hide the binary areas of the file.
- **U_FILE:** input, integer
  Used to specify the file number to be accessed on a tape. If U_FILE = 0, the tape advances to the next file (the default). If U_FILE > 0, the tape is moved to the absolute file number given. U_FILE may not be less than zero. The first file on a tape is file number one; thus if U_FILE = 1, the tape will be rewound.
- **HOST, string:** The type of computer used to generate the image. The value for HOST appears in the system image label. It is used only for documentation; the RTL uses the INTFMT and REALFMT label items to determine the representation of the pixels in the file. HOST will default to the type of computer the program is running on, so it normally will not be needed. It is used to write a file in a host format other than the native one. While this is not recommended (the general rule is read anything, write native format), it is allowed. See 1.3 Data Types and Host Representations (page 8), for more information. New values for HOST will appear every time the RTL is ported to a new machine, so no checking is done on the string. However, the currently accepted values and what they represent, are listed. See Table 1: Valid VICAR HOST Labels and Machine Types (page 10). In addition, the following values are also accepted:
  - NATIVE: The host type of the currently running machine
  - LOCAL: Same as NATIVE

- **INTFMT, string**: The format used to represent integers in the file. INTFMT, REALFMT, and HOST should all match, so if you change one please change all three. INTFMT is a system label item in the image used by the RTL to automatically convert data read in, via **x/zvread,** to the native integer representation. It defaults to the representation of the machine the program is running on, so it only needs to be set when writing in a non-native format. If you are reading a file in such a way

that the RTL does *not* automatically convert datatypes, such as Array I/O or CONVERT OFF, pay attention to the INTFMT label (via **x/zvget**) and use the **x/zvtrans** representation (for binary labels see BINTFMT below). See 1.3 Data Types and Host Representations (page 8), for more information. The valid values of INTFMT may change as the RTL is ported to new machines. However, the currently valid values are listed. See Table 2: Valid VICAR Integer Formats (page 11). In addition, the following values are also accepted:

- NATIVE: The host type of the currently running machine
- LOCAL: Same as NATIVE

- **REALFMT, string** :The format used to represent floating point numbers in the file. INTFMT, REALFMT, and HOST should all match, so if you change one please change all three. REALFMT is a system label item in the image used by the RTL to automatically convert data read in, via **x/zvread**, to the native floating point representation. It defaults to the representation of the machine the program is running on, so it only need s to be set when writing in a non-native format. If you are reading a file in such a way that the RTL does *not a*utomatically convert data types, such as Array I/O or CONVERT OFF, pay attention to the REALFMT label (via **x/zvget**) and use the **x/zvtrans** family of routines to translate to the native representation (for binary labels see INTFMT below). See 1.3 Data Types and Host Representations (page 8), for more information. The valid values of REALFMT may change as the RTL is ported to new machines. However, the currently valid values are listed. See Table 3: Valid VICAR Real Number Formats (page 11). In addition, the following values are also accepted:
  - **NATIVE**: The host type of the currently running machine
  - **LOCAL**: Same as NATIVE

- **BHOST, string**: The type of computer used to generate the binary labels. The value for BHOST appears in the system image label. It is used only for documentation; the RTL uses the BINTFMT and INTFMT label items to determine the representation of the binary labels in the file. For input (or update) files, the BHOST optional is used if the INTFMT label is not present in the file (if neither are present it defaults to VAX-VMS). For output files, the BHOST optional is used if is OFF (if not present the file's BHOST comes from the primary input, or defaults to VAX-VMS). If is ON, the BHOST optional is ignored since the file's INTFMT gets set to the native host. See 1.3 Data Types and Host Representations (page 8), for more information. The valid values for BHOST are exactly the same as for HOST above (including NATIVE and LOCAL).
- **BINTFMT, string**: The format used to represent integers in the binary label. BINTFMT, BREALFMT, and BHOST should all match, so if you change one change all three. BINTFMT is a system label item in the image made available by the RTL to help applications to convert binary labels read or written to the file. For input (or update) files, the BINTFMT optional is used if the BINTFMT label is not present in the file (if neither are present it defaults to the integer format for VAX-VMS). For output files, the BINTFMT optional is used if is OFF (if not present the file's BINTFMT comes from the primary input, or defaults to the integer format for VAX-VMS). If is ON, the BINTFMT optional is ignored since the file's BINTFMT gets set to the native host integer format. See 1.3 Data Types and Host Representations (page 8), for more information. Applications using binary labels should pay close attention to BINTFMT, as they are responsible for doing their own data format conversions.

The valid values for BINTFMT are exactly the same as for INTFMT above (including NATIVE and LOCAL).

- **BREALFMT, string**: The format used to represent real numbers in the binary label. BREALFMT, BINTFMT, and BHOST should all match, so if you change one change all three. INTFMT is a system label item in the image made available by the RTL to help applications to convert binary labels read or written to the file. For input (or update) files, the BREALFMT optional is used if the BREALFMT label is not present in the file (if neither are present it defaults to the floating-point format for VAX-VMS). For output files, the BREALFMT optional is used if is OFF (if not present the file's INTFMT comes from the primary input, or defaults to the floating-point format for VAX-VMS). If is ON, the BREALFMT optional is ignored since the file's BREALFMT is set to the native host floating-point format. See 1.3 Data Types and Host Representations (page 8), for more information. Applications using binary labels should pay close attention to BREALFMT, as they are responsible for doing their own data format conversions.
  The valid values for BREALFMT are exactly the same as for REALFMT above (including NATIVE and LOCAL).

- **BLTYPE, string:** The type of the binary label**.** The value for BLTYPE appears in the system image label. This is not type in the sense of datatype, but is a string identifying the kind of binary label in the file. The RTL does not do any interpretation or checking of BLTYPE. It is intended mainly for documentation, so people looking at the image will know what kind of binary label is present. It may also be used by application programs to make sure they can process the given type of binary label, or to make sure it is processed correctly. For input (or update) files, the BLTYPE optional is used if the BLTYPE label is not present in the file (not that "used" in this sense means only being made available to **x/zvget**). For output files, the BLTYPE optional is used for the BLTYPE system label of the file (regardless of the setting). If the optional is not present, the file's BLTYPE comes from the primary input. See 1.3 Data Types and Host Representations (page 8), for more information. The valid values of BLTYPE are maintained in a name registry, so that all possible kinds of binary labels can be documented in one place. Although the RTL does no checking on the given name, only names that are registered should be used in BLTYPE. See 1.7.2 Programming and Binary Labels (page 16), for more details.

- **CONVERT**, string: The valid values for CONVERT are ON and OFF. The default is ON. Normally, the RTL will automatically convert pixels that are read from a file to the native host representation, and to the data type specified in U_FORMAT.

Setting CONVERT to OFF turns off both of these conversions, giving you the bit patterns that are in the file directly. When writing a file, host conversion is possible (although not normally used), but the U_FORMAT data type conversion is normally performed. Setting CONVERT to OFF turns off both of these conversions as well, writing the bit patterns that are in memory directly to the file.

Note: Be careful! If you turn CONVERT OFF, you are responsible for doing your own data type conversions. Any given program should be able to read files written on any machine, so if you turn off CONVERT you should make use of the **x/ztvtrans** family of routines. See 1.3 Data Types and Host Representations (page 8), for details.

- **BIN_CVT**, string: The valid values for BIN_CVT are ON and OFF. The default is OFF. BIN_CVT is used to inform the RTL whether or not you will be converting binary labels to the native host representation. If BIN_CVT is ON, then the host formats in the output file (BHOST, BINTFMT, and BREALFMT) will be set to the native machine's host formats. This will be the standard case for applications that know how to interpret the binary label. Since the binary label type is known, the application can convert the data to the native format before writing the file.

If the application does not know the binary label type, however (such as a general-purpose application), then it cannot convert the host format. In this case, set BIN_CVT to OFF. This means the output file will receive the binary label host types of the primary input file, defaulting to VAX-VMS if not available. The general-purpose application may then simply transfer the binary labels over to the output file without re-formatting them. The BHOST, BINTFMT, and BREALFMT optional arguments will override this setting, but if BIN_CVT is OFF (they are ignored if BIN_CVT is on). BIN_CVT is ignored for input files. See 1.7.2 Programming and Binary Labels (page 16), for more details.

- **UPD_HIST**, string: The valid values for UPD_HIST are ON and OFF. Default is OFF. UPD_HIST controls whether or not to write a history label when the file is opened in UPDATE mode. If it is ON, a label is written; if OFF (the default), no label is written. UPD_HIST is only used in UPDATE mode, not in READ or WRITE. In WRITE mode, a history label is always created).

3.2.6 x/zvread—Read a line

call xvread(unit, buffer, status, <optionals>, ' ')
status = zvread(unit, buffer, <optionals>, 0);

x/zvread will read a single line from the file associated with the UNIT. The data will be returned in BUFFER and the status indicator will be returned in STATUS.

Arguments:

- **UNIT**: input, integer
  Unit number of file from **x/zvunit**.
- **BUFFER**: output, array of anything
  Array to receive input data.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107). See also the optional arguments OPEN_ACT, IO_ACT, LAB_ACT, and ERR_ACT.

Optional arguments:

- **LINE**: input, integer
  Indicates the starting line for the read. For BSQ images, by default the starting line is incremented from its position in the last read until each band is exhausted (after NL reads), at which time it resets to the first line for the new band. For BIL images, the line number increments after NB reads, and for BIP images the line number increments after NS x NB reads.
- **SAMP**: input, integer
  Indicates the starting sample (starting pixel in the SAMP dimension) for the read. For BSQ and BIL images, the default is 1. For BIP images, the starting sample is incremented from the last read by default.
- **BAND**: input, integer
  Indicates the starting band for the read. For BIP images, the default is 1. For BIL images, the starting band is incremented from the last read by default. For BSQ images, the starting band increments when each band is exhausted, that is, every NL lines.
- **NSAMPS**: input, integer
  Indicates the number of samples to be accessed in a single operation. Defaults to the number of pixels in an image line if the file organization is BSQ or BIL. Not yet available for BIP images.

- **NBANDS**: input, integer
  As in NLINES but for the band dimension. Currently only available for a file organization of BIP.
- **NLINES**: input, integer
  Indicates the number of lines to be accessed in a single operation. Defaults to 1.
- **IO_MESS:** input, string, maximum length 132
  Holds the user message that is issued under certain settings of the IO_ACT argument when I/O errors occur. Default is specified by **x/zveaction**.
- **OP**: input, string
  Indicates the intended operation. Valid values are 'READ', 'WRITE', and 'UPDATE'. Default is 'READ'. **Output files must have OP equal to WRITE.**
- **OPEN_ACT:** input, string
  Indicates the action to be taken by the Executive when an error condition is raised as a result of the open. A valid string may contain one or more of three characters in any order, 'A', 'U', 'S'. The presence of each character has the following meanings:
  - **A**: an error will cause an immediate program abort. Otherwise, the routine will return an error number in STATUS.
  - **U**: an error will cause the string that the user has specified with the OPEN_MES optional to be printed.
  - **S**: an error will cause a system message to be printed.

  The three values are independent. Thus 'SA' will cause a system message to be printed and the program to abort if an error occurs. If OPEN_ACT consists only of blanks, an error will cause the **x/zvopen** to return an error number in STATUS. Default is the action specified with **x/zveaction**.
- **U_FORMAT:** input, string
  Indicates the format of pixels in the buffer returned by **x/zvread** or the format of the pixels the program will pass to **x/zvwrit**. If for input files U_FORMAT is not equal to the input file format, then VICAR will convert. On output, if U_FORMAT is not equal to O_FORMAT then VICAR will convert. Default for input files is the input file format; for output files, it is the primary input U_FORMAT.

3.2.7 x/zvsignal—Signal an error

```
call xvsignal (unit, status, abend_flag)
status = zvsignal (unit, status, abend_flag)
```

x/zvsignal will signal an error if one has occurred in any of the image I/O routines (x/zv...) or label I/O routines (x/zl...). It uses the same internal routines that the OPEN_ACT, IO_ACT, LAB_ACT, and ERR_ACT optionals use, so no error checking is duplicated.

Arguments:

- **UNIT**: input, integer
  The unit number of a file. **x/zvsignal** will use this unit number to determine routine in which an error may have occured, and if it was a read or write, the line number in the image.
- **STATUS**: input, integer
  The status returned from a previous subroutine call to be checked.

- **ABEND_FLAG:** input, logical*4 or integer
  Flag to indicate whether ABEND (abnormal program terminator) should be called to terminate execution of the program. If ABEND_FLAG is TRUE (non-zero) and an error has occurred, **x/zvsignal** will not return and the program will terminate abnormally.

3.2.8 x/zvunit—Assign a unit number to a file

```
call xvunit(unit, name, instance, status, <optionals>, ' ')
status = zvunit(unit, name, instance, <optionals>, 0);
```

x/zvunit will return an unused file unit number. The programmer may use this number in any of the above routines. This routine must be called to obtain a unit number for every file that is to be accessed by the VICAR image and label I/O subroutine package.

Under UNIX, filenames accepted by the RTL (via **x/zvunit**, either the U_NAME optional or the INP or OUT parameters) can contain environment variables andusernames. A reference of the form $*var* will be replaced with the value of the environment variable *var*. The name of the environment variable (but not the dollar sign) may optionally be enclosed in curly braces (${*var*}). A tilde (~) followed by a username will be replaced with the home directory of that user. A tilde without a username will be replaced with the home directory of the current account. Both of these expansions exactly mimic the behavior of the C-shell, so they should be familiar to most UNIX users.

Arguments:

- **UNIT**: output, integer
  An unused file unit number that may be used with any of the image I/O routines.
- **NAME**: input, string, maximum length 32
  Indicates which file is to be associated with the returned unit number. By convention, the string 'INP' denotes the input files, and the string, 'OUT', the output files that appear on the command line. If NAME is any other string other than 'INP' or 'OUT', **x/zvunit** will create a new file for output and return the unit number for that file. The size and other attributes of the file will be determined by **x/zvopen** as for any output file asociated with 'OUT'.
- **INSTANCE**: input, integer
  Indicates which file of possibly several 'INP' or 'OUT' files is to be selected.
- **STATUS**: output, integer
  Error status indicator. A value of one indicates success. Errors are handled in the manner specified by **x/zveaction**.

Optional arguments:

- **U_NAME:** input, string, maximum length 132
  If the NAME argument is not 'INP' or 'OUT' then **x/zvunit** will return the unit number of a file not associated with any file name appearing on the command line. This argument is used to name that file. It may take any form that a command line file name may take. To get back unit numbers for more than one file, increment the INSTANCE argument for each file.

3.2.9 x/zvwrit—Write an image line

```
call xvwrit(unit, buffer, status, <optionals>, ' ')
status = zvwrit(unit, buffer, <optionals>, 0);
```

x/zvwrit will write a single line of pixels to the image file associatedwith the UNIT. The data will

be written from BUFFER and the status indicator will be returned in STATUS.

Arguments:

- **UNIT**: input, integer
  Unit number of file from **x/zvunit**.
- **BUFFER**: output, array of anything
  Array containing output data.

Optional arguments:

- **LINE**: input, integer
  Indicates the starting line for the read. For BSQ images, by default the starting line is incremented from its position in the last read until each band is exhausted (after NL reads), at which time it resets to the first line for the new band. For BIL images, the line number increments after NB reads, and for BIP images the line number increments after NS x NB reads.
- **SAMP**: input, integer
  Indicates the starting sample (starting pixel in the SAMP dimension) for the read. For BSQ and BIL images, the default is 1. For BIP images, the starting sample is incremented from the last read by default.
- **BAND**: input, integer
  Indicates the starting band for the read. For BIP images, the default is 1. For BIL images, the starting band is incremented from the last read by default. For BSQ images, the starting band increments when each band is exhausted, that is, every NL lines.
- **NSAMPS**: input, integer
  Indicates the number of samples to be accessed in a single operation. Defaults to the number of pixels in an image line if the file organization is BSQ or BIL. Not yet available for BIP images.
- **NBANDS**: input, integer
  As in NLINES but for the band dimension. Currently only available for a file organization of BIP.
- **NLINES**: input, integer
  Indicates the number of lines to be accessed in a single operation. Defaults to 1.
- **U_NL:** input, integer
  Indicates that the Executive is to create an output file capable of holding NL lines. If the file associated with the unit number already exists but is too small, it will be enlarged.

4. Label I/O

This Section describes the label I/O to be used with the MIPL VICAR image processing executive.

4.1 Introduction

This Section contains a programmer's guide for reading, adding, and deleting items in an image file label. A label describes the size, nature, processing history, origin and attributes of an associated image file. When an image is generated as an output file by an image processing task, the Executive will create an output label, combining items in the input file label with new label information derived from the current processing function. In this way, a history of the processing of the data contained in

the output file will be maintained. In addition, the Executive will guarantee that required label information, such as the output image size, will be correctly produced. The user and owner of a file may also modify the label of that file from the command language.

A programmer may modify the label of any file that is written or updated by a program. The routines described in this Section permit the programmer to make such modifications.

4.1.1 A Label Model

A label is composed of label items. A label item is a text key word of 32 character maximum size that identifies the label item, and a value which is the information part of the label item. A value may be multi-valued, that is, contain more than one value. As an example, a label item for an image file may be the dimension of the image, comprised of a key word DIM, and a value, 3. For another example, consider as a label item the image size. The key word might be SIZE, and the value, (800,800,3). This label item is multi-valued.

The set of keyword-value label items that comprise a label is considered by the executive to be partitioned into three sets of items, the system, history and property. The system items consist of those items that are independent of the history of the file. These items will include the size of the image, its organization, its pixel format, and the blocksize.

The organization of the label into system, history and property, and the further breakdown of the history into subsets based on processing task reflects the model of the image label that programmers have found to be convenient at MIPL.

System labels are defined by the VICAR RTL and contain all the information necessary for the RTL to access the image, such as size, pixel type, and host format information. System label items are not normally modified once the image is created. Since they are defined by the RTL, application programmers may not add their own system label items.

History labels contain the processing history of the image. Each time a VICAR task is run on an image, a new history task gets added to the history label of the image. The history labels are copied from the "primary input" file (usually the first input file), and the new task is appended to the end. Application programs are free to add label items as they wish to the history label.

Property labels are used to describe the current state (or properties) of the image. They should eventually take over that function from the history labels, leaving the history labels to be *only* history. This may take a while, due to the large amount of software and images that use the history labels, but the goal is to make property labels the only place for non-historical image labels.

Property labels are divided into named groups or sets called *properties*, much like history labels are divided into named groups called *tasks*. There is only one instance of each property name, unlike history tasks. For example, while there may be a half dozen LOGMOS tasks in the history label, there could be only one MIDR property in the property label.

Within each property group are individual label items. The label items look identical to their counterparts in the system and history labels. They may be string, integer, real, or double precision, and may have multiple values (i.e. an array). The keys for each label item must be unique within the property group, but may be duplicated between groups. The keys can be up to 32 characters long, just like any other label key. The label keys PROPERTY and TASK are reserved to separate property and history label sets, and may not be used by applications.

An example property label is listed (via `label-list`) below. The properties and names listed are examples, not official names.

```
----Property: TSTMAP ----
PROJ='mercator'
CENTER=(45.0, 12.7)
LINE=5
SAMP=5
SCALE=10.0
----Property: TSTLUT ----
RED=(1, 2, 3, 4)
GREEN=(4, 5, 6, 7)
BLUE=(7, 8, 9, 10)
```

4.1.2 Property Labels

Before property labels were introduced, history labels served a dual role: they contained processing history (which tasks were run in what order) along with information about the image's current state. A good example is the Magellan MIDR label. All descriptive information about the MIDR product is kept by convention in the first LOGMOS history task. They have no historical meaning whatsoever, as they are updated by later processing runs. They describe the current state of the image.

Another example is map labels, which describe the image's map projectio. A program that changes the projection adds a new map label entry to the history label. The last entry is the current projection. While in this case the map labels are historical, it is confusing to search through all the map labels to find the last (current) one. It is useful to keep a history of previous projections, but it is not clear to an inexperienced user which projection is current.

This dual role for history labels was confusing. Which of potentially several LOGMOS runs were the MIDR labels kept in? What task created the map labels? What was the actual processing history ? Since previous tasks were modified, some historical information was lost. These problems led to the creation of property labels.

Property labels are used in place of binary labels for most applications. Property labels do not suffer the same portability problems as binary labels, and may be written and read by standard VICAR RTL routines. They replace binary headers in most cases. Replacing binary prefixes may be more difficult. Although binary labels are still allowed replacing them with property labels is strongly encouraged.

4.1.2.1 Using Property Labels

Property labels are accessed in much the same way that history labels are. The routines **x/zladd**, **x/zldel**, **x/zlget**, and **x/zlinfo** have been modified to accept "PROPERTY" for the *type* argument, as well as "SYSTEM" and "HISTORY". Specify which property the label belongs to via the "PROPERTY" optional argument to these routines. Like history labels, property labels may have multiple instances, accessed through the "INSTANCE" keyword. The "HIST" key word is not used for property labels. In addition, a new routine, **x/zlpinfo**, has been added to get a list of properties in the label, similar to **x/zlhinfo**.

Internally, property labels are stored in between the system and history labels. A property set starts with a `PROPERTY=`*property-name*, label item, followed by all labels for that property. The property ends at the next PROPERTY key word or at the start of the history labels (indicated by a TASK key word).

Properties appear automatically when a label item for that property is added. There is no explicit

creation step to add a new property; just add a label item in that property and it will be created with an instance number of 1. The same is not true for deletion (via the RTL): if you want to delete an entire property delete all the elements from it and then delete the PROPERTY marker (using PROPERTY as the key word to delete from the property set). The `label-delete` program can do this. It is important to that this is exactly the way history labels work: tasks are created automatically when programs are run, but to delete a task delete all the key words for that task, including the marker labels (TASK, DAT_TIM, and USER).

Property labels, like history labels, are automatically copied from the primary input file to the output file. This is done because most properties will not change in a typical program run. Programs should update any properties that do change,. The copying of property labels can be controlled via **x/zvselpi** which allows you to change the source for the primary input file.

If you are implementing something new, like a look-up table or a new flight label, then you can use the property labels without problem. However, if you are moving an image that used to use history labels to the property label, like the map projection or an old flight label, you have to make sure that you can read files with the information either in the property label or in the history label. Old images with the information in the history label exist and will be used. For this case, you might want to adopt a similar strategy to the reading of other host's data: read the label from either the property or history label (wherever it is), but only write out property labels.

4.1.2.2 Property Instance Numbers

Properties record the current state of an image, while history labels record the operations that have been performed on an image. Property instances are used when multiple “current states" are valid. For example, when different methods may be used for to generate pointing correction, or different people do this manually, different pointing corrections will be obtained. These corrections are all valid for their intended use; since there is no way to determine the correct pointing, they are all simultaneously valid. Each pointing correction is stored as an instance of a single property.

There is no change to the label format in the file; a property instance is just another property set with the same name as a previous instance. There is no explicit identifier in the label. We suggest that you create a special label item in each multi-instance property type that identifies the type. Then you have two choices for finding a specific instance of a property. You can use **x/zlpinfo** to get a list, then cycle through all the instances of the property you care about, looking for the special label item. The alternative is to use **x/zlget** on the identifier for the property in question, incrementing the instance number until an error is returned, indicating that no more instances of this property exist.

4.1.2.3 Property Names

The RTL does no error or valid value checking on the name of a property, other than to ensure that it is 32 characters or less and is composed of printable ASCII characters. Any name at all can be used as a property name. However, there must be some control over property names, and the items that go in the property, or chaos will result.

It is up to the application programmer community to define how the property labels will be used, what they will be called, and what will go in them. The property name should be a short description of what the property is. For example, good names might be MAP, LUT, GLL-SSI, MGN-MIDR, or VIEWING-GEOMETRY (these examples, not official names). If it can be general purpose, make it so, otherwise include the project name in the property name. It is possible to put a version number in the property name if necessary, such as “MAP V2. 0”, but this should be done only if a major revision redefines the existing label items. Label items can be added to an existing property without changing a

version number. And if you have a version number in the name, all existing code that wants to find that property will have to be changed to include the version number.

In order to maintain a consistent set of names, a name registry (similar to the one for BLTYPE) has been established for properties. Every property name must be entered into this registry, with a pointer to documentation describing the label items that can appear in the property. If you want to create a new property, simply tell the keeper of the registry what name you want to use and what the label items that make it up are (either explicitly or by referring to a document). The registrar will check for duplicates, approve your request, and enter your name into the registry.

At the present time, the keeper of this registry is the VICAR system programmer.

The name registry system is voluntary; the RTL makes no checks on the validity of the names used. It is the responsibility of each individual programmer to make sure that they use this system. Failure to do so may result in your program not being accepted for delivery.

4.2 Image Label Access API

This Section describes the subroutines that make up the programmer's interface for label processing.

4.2.1 x/zladd Add information to an existing label item

```
call xladd(unit, type, key, value, status, <optionals>, ' ')
status = zladd(unit, type, key, value, <optionals>, 0);
```

x/zladd adds label items or values to multi-valued label items to a label. Strings in the image label are delimited by single quotes (`'`). A single quote in an input string will be repeated before being put in the label. On output (from **x/zlget**), the pair will be returned as one single quote. In other words, the repeating of quotes is transparent to the application program.

Arguments:

- **UNIT**: input, integer
  Unit of associated file.
- **TYPE**: input, string
  `PROPERTY', 'HISTORY' or 'SYSTEM'.
- **KEY**: input, string
  The key word of the label item.
- **VALUE**: input, array (size: NELEMENT)
  Holds the values of the label item. The type of argument required depends on the FORMAT optional given. For single values, INT requires an integer, REAL requires floating point (single-precision),

and STRING requires a simple string argument ((char *) in C, CHARACTER in FORTRAN). For multi-valued items (if NELEMENT is not 1), INT and REAL take single-dimension arrays of integers or reals. A STRING must be a two-dimensional array of characters in either FORTRAN or C.
For example, a string array capable of handling 10 elements, with 80 characters in each element, is declared in C: `char value[10][80];`
The standard C null string terminator is required, so the above array could only hold 79 characters. You will still pass in the length (in ULEN) as 80. ULEN is required for C strings that have more than one element. It is not legal to pass in an array of string pointers; the value must be a true array of characters.
In FORTRAN you use a CHARACTER array: `CHARACTER*80 VALUE (10)`
ULEN is not required, because the character descriptor contains the length of each string. Trailing spaces are stripped from FORTRAN strings by the executive; you can not add a label item that ends in a blank.

- **STATUS**: output, integer
  Error status. The possible errors are listed in 10 Appendix B: Error Messages (page 128).

Optional arguments:

- **ERR_ACT:** input, string
  Indicates the action to be taken by the Executive when an error condition is raised as a result of a call to this routine. A valid string may contain one or more of three characters in any order, 'A', 'U', 'S'. The presence of each character has the following meanings:

  - **A**: an error will cause an immediate program abort. Otherwise, the routine will return an error number in STATUS.
  - **U**: an error will cause the string that the user has specified with the ERR_MESS optional to be printed.
  - **S**: an error will cause a system message to be printed.

The three values are independent of each other. Thus 'SA' will cause a system message to be printed and the program to abort if an error occurs. If ERR_ACT consists only of blanks, an error will cause routine to return an error number in STATUS.
If ERR_ACT is not specified, then the action defaults to the value of the LAB_ACT optional to **x/zvopen** (the default for LAB_ACT is specified by **x/zveaction**).

- **ERR_MESS:** input, string
  Holds the user message that the programmer wishes printed when an error occurs as directed by ERR_ACT. Default is specified by **x/zveaction**.
  if the ERR_ACT option is not specified, and the LAB_ACT option given to **x/zvopen** specifies a message be printed, the message given by LAB_MESS, *not* ERR_MESS is printed.

- **FORMAT**: input, format
  Indicates the format of the given label item value. Valid values are: 'INT', 'REAL', 'DOUB' and 'STRING'. The VALUE array type should match FORMAT, i.e. float for REAL, double for DOUB, etc. See Table 5: C Declarations for Run-Time Library Arguments (page 23) for matching C declarations. See Table 6: FORTRAN declarations for pixel types (page 25) for matching FORTRAN declarations.

FORMAT is required. If the value in the FORMAT optional does not match the format of the currently existing label item, the label item is changed to match the optional. If this is not possible (i.e.,

a non-numeric string being changed to an integer), an error is returned.

- **LEVEL**: input, integer
  In the ADD routine, which allows label items to be added to a label, a level may be associated with a label item when that item is added. LEVEL holds the positive integer which is the level of the label item. (NOT YET AVAILABLE)

- **ELEMENT**: input, integer
  For a multi-value item, the starting point to add the VALUE. If there are 4 elements currently, and ELEMENT is 3, then the new value will become the third element, and the old third and fourth will either be moved down to make room (if MODE is 'INSERT'), or will be replaced (if MODE is 'REPLACE'). A value of -1 says to add the element at the end, after all existing elements. The default is -1, i.e. to add it at the end.

- **NELEMENT**: input, integer
  The number of values to be added to a multi-value item. Default: 1.

- **HIST**: input, string
  If TYPE is 'HISTORY' then this argument holds the name of the processing task that identifies the particular history subset in which 'KEY' may be found; if not entered, defaults to current task.

- **INSTANCE**: input, integer
  If TYPE is 'HISTORY', 'INSTANCE' specifies into which instance the name of the task from 'HIST' will be entered. If TYPE is 'PROPERTY', 'INSTANCE' specifies into which instance the name of the property set from 'PROPERTY' will be entered. Attempting to add a label to a non-esistent property instance will cause a new instance to be created. Instance numbers created may not match instance number entered. If there are 5 instances now, and you add instance 10, the label will go in a new instance 6. INSTANCE=0 forces creation of a new instance. Default: 1.

- **ULEN**: input, integer
  The length of an input string. This optional only applies if the FORMAT is STRING. For multi-valued items, ULEN is the maximum length of each string in the array, i.e. it is the size of the first dimension of the array. ULEN is required in some cases, and it is ignored in others. These are explained below.
  There are two ways to pass strings: C char and FORTRAN CHARACTER. See the VALUE argument for an example of each.

  - **C:** If NELEMENT is not 1 (multi-valued), then ULEN is required**.** If NELEMENT is 1, then ULEN is optional. If present, it is used, otherwise the length is the null-terminated length of the string.

  - **FORTRAN CHARACTER:** ULEN is always ignored. The length is obtained from the character descriptor.

- **MODE**: input, string
  Specifies what to do with labels that already exist. There are three allowable values: 'ADD', 'INSERT', and 'REPLACE'.

  - **'ADD':** this is the default action. If the label already exists, a DUPLICATE_KEY error is returned, and the label is not changed. If the label doesn't exist, it is created.

  - **'INSERT'**: If the label does not exist in the file, a new one is created. If it does exist, then all

elements that are added are inserted into the existing list of elements. All elements in the old label past the point of insertion are moved down to make room. For example, if the label has 4 elements, and you are adding 2 elements (NELEMENT is 2) at position 3 (ELEMENT is 3), then the two new elements would become numbers 3 and 4, and the old 3 and 4 would get moved to numbers 5 and 6.

- '**REPLACE**': If the label does not exist in the file, a new one is created. If it does exist, then all elements that are added replace any elements in the old label with the same sequence numbers (if any). For example, if the label has 5 element s, and you are adding 2 elements (NELEMENT is 2) at position 3 (ELEMENT is 3), then the two new elements would replace the old element numbers 3 and 4. Items 1, 2, and 5 remain unchanged. 'REPLACE' is not a reliable way to replace the entire contents of a label, as you do not always know ahead of time how many elements are present. To do that, call **x/zldel** and **x/zladd** in sequence.

- **PROPERTY**, string: The name of the property set that the label routine is accessing The PROPERTY optional argument is only valid if the TYPE argument of the above routines is set to "PROPERTY". See 4.1.2.1 Using Property Labels (page 67), for more information.

4.2.2 x/zldel Remove a label item

```
call xldel(unit, type, key, status, <optionals>, ' ')
status = zldel(unit, type, key, <optionals>, 0);
```

x/zldel removes a label item from a file that the user owns (an output file on a command line) or for which the user has update privilege. **x/zldel** permits the partial deletion of values in a multi-value label item.

Arguments:

- **UNIT**: input, integer
  Unit of associated file.

- **TYPE**: input, string
  `PROPERTY', 'HISTORY' or 'SYSTEM'.

- **KEY**: input, string
  The key word of the desired label item.

- **STATUS**: output integer
  Error status. The possible errors are listed in Appendix B: Error Messages (page 128).

Optional arguments:

- **ERR_ACT:** input, string
  Indicates the action to be taken by the Executive when an error condition is raised as a result of a call to this routine. A valid string may contain one or more of three characters in any order, 'A', 'U', 'S'. The presence of each character has the following meanings:

  - **A**: an error will cause an immediate program abort. Otherwise, the routine will return an error number in STATUS.

  - **U**: an error will cause the string that the user has specified with the ERR_MESS optional to be printed.

  - **S**: an error will cause a system message to be printed.

The three values are independent of each other. Thus 'SA' will cause a system message to be printed and the program to abort if an error occurs. If ERR_ACT consists only of blanks, an error will cause routine to return an error number in STATUS.
If ERR_ACT is not specified, then the action defaults to the value of the LAB_ACT optional to **x/zvopen** (the default for LAB_ACT is specified by **x/zveaction**).

- **ERR_MESS:** input, string
  Holds the user message that the programmer wishes printed when an error occurs as directed by ERR_ACT. Default is specified by **x/zveaction**.
  If the ERR_ACT option is not specified, and the LAB_ACT option given to **x/zvopen** specifies a message be printed, the message given by LAB_MESS, *not* ERR_MESS is printed.
- **ELEMENT**: input, integer
  For a multi-value item, the starting value to delete. Default: 1.
- **NELEMENT**: input, integer
  The number of values to be deleted from a multi-value item. A value of -1 means to delete all the values. The default is -1, i.e. to delete all the values.
- **NRET**: output, integer
  The number of values actually deleted by the call to **x/zldel**.
- **HIST**: input, string
  If TYPE is 'HISTORY' then this argument holds the name of the processing task that identifies the particular history subset in which 'KEY' may be found; if not entered, it defaults to the current task.
- **INSTANCE**: input, integer
  If TYPE is 'HISTORY' and HIST has been entered, INSTANCE gives the sequence number of the particular instance of the processing task for the condition where a processing task occurs more than once in the history subsets. If TYPE is 'PROPERTY' and PROPERTY has been entered, INSTANCE gives the sequence number of the particular instance of the property label for the condition where a property set with identical names occurs more than once. Default: 1.
- **PROPERTY**, string: The name of the property set that the label routine is accessing The PROPERTY optional argument is only valid if the type argument of the above routines is set to "PROPERTY". See 4.1.2.1 Using Property Labels (page 67), for more information.

4.2.3 x/zlget—Return the value of a label item

```
call xlget(unit, type, key, value, status, <optionals>, ' ')
status = zlget(unit, type, key, value, <optionals>, 0);
```

x/zlget returns the value or values associated with a label item key word. VALUE may be an array; the optional argument NELEMENT will indicate the dimension of VALUE.

Arguments:

- **UNIT**: input, integer
  Unit of associated file.
- **TYPE**: input, string
  `PROPERTY', 'HISTORY' or 'SYSTEM'.
- **KEY**: input, string
  The key word of the desired label item.

- **VALUE**: output, array (size:NELEMENT) Receives the values of the label item. The type of argument required depends on the FORMAT optional given. For single values, INT returns an integer, REAL returns floating point (single-precision), and STRING returns a simple string argument ((char *) in C and CHARACTER in FORTRAN). For multi-valued items (if NELEMENT is not 1), INT and REAL take single-dimension arrays of integers or reals. ASTRING must be a two-dimensional array of characters in either FORTRAN or C.

For example, if you wanted to have a string array capable of handling 10 elements, with 80 characters in each element, this is how you would declare it in C:

```
char value[10][80];
```

The standard C null string terminator is always returned in the string, so the above array could only hold 79 characters. You would still pass in the length (via ULEN) as 80.

From C, ULEN is required for multivalued strings (string arrays), as it specifies the inner dimension of the array. For scalar strings (not an array), ULEN is not strictly required as the length of the string is not required. BUT, you will have no protection against overflowing the buffer. Specifying ULEN for a scalar string ensures that an exceptionally long string value will not exceed the size of your buffer. So it is a good idea to always specify ULEN from C, although it's only really required for a string array.

```
In FORTRAN you use a CHARACTER array: CHARACTER*80 VALUE (10)
```

In this case ULEN is not required, as the character descriptor contains the length of each string. The space after the string itself is filled with blanks.

- **STATUS**: output, integer
  Error status. The possible errors are listed in 10 Appendix B: Error Messages (page 128).

Optional arguments:

- **ERR_ACT:** input, string
  Indicates the action to be taken by the Executive when an error condition is raised as a result of this call. A valid string may contain one or more of three characters in any order, 'A', 'U', 'S'. The presence of each character has the following meanings:
  - **A**: an error will cause an immediate program abort; otherwise, the routine will return with an error number in STATUS
  - **U**: an error will cause the string that the user has specified with the ERR_MESS optional to print
  - **S**: an error will cause a system message to print.

  The three values are independent of each other. Thus 'SA' will cause a system message to be printed and the program to abort if an error occurs. If ERR_ACT consists only of blanks, an error will cause routine to return an error number in STATUS.
  If ERR_ACT is not specified, then the action defaults to the value of the LAB_ACT optional to **x/zvopen** (the default for LAB_ACT is specified by **x/zveactiion**).

- **ERR_MESS:** input, string
  Holds the user message that the programmer wishes printed when an error occurs as directed by ERR_ACT. Default is specified by **x/zveaction**.
  if the ERR_ACT option is not specified, and the LAB_ACT option given to **x/zvopen** specifies a message be printed, the message given by LAB_MESS, *not* ERR_MESS is printed.

- **FORMAT**: input, format
  Indicates the format of the given label item value. Valid values are: 'INT', 'REAL', 'DOUB' and 'STRING'. The VALUE array type should match FORMAT, i.e. float for REAL, double for DOUB, etc. See Table 5: C Declarations for Run-Time Library Arguments (page 23) for matching C declarations. See Table 6: FORTRAN declarations for pixel types (page 25) for matching FORTRAN declarations.

In **the present implementation, FORMAT is required.** If the format does not match the format of the currently existing label item, the label item is changed to match what is given in this optional. If this is not possible (i.e., a non-numeric string being changed to an integer), an error is returned.

- **INSTANCE**: input, integer
  If TYPE is 'HISTORY' and HIST has been entered, INSTANCE gives the sequence number of the particular instance of the processing task for the condition where a processing task occurs more than once in the history subsets. If TYPE is 'PROPERTY' and PROPERTY has been entered, INSTANCE gives the sequence number of the particular instance of the property label for the condition where a property set with identical names occurs more than once. Default: 1.

- **LEVEL**: output, integer
  In the ADD routine, which allows label items to be added to a label, a level may be associated with a label item when that item is added. LEVEL returns the positive integer which is the level of the label item. (NOT YET AVAILABLE)

- **LENGTH**: output, integer
  The length of returned values. The length of 'INT' and 'REAL' formats is always 4. The length of a string item is the length of the longest string element, not including the null terminator (if present).

- **ELEMENT**: input, integer
  For a multi-value item, the starting value to return into VALUE. Default: 1.

- **NELEMENT**: input, integer
  The number of values to be returned of a multi-value item. A value of -1 means to return all elements. The default is 1.

- **NRET**: output, integer
  The number of values actually returned by the call to **x/zlget**.

- **HIST**: input, string
  If TYPE is 'HISTORY' then this argument holds the name of the processing task that identifies the particular history subset in which 'KEY' may be found; if not entered it becomes the current task.

- **PROPERTY**, string: The name of the property set that the label routine is accessing The PROPERTY optional argument is only valid if the type argument of the above routines is set to "PROPERTY". See 4.1.2.1 Using Property Labels (page 67), for more information.

- **ULEN**: input, integer
  The length of an input string. This optional only applies if the FORMAT is STRING. For multi-valued items, ULEN is the maximum length of each string in the array, i.e. it is the size of the first dimension of the array. ULEN is required in some cases, and it is ignored in others. These are explained below.
  There are two ways to pass strings: C char and FORTRAN CHARACTER. See the VALUE argument for an example of each.

  - **C:** If NELEMENT is not 1 (multi-valued), then ULEN is required**.** If NELEMENT is 1, then

ULEN is optional. If present, it is used, otherwise the length is the null-terminated length of the string.

- **FORTRAN CHARACTER:** ULEN is always ignored. The length is obtained from the character descriptor.

4.2.4 x/zlhinfo—Return history label information

```
call xlhinfo(unit, tasks, instances, nhist, status, <optionals>, ' ')
status = zlhinfo(unit, tasks, instances, nhist, <optionals>, 0);
```

A history label is partitioned into subsets, one for each processing task that has 'read' or 'written' the data in the file described by the label. **x/zlhinfo** returns the names and instances of the history label subsets for the file associated with UNIT.

Arguments:

- **UNIT**: input, integer
  The unit number of the file described by the label of interest.
- **TASKS**: output, string, fixed length 32 FORTRAN, 33 in C bytes
  An array of the task names in order of occurrence. At the present time the length of a task name is still truncated to 8 (only 8 are significant), but when calling **x/zlhinfo** you should now provide a 32 character buffer (33 in C for the null terminator) in order to accomodate future expansion.
- **INSTANCES**: output, integer array
  An array with the same number of elements as TASKS giving the instance of each task. If, for instance, the task A is the first third and sixth element in TASKS, then INSTANCES (1)=1, INSTANCES (3)=2, and INSTANCES (6)=3.
- **NHIST**: input/output, integer
  On input, it is the max dimension of TASKS and INSTANCES. On output, it is the number of tasks in the current label.
- **STATUS**: output, integer
  The error status. The values are listed in 10 Appendix B: Error Messages (page 128).

Optional arguments:

- **ERR_ACT:** input, string
  Indicates the action to be taken by the Executive when an error condition is raised as a result of this call. A valid string may contain one or more of three characters in any order, 'A', 'U', 'S'. The presence of each character has the following meanings :
  - **A**: an error will cause an immediate program abort; otherwise, the routine will return with an error number in STATUS
  - **U**: an error will cause the string that the user has specified with the ERR_MESS optional to print
  - **S**: an error will cause a system message to print

The three values are independent of each other. Thus 'SA' will cause a system message to be printed and the program to abort if an error occurs. If ERR_ACT consists only of blanks, an error will cause routine to return an error number in STATUS.
If ERR_ACT is not specified, then the action defaults to the value of the LAB_ACT optional to

x/zvopen (the default for LAB_ACT is specified by **x/zveaction**).

- **ERR_MESS:** input, string
  Holds the user message that the programmer wishes printed when an error occurs as directed by ERR_ACT. Defaults is specified by **x/zveaction**.
  if the ERR_ACT option is not specified, and the LAB_ACT option given to **x/zvopen** specifies a message be printed, the message given by LAB_MESS, *not* ERR_MESS is printed.
- **NRET**
  Returns the total number of tasks in the label. This can differ from the NHIST argument if the buffers provided were too small. NRET returns the total number of tasks available, while NHIST returns the number actually in the buffers.
- **ULEN** specifies the length of each element in the TASKS array. You can specify any length (up to the maximum of 32 plus one for the terminator) for task names. In FORTRAN, the FORTRAN string length is used. In C, the length comes from ULEN (and defaults to 8 if ULEN is not given for backwards compatibility). Use a ULEN of at least 9 from C, in order to get all 8 characters with a null terminator.

4.2.5 x/zlinfo—Return information about a single label item

```
call xlinfo(unit,type,key,format,maxlen,nelement,status,<optionals>,'
')
status = zlinfo(unit,type,key,format,maxlen,nelement,<optionals>, 0);
```

x/zlinfo will return certain information about a single label item of given TYPE and KEY.

Arguments:

- **UNIT**: input, integer
  Unit of associated file.
- **TYPE**: input, string
  `PROPERTY', 'HISTORY' or 'SYSTEM'.
- **KEY**: input, string
  The key word of the desired label item.
- **FORMAT**: output, string, maximum length 8
  FORMAT indicates the format of the label item. Valid values are: 'INT', 'REAL', or 'STRING'.
- **MAXLENGTH**: output, integer
  Indicates the maximum length of any value associated with this key word. The length of 'INT' and 'REAL' items is always 4, meaning 4 bytes. The length of a 'STRING' item does *not* include the null terminator, so if you are dynamically allocating an array in C to hold the values, dimension it to MAXLENGTH + 1 to leave room for the null terminator.
- **NELEMENT**: output, integer
  Returns the number of values associated with the entered key word.
- **STATUS**: output, integer
  The error status. The values are listed in 10 Appendix B: Error Messages (page 128).

Optional arguments:

- **ERR_ACT:** input, string

Indicates the action to be taken by the Executive when an error condition is raised as a result of this call. A valid string may contain one or more of three characters in any order, 'A', 'U', 'S'. The presence of each character has the following meanings:

- **A**: an error will cause an immediate program abort; otherwise, the routine will return with an error number in STATUS
- **U**: an error will cause the string that the user has specified with the ERR_MESS optional to print
- **S**: an error will cause a system message to print

- The three values are independent of each other. Thus 'SA' will cause a system message to be printed and the program to abort if an error occurs. If ERR_ACT consists only of blanks, an error will cause routine to return an error number in STATUS.
  If ERR_ACT is not specified, then the action defaults to the value of the LAB_ACT optional to **x/zvopen** (the default for LAB_ACT is specified by **x/zveaction**).
- **ERR_MESS:** input, string
  Holds the user message that the programmer wishes printed when an error occurs as directed by ERR_ACT. Default is specified by **x/zveaction**.
  If the ERR_ACT option is not specified, and the LAB_ACT option given to **x/zvopen** specifies a message be printed, the message given by LAB_MESS, *not* ERR_MESS is printed.
- **MOD**: output, integer
  Indicates whether the label item is modifiable by the programmer. Values: 0 modifiable, 1 not modifiable. (NOT YET AVAILABLE)
- **STRLEN**: output, integer
  This optional returns the maximum length of any label element when the elements are treated as strings. For STRING format labels, this is identical to MAXLENGTH. For INT or REAL formats, the label is treated as if the integer or real values were strings, and the length of the character representation of the value is returned. This is needed in combination with calling **x/zlget** with a FORMA T of STRING, which causes all items to be returned as strings. This is useful when printing the values, as no format conversion is required.
- **INSTANCE**: input, integer
  If TYPE is 'HISTORY' and HIST has been entered, INSTANCE gives the sequence number of the particular instance of the processing task for the condition where a processing task occurs more than once in the history subsets. If TYPE is 'PROPERTY' and PROPERTY has been entered, INSTANCE gives the sequence number of the particular instance of the property label for the condition where a property set with identical names occurs more than once. Default: 1.
- **HIST**: input, string
  If TYPE is 'HISTORY' then this argument holds the name of the processing task that identifies the particular history subset in which 'KEY' may be found; if not entered, defaults to current task.
- **PROPERTY**, string: The name of the property set that the label routine is accessing The PROPERTY optional argument is only valid if the TYPE argument of the above routines is set to "PROPERTY". See 4.1.2.1 Using Property Labels (page 67), for more information.

4.2.6 x/zlninfo—Return name of next key

```
call xlninfo(unit,key,format,maxlength,nelement,status,<optionals>,' ')
```

```
status = zlninfo(unit,key,format,maxlength,nelement,<optionals>, 0);
```

x/zlninfo operates much like the routine **x/zlinfo** except that **x/zlninfo** does not expect an input for KEY. Instead, **x/zlninfo** will return a string in KEY which holds the next label item key word in the sequence of label items for the given TYPE. In addition, **x/zlninfo** will return the same information as **x/zlinfo**. If the last available label item has already been returned then that condition will be indicated in STATUS.

Arguments:

- **UNIT**: input, integer
  Unit of associated file.
- **KEY**: output, string, maximum length 32
  The key word of the next label item.
- **FORMAT**: output, string, maximum length 8
  Indicates the format of the label item. Values: 'INT', 'REAL', 'STRING'.
- **MAXLENGTH**: output, integer
  Indicates the maximum length of any value associated with this key word. The length of 'INT' and 'REAL' items is always 4, meaning 4 bytes. The length of a 'STRING' item does *not* include the null terminator, so if you are dynamically allocating an array in C to hold the values, dimension it to MAXLENGTH + 1 to leave room for the null terminator.
- **NELEMENT**: output, integer
  Returns the number of values associated with the entered key word.
- **STATUS**: output, integer
  The error status. The values are listed in 10 Appendix B: Error Messages (page 128).

Optional arguments:

- **ERR_ACT:** input, string
  Indicates the action to be taken by the Executive when an error condition is raised as a result of this call. A valid string may contain one or more of three characters in any order, 'A', 'U', 'S'. The presence of each character has the following meanings:
  - **A**: an error will cause an immediate program abort; otherwise, the routine will return with an error number in STATUS
  - **U**: an error will cause the string that the user has specified with the ERR_MESS optional to print
  - **S**: an error will cause a system message to print
- The three values are independent of each other. Thus 'SA' will cause a system message to be printed and the program to abort if an error occurs. If ERR_ACT consists only of blanks, an error will cause routine to return an error number in STATUS.
  If ERR_ACT is not specified, then the action defaults to the value of the LAB_ACT optional to **x/zvopen** (the default for LAB_ACT is specified by **x/zveaction**).
- **ERR_MESS:** input, string
  Holds the user message that the programmer wishes printed when an error occurs as directed by ERR_ACT. Default is specified by **x/zveaction**.
  If the ERR_ACT option is not specified, and the LAB_ACT option given to **x/zvopen** specifies a

message be printed, the message given by LAB_MESS, *not* ERR_MESS is printed.

- **MOD**: output, integer
  Indicates whether the label item is modifiable by the programmer. Values: 0 modifiable, 1 not modifiable. (NOT YET AVAILABLE)

- **STRLEN**: output, integer
  This optional returns the maximum length of any label element when the elements are treated as strings. For STRING format labels, this is identical to MAXLENGTH. For INT or REAL formats, the label is treated as if the integer or real values were strings, and the length of the character representation of the value is returned. This is needed in combination when calling **x/zlget** with a FORMAT of STRING, which causes all items to be returned as strings. This is useful when printing the values, as no format conversion is required.

4.2.7 x/zlpinfo—Returns the names of property subsets in the given file

```
call xlpinfo(unit, properties, nprop, status, <optionals>, ' ')
status = zlpinfo(unit, properties, nprop, <optionals>, 0);
```

Returns the names of property subsets in the given file. Property labels are broken up into subsets, each with a property name. This routine returns a list of all property names in the file specified by UNIT, which must be open. This routine is identical to **xlhinfo**, except that it returns property names instead of task names and instances.

Arguments:

- **UNIT**: integer, input
  UNIT is the unit number of an open file. The property names in the property label of this file are returned.

- **PROPERTIES**: string array, output
  PROPERTIES is a string array that gets the list of property names. When **x/zlpinfo** is called from C, the size of each string in the array is given by the ULEN optional argument (which is required from C). There is no 8-character default as there is in **zlhinfo**. From FORTRAN, ULEN is optional, since the string length is obtained from the array itself (which must be a CHARACTER*n array). Since property names can be up to 32 characters, you should declare a FORTRAN array to be at least CHARACTER*32, and a C array should be at least 33 characters long (one additional character for the null terminator). The number of strings in the array is specified by the NPROP argument.

- **NPROP**: integer, input/output
  On input, NPROP is the major dimension (maximum number of strings) in PROPERTIES. On output, it returns the number of properties returned in PROPERTIES. If there are more properties than the input NPROP allows, then the returned NPROP will be the same as the input since the maximum number of properties are returned. If you want the total number of properties in the label, regardless of the size of your supplied buffer, use the NRET optional argument.

- **STATUS**: output, integer
  The error status. The values are listed in 10 Appendix B: Error Messages (page 128).

Optional Arguments:

- **ERR_ACT**: string, input
  Indicates the action to be taken by the RTL when an error occurs in this routine. A valid string may

contain one or more of the characters listed below, in any order.

- **A** : Abort the program on error; otherwise the routine returns with STATUS set.
- **U** : Print the string specified in ERR_MESS if an error occurs.
- **S** : Print a system error message describing the error.

The three values are independent of each other. Thus, “SA” will cause a system error message to be printed and the program to abort if an error occurs. If ERR_ACT is empty or contains only blanks, no action will be taken on error, and the error code will be returned in STATUS.
If ERR_ACT is not specified, then the action defaults to the value of the LAB_ACT optional to **x/zvopen**. The default for LAB_ACT is specified by **x/zveaction**.

- **ERR_MESS:** string, input
  Specifies the user message to be printed when an error occurs as directed by ERR_ACT. If the ERR_ACT option is not specified, and the LAB_ACT option given to **x/zvopen** specifies that a message be printed, the message given by LAB_ACT, *not* ERR_MESS, is printed.
- **NRET**, integer, output
  NRET returns the total number of properties in the label. This can differ from the NPROP argument if the buffer provided was too small. NPROP returns the number actually in the buffer, while NRET returns the total number available.
- **ULEN**, integer, input
  ULEN specifies the length of each element in the PROPERTIES array. ULEN is required on **x/zlpinfo** when called from C to define the inner dimension of the array. Make sure to leave space for the null terminator, so ULEN should be at least 33. From FORTRAN, ULEN is optional. If it is not given, the string length will be picked up from the string array itself. The CHARACTER*n variable should be at least 32 characters.
- **INST_NUM**, integer array, output.
  An array with the same number of elements as NPROPS giving the instance of each property. INST_NUM returns a list of instances that match the names already returned. For example, if the property A is the first third and sixth element in PROPERTIES, then INST_NUM (1)=1, INST_NUM (3)=2, and INST_NUM (6)=3. If INST_NUM is not supplied, you won't get instance numbers, but you will still get repeated occurrences of names in the property name list if multiple instances are present.

5. Parameter I/O

5.1 Introduction

A user of an application program under the TAE controls the behavior of the program by entering parameters either on the command line of the program, via the tutor mode, or in response to an interactive prompt from the program.

Parameters may be of type real, key word, integer, or string, and may be singly or multiply valued.

Input and output files to an application program are considered parameters as well, of type string.

5.2 Parameter I/O API

VICAR has sixteen parameter retrieval routines that may be called from an application program:

5.2.1 x/zvintract—Prompt user for interactive command

```
call xvintract( subcmd, prompt)
zvintract( subcmd, prompt)
```

A call to this subroutine initiates a TAE interactive parameter session and returns in a buffer the parameters input by the user. This buffer is not visible to the program, but these parameters can be retrieved by the program using subroutine **x/zviparm**.

Arguments:

- **SUBCMD**: input, string
  Subcommand in the PDF. The string may contain up to 9 characters, and it maybe specified either as a FORTRAN CHARACTER type (by descriptor) or as a char* for C. If it's an empty string or a single blank, the entire PDF is used. This string may include the leading dash (“-”); one will be added if the programmer omits it. If this string only contains a dash, then the user will be required to specify a subcommand name as the first interactive input item.
- **PROMPT**: input, string
  Prompt string to be used in the interactive parameter session. The string may contain up to 20 characters, and it may be specified either as a FORTRAN CHARACTER type(by Descriptor)) or as a char* for C. If it's an empty string or a single blank, the TAE default (a list of the parameters) is used.

5.2.2 x/zviparm—Return interactive parameter values

```
call xviparm(name, value, count, def, maxcnt)
status = zviparm(name, value, count, def, maxcnt, length);
```

x/zviparm returns information about the value(s) specified for a parameter interactively, given a

parameter name. The returned information includes:

- The value(s) specified by the user or by default.
- The count, i.e. number of values specified for the parameter.
- The default flag. This flag should not be used. It is maintained for compatibility reasons, but the parameter count should be used instead.

Both the parameter name and, if it is a character string, the value may be specified as CHARACTER*n type for FORTRAN or char for C.

For multivalued strings, the value should be a two-dimensional array of char (C) or an array of CHARACTER*n (Fortran). For C, the length of the inner dimension must be specified via the "length" parameter. If the length is 0, or a BYTE array is passed in Fortran, then the strings will be returned in the old x/zvsptr format, which has been deprecated and should no longer be used.

A call to subroutine **x/zvintract** is required before this routine is called, in order to initiate an interactive parameter session.

If an error occurs, such as the parameter wasn't found, then COUNT is returned as 0, and the action specified by **x/zveaction** is taken.

Arguments:

- **NAMARG**: input, string, maximum length 8
  NAMARG specifies the parameter name as given in the PDF.
- **VALARG**: output, string
  If integer/real: parameter value(s) returned. This may be an array of string: argument for returned string. CHARACTER*n type for FORTRAN or char for C. For multivalued strings, the value should be a two-dimensional array of char (C) or an array of CHARACTER*n (Fortran). For C, the length of the inner dimension must be specified via the "length" parameter.
- **COUNT**: output, integer
  COUNT gives the number of parameter values returned. If zero, the parameter was not found.
- **DEF**: output, integer
  DEF is the default flag. If DEF = 0, then values are user-specified. If DEF = 1, then values are PDF defaults. This flag should not be used. It is maintained for compatibility reasons, but the parameter count should be used instead. If you want to have a program-supplied default, then allow the parameter to be nullable. If there are no values entered, then you can provide a default value in the program. Otherwise, the defaults in the PDF should be real default values, and it shouldn't matter whether the user entered them or not.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107).

Optional arguments:

- **MAXCNT**: input, integer
  The maximum number of parameter values to be returned. This should be the dimension of the parameter array in the calling routine.

5.2.3 x/zvip—Interactive version of x/zvp; abbreviated version of

x/zviparm

```
call xvip(name, value, count)
status = zvip(name, value, count);
```

Interactive version of **x/zvp**; abbreviated version of **x/zviparm**. Returns the value(s) of the given interactive parameter, and the number of values.

Arguments:

- **NAME**: string, input
  NAME is the name of the interactive parameter to get values from.
- **VALUE**: parameter—value-array, output
  The value of the parameter is returned in the VALUE array. The type of the value depends on the type of the parameter, which is either INTEGER, STRING or REAL (single-precision).
  There is no provision for providing a string length under C. Therefore, **zvip** should *not* be used for a string array. It is okay for a single string value, but if you want to get a multi-valued string, use **zviparm** instead. If you do get a multi valued string with **zvip**, it is returned in the **zvsptr** packed format, which is obsolete and should not be used. In FORTRAN, this restriction does not apply, since the string length for each element can be obtained from the string itself.
  There is no way to specify the size of the value buffer, so make sure it is big enough to handle the maximum count allowed in the PDF.
- **COUNT**: integer, output
  The number of values in the parameter is returned in COUNT.
  Warning: In **xvp**, the value of COUNT is set to 0 if the parameter has been defaulted. This is actually a design flaw, but has been retained for compatibility. The other three routines, **zvp**, **xvip**, and **zvip**, do not have this flaw. Being new routines, they can be done correctly without compatibility problems. The COUNT parameter returns the actual number of parameters in the value parameter, whether they are defaults or not. However, the variant behavior of **xvp** is likely to cause some confusion. Do not depend on this behavior in **xvp**, as it may change to be consistent in the future. If you are using **xvp**, either construct the PDF such that this problem won't matter, or use **xvpcnt** to get the actual count. This is a design flaw for two reasons: With the count returned as 0, there is no way to know if there is anything valid in the VALUE parameter, so it becomes useless. Second, the default flag should never be used; it is maintained for compatibility purposes only. The count of the parameter should be used instead.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107).

5.2.4 x/zviparmd—Interactive version of x/zvparmd

```
call xviparmd(name, value, count, def, maxcnt)
status = zviparmd(name, value, count, def, maxcnt, length);
```

Interactive version of **x/zvparmd**.

This routine is exactly like **x/zviparm**, except that if the interactive parameter being returned is REAL, the VALUE parameter is returned in double precision format. **x/zviparm** returns the value in single precision. Although VALUE supports integers and strings as well, **x/zviparmd** should generally be used only for double-precision numbers. **x/zviparm** should be used for integers and strings.

This routine replaces the functionality of the R8FLAG parameter that was previously on **x/zviparm.**

Arguments:

- **NAME**: string, input
  NAME is the name of the interactive parameter to get values from.
- **VALUE**: parameter—value-array, output
  The value of the parameter is returned in the VALUE array. The type of the value depends on the type of the parameter, which is either INTEGER, STRING, or REAL (double-precision).
- **COUNT**: integer, output
  Reports the number of values returned in the VALUE parameter. A COUNT of 0 means the parameter either had a null value or was not found.
- **DEF**: integer, output
  Returns 1 if the parameter was defaulted, and 0 otherwise.
  NOTE: The DEF flag is obsolete and should not be used.
- **MAXCNT**: integer, input
  Specifies the maximum number of values to return. 0 means no limit.
- **LENGTH**: integer, input
  Specifies the length of each string if a string array is passed in for VALUE. Useful only from C; FORTRAN gets string lengths automatically. If the parameter is not a string, or is only a single string, set LENGTH to 0.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107).

5.2.5 x/zvipcnt—Return the count of a parameter.

```
call xvipcnt(name, count)
status = zvipcnt(name, count);
```

x/zvpcnt returns the parameter size information for an interactive parameter whose name is passed in the NAMARG parameter.

If only two arguments are given, the actual count, or dimension of the parameter is passed back in COUNT.

If an error occurs, such as the parameter wasn't found, then COUNT is returned as 0, and the action specified by **x/zveaction** is taken.

x/zvipcnt may only be called after the user is prompted with **x/zvintract**.

Arguments:

- **NAMARG**: input, string
  The name of the parameter whose count is desired. NAMARG may be passed either by descriptor (FORTRAN character) or by reference.
- **COUNT**: output, integer
  The actual count of the parameter. For example, if the parameter is declared in the PDF file as

having a count of up to 3, as in `PARM I TYPE=INTEGER COUNT=1:3,` and if the user specifies I=(4,7), then COUNT will be returned as 2.

- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107).

5.2.6 x/zvipone—Interactive version of x/zvpone

```
status = xvipone(name, value, instance, maxlen)
status = zvipone(name, value, instance, maxlen);
```

Interactive version of **x/zvpone**.

This routine returns a single value from a multi-valued interactive parameter. It is most useful to get a string from a list of strings without having to mess with string arrays, but can be used for integer or real (single-precision) values as well. Note that **x/zvipone** is a FORTRAN function with a status return, which differs from most FORTRAN RTL routines**.**

Arguments:

- **NAME**: string, input
  NAME is the name of the interactive parameter to get a value from.
- **VALUE**: parameter—value, output
  The value of the parameter is returned in VALUE. The type of the value depends on the type of the parameter, which is either INTEGER, STRING, or REAL (single-precision). There is no equivalent to **x/zvipone** for double precision floating point; use **x/zviparmd** instead.
- **INSTANCE**: integer, input
  INSTANCE specifies which value you want. INSTANCE starts counting at 1, so the fifth value would have an INSTANCE of 5.
- **MAXLEN**: integer, input
  MAXLEN specifies the maximum length of the string buffer if the parameter is a string. It is used to avoid overflowing your buffer. If the parameter is not a string, or you don't care, set MAXLEN to 0. MAXLEN is rarely needed in FORTRAN, since string lengths are available from the strings themselves.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107).

5.2.7 x/zvipstat Interactive version of x/zvpstat

```
call xvipstat(name, count, def, maxlen, type)
status = zvipstat(name, count, def, maxlen, type);
```

This routine returns information about an interactive parameter without returning its value. It is most useful to get the maximum length of any string and the number of strings in order to allocate a buffer before calling x/zviparm. This routine is also the only way to determine the data type of a parameter given only its name. The program should know the type in the PDF, but there are situations where this could be useful.

Arguments:

- **NAME**: string, input
  NAME is the name of the interactive parameter to get information about.
- **COUNT**: integer, output
  Returns the number o f items in the parameter. If 0, parameter is either not found or is null.
- **DEF**: integer, output
  Returns 1 if the parameter was defaulted, and 0 otherwise.
  NOTE: The DEF flag is obsolete and should not be used.
- **MAXLEN**: integer, output
  Returns the maximum length of any value in the parameter. For REAL and INT, it is just the size of a REAL or INT. For STRING, it is the length of the longest string in the parameter. It does not include the null terminator, so you should add one before allocating a buffer in C.
- **TYPE**: string, output
  Returns the data type of the parameter. The possible values are “INT”, “REAL”, and “STRING”.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107).

5.2.8 x/zviptst Interactive version of x/zvptst

```
GIVEN = xviptst ( KEY )
GIVEN = zviptst ( KEY );
```

The interactive version of x/zvptst. This function returns true if a key word was specified and false if not. For example, a program could check if the value “HALF” was given for a key word parameter with the code:

```
logical lhalf,xviptst

      lhalf = xviptst('HALF')
      if (lhalf) then
        .
        .
        .
```

The key word must be a value specified for a parameter of type KEY WORD.

xviptst must be declared LOGICAL in a FORTRAN program calling it.

x/zviptst, like the other **x/zvi** routines, may only be called after the user is prompted with **x/zvintract**.

Arguments:

- **KEY**: input, string
  Name of key word to be tested for. Note this is not the parameter name, but one of the valid values for the key word parameter. Passed either by descriptor in FORTRAN (if CHARACTER string is passed) or reference in C (if char* is passed).

5.2.9 x/zvp Abbreviated version of x/zvparm

```
call xvp(name, val, count)
status = zvp(name, val, count)
```

x/zvp returns information about the value(s) specified for a parameter whenthe program was run, given a parameter name. It is an abbreviated version of **x/zvparm**, for user convenience. The returned information includes:

- The value(s) specified by the user or by default.
- The count, i.e. number of values specified for the parameter.

Both the parameter name and, if it is a character string, the value may be specified as CHARACTER*n type for FORTRAN or char for C.

For multivalued strings, the value should be a two-dimensional array of char (C) or an array of CHARACTER*n (Fortran). For C, the length of the inner dimension must be specified via the "length" parameter. If the length is 0, or a BYTE array is passed in Fortran, then the strings will be returned in the old x/zvsptr format, which has been deprecated and should no longer be used.

If an error occurs, such as the parameter wasn't found, then COUNT is returned as 0, and the action specified by **x/zveaction** is taken.

Arguments:

- **NAME**: input, string
  NAME is the parameter name argument. This specifies the parameter name. It maybe either a descriptor, if a FORTAN CHARACTER type is passed, or reference (memory address), if a char (C) array is passed).
- **VALARG**: output, string
  If integer/real: parameter value(s) returned. This may be an array of string: argument for returned string. CHARACTER*n type for FORTRAN or char for C. For multivalued strings, the value should be a two-dimensional array of char (C) or an array of CHARACTER*n (Fortran). For C, the length of the inner dimension must be specified via the "length" parameter.
- **COUNT**: output, integer
  COUNT gives the number of parameter values returned. If zero, the parameter was not found or the parameter was defaulted by the user. If the parameter was defaulted (given a default in the PDF but not by the user), then COUNT is returned as 0, regardless of the actual number of parameters in the defaulted value. This is a design flaw that must be maintained for historical reasons.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107).

5.2.10 x/zvparm—Return a parameter value

```
call xvparm( namarg, valarg, count, def, maxcnt)
status = zvparm( namarg, valarg, count, def, maxcnt, length)
```

x/zvparm returns information about the value(s) specified for a parameter when the program was run, given a parameter name. The returned information includes:

- The value(s) specified by the user or by default.
- The count, i.e. number of values specified for the parameter.
- The default flag. This flag should not be used. It is maintained for compatibility reasons, but the parameter count should be used instead.

Both the parameter name and, if it is a character string, the value may be specified as CHARACTER*n type for FORTRAN or char for C.

For multivalued strings, the value should be a two-dimensional array of char (C) or an array of CHARACTER*n (Fortran). For C, the length of the inner dimension must be specified via the "length" parameter. If the length is 0, or a BYTE array is passed in Fortran, then the strings will be returned in the old x/zvsptr format, which has been deprecated and should no longer be used.

If an error occurs, such as the parameter wasn't found, then COUNT is returned as 0, and the action specified by **x/zveaction** is taken.

Arguments:

- **NAMARG**: input, string
  NAMARG is the parameter name argument. It specifies the parameter name. It may be either a descriptor, if a CHARACTER (FORTRAN) type is passed, or reference (memory address), if a char (C) array is passed.
- **VALARG**: output, string
  If integer/real: parameter value(s) returned. This may be an array of string: argument for returned string. CHARACTER*n type for FORTRAN or char for C. For multivalued strings, the value should be a two-dimensional array of char (C) or an array of CHARACTER*n (Fortran). For C, the length of the inner dimension must be specified via the "length" parameter.
- **COUNT**: output, integer
  COUNT gives the number of parameter values returned. If zero, the parameter was not found.
- **DEF**: output, integer
  DEF is the default flag. If DEF = 0, then values are user-specified. If DEF = 1, then values are PDF defaults. This flag should not be used. It is maintained for compatibility reasons, but the parameter count should be used instead. If y o u want to have a program-supplied default, then allow the parameter to be nullable. If there are no values entered, then you can provide a default value in the program. Otherwise, the defaults in the PDF should be real default values, and it shouldn't matter whether the user entered them or not.
- **MAXCNT**: input, integer
  MAXCNT is the maximum number of parameter values to be returned. This should be the dimension of the parameter array in the calling routine.
- **LENGTH**: integer, input
  Specifies the length of each string if a string array is passed in for VALUE. Useful only from C; FORTRAN gets string lengths automatically. If the parameter is not a string, or is only a single string, set LENGTH to 0.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107).

5.2.11 x/zvparmd—Double-precision version of x/zvparm

```
call xvparmd(name, value, count, def, maxcnt)
status = zvparmd(name, value,count, def, maxcnt, length);
```

Double-precision version of **x/zvparm**.

This routine is exactly like **x/zvparm**, except that if the parameter being returned is REAL, the VALUE parameter is returned in double precision format. **x/zvparm** returns the value in single precision. Although VALUE supports integers and strings as well, **x/zvparmd** should generally be used only for double-precision numbers. **x/zvparm** should be used for integers and strings.

This routine replaces the functionality of the R8FLAG parameter that was previously on **x/zvparm.**

Arguments:

- **NAME**: string, input
  NAME is the name of the parameter to get values from.
- **VALUE**: parameter—value-array, output
  The value of the parameter is returned in the VALUE array. The type of the value depends on the type of the parameter, which is either INTEGER, STRING, or REAL (double-precision).
- **COUNT**: integer, output
  Reports the number of values returned in the VALUE parameter. A COUNT of 0 means the parameter either had a null value or was not found.
- **DEF**: integer, output
  Returns 1 if the parameter was defaulted, and 0 otherwise.
  NOTE: The DEF flag is obsolete and should not be used.
- **MAXCNT**: integer, input
  Specifies the maximum number of values to return. 0 means no limit.
- **LENGTH**: integer, input
  Specifies the length of each string if a string array is passed in for VALUE. Useful only from C; FORTRAN gets string lengths automatically. If the parameter is not a string, or is only a single string, set LENGTH to 0.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107).

5.2.12 x/zvpcnt—Return the count of a parameter.

```
call xvpcnt( namarg, count)
status = zvpcnt( namarg, count)
```

x/zvpcnt returns the parameter size information for a given parameter whose name is passed in the NAMARG parameter.

If an error occurs, such as the parameter wasn't found, then COUNT is returned as 0, and the action specified by **x/zveaction** is taken.

Arguments:

- **NAMARG**: input, string
  The name of the parameter whose count is desired. NAMARG may be passed either by descriptor (FORTRAN character) or by reference.
- **COUNT**: output, integer
  The actual count of the parameter. For example, if the parameter is declared in the PDF file as

having a count of up to 3, as in `PARM I TYPE=INTEGER COUNT=1:3,` and if the user specifies I=(4,7), then COUNT will be returned as 2.

- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107).

5.2.13 x/zvpone Single value from a multivalued parameter

```
status = xvpone(name, value, instance, maxlen)
status = zvpone(name, value,instance, maxlen);
```

This routine returns a single value from a multi-valued parameter. It is most useful to get a string from a list of strings without having to mess with string arrays, but can be used for integer or real (single-precision) values as well. **xvpone** is a FORTRAN function with a status return, which differs from most FORTRAN RTL routines**.**

Arguments:

- **NAME**: string, input
  NAME is the name of the parameter to get a value from.
- **VALUE**: parameter—value, output
  The value of the parameter is returned in VALUE. The type of the value depends on the type of the parameter, which is either INTEGER, STRING, or REAL (single-precision). There is no equivalent to **x/zvpone** for double precision floating point; use **x/zvparmd** instead.
- **INSTANCE**: integer, input
  INSTANCE specifies which value you want. INSTANCE starts counting at 1, so the fifth value would have an INSTANCE of 5.
- **MAXLEN**: integer, input
  MAXLEN specifies the maximum length of the string buffer if the parameter is a string. It is used to avoid overflowing your buffer. If the parameter is not a string, or you don't care, set MAXLEN to 0. MAXLEN is rarely needed in FORTRAN, since string lengths are available from the strings themselves.
- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107).

5.2.14 x/zvpstat Information about a parameter

```
call xvpstat(name, count, def, maxlen, type)
status = zvpstat(name, count,def, maxlen, type);
```

This routine returns information about a parameter without returning its value. It is most useful to get the maximum length of any string and the number of strings in order to allocate a buffer before calling **x/zvparm**. This routine is also the only way to determine the data type of a parameter given only its name. The program should know the type in the PDF, but there are situations where this could be useful**.**

Arguments:

- **NAME**: string, input

NAME is the name of the parameter to get information about.

- **COUNT**: integer, output
  Returns the number of items in the parameter. If 0, parameter is either not found or is null.
- **DEF**: integer, output
  Returns 1 if the parameter was defaulted, and 0 otherwise.
  NOTE: The DEF flag is obsolete and should not be used.
- **MAXLEN**: integer, output
  Returns the maximum length of any value in the parameter. For REAL and INT, it is just the size of a REAL or INT. For STRING, it is the length of the longest string in the parameter. It does not include the null terminator, so you should add one before allocating a buffer in C.
- **TYPE**: string, output
  Returns the data type of the parameter. The possible values are “INT”, “REAL”, and “STRING”.

5.2.15 x/zvptst Indicate whether key word was specified

```
GIVEN = xvptst( key )
GIVEN = zvptst( key );
```

This function returns true if a key word was specified and false if not. For example, a program could check if the value “HALF” was given for a key word parameter with the code:

```
logical lhalf,xviptst

      lhalf = xviptst('HALF')
      if (lhalf) then
        .
        .
        .
```

The key word must be a value specified for a parameter of type KEYWORD. **Note:** Should be declared as a LOGICAL function in FORTRAN, as in the above example.

Arguments:

- **KEY**: input, string
  Name of key word to be tested for. Note this is not the parameter name, but one of the valid values for the key word parameter. Passed either by descriptor in FORTRAN (if CHARACTER string is passed) or reference in C (if char* array is passed).

5.3 Examples

When using the “vparm” routines to get parameter input from the user, you should always look at the count field, never the def field, to see if a value has been entered. But if there are no values, the return value is not necessarily touched... you're guaranteed "count" valid values but that might be 0.

To allow for all possible cases, use this test:

```
if (count == 0 || strlen(value) == 0)
```

This protects against null input. The user could enter a null value, in which case count is 1 but the value is empty.

6. Translation Routines

6.1 Introduction

The translation routines allow conversion of data between different data types and different host representations. See 1.3 Data Types and Host Representations (page 8), for more information on when and how to call these routines.

These routines all take a translation buffer as an argument. This is an opaque structure that is at least 12 integers (usually 48 bytes) long that will describe the translation. The internals of the buffer are unknown to the application; it is a private RTL data structure. One of five routines must be called first to set up this buffer. Then, **x/zvtrans** can be called as often as necessary to perform the translation. You may have several translations available at once by using different translation buffers.

The first integer in the buffer is special. If it is NULL (0) after the setup routine has been called, then no translation is needed. **x/zvtrans** will simply move the data in this case, but you can decide to forego calling **x/zvtrans** if it would be more efficient. This first integer is the *only* item an application may look at in the translation buffer. Using *any* other knowledge about the internals of the buffer may cause your program to break in the future, as the structure may change without notice.

6.2 Translation API

Below are the ten subroutines that comprise the Translation API.

6.2.1 x/zvhost—Integer and real data representations of a host given the host type name

```
call xvhost(host, intfmt, realfmt, status)
status = zvhost(host, intfmt, realfmt);
```

Returns the integer and real data representations of a host given the host type name. The returned values may be used with the translation routines or the INTFMT and REALFMT system label items. This routine can also return the host type name, for use with the HOST system label.

This routine is rarely needed, as the normal case is to read any type of file (where you get INTFMT and REALFMT from the file), and to write in native format (where you don't need to specify the formats). However, this routine is useful in the rare case that a program needs to write in a non-native format. The user can select the machine type for output, and this routine will return the data

representations needed for that machine type.

Arguments:

- **HOST**: string, input
  HOST is the type name of the host you want information about. It is not the name of a particular machine, rather, it is the name of a type of machine. HOST corresponds to the HOST system label. The HOST system label is intended for documentation only, however, the same values are valid in this parameter. Normally, the HOST parameter will come from user input. The valid values will change as VICAR is ported to more machines, so the program should allow any value as user input, then check the status from **x/zvhost** to see if the machine name is valid. The currently valid values are listed. See Table 1: Valid VICAR HOST Labels and Machine Types (page 10). In addition, the following values are accepted:
    - NATIVE : Returns the INTFMT and REALFMT for the machine currently running.
    - LOCAL : Same as NATIVE.
    - HOSTNAME : Special, see below.

The value “HOSTNAME” is special. If this value is given for HOST, then the return values change. The parameter INTFMT returns the host type name for the machine currently running. The parameter REALFMT is undefined on output. The value returned in INTFMT could then be used in another call to **x/zvhost** (which wouldn't gain much since “NATIVE” or “LOCAL” do the same thing), or it could be used in a HOST label. This should be needed only from FORTRAN. A C program should use the “ NATIVE _HOST_LABEL” macro defined in x/zvmaininc.h instead. Similarly, “NATIVE” and “LOCAL” will be mainly useful in a FORTRAN program, as a C program should use the NATIVE_INTFMT and NATIVE_REALFMT macros.

- **INTFMT**: string, output
  The integral host representation for the machine in question is returned in INTFMT. It corresponds to the INTFMT label item in a file, and the related parameters to the translation routines. The returned value will be one of the supported integer data types, which are listed. See Table 2: Valid VICAR Integer Formats (page 11).
  If the special host name “HOSTNAME” is given, then INTFMT instead returns the host type name (HOST label) of the native machine.

- **REALFMT**: string, output
  The floating-point host representation for the machine in question is returned in REALFMT. It corresponds to the REALFMT label item in a file, and the related parameters to the translation routines. The returned value will be one of the supported floating-point data types, which are listed. See Table 3: Valid VICAR Real Number Formats (page 11).
  If the special host name “HOSTNAME” is given, then the value returned in REALFMT is undefined and should not be used.

- **STATUS**: integer, output
  The returned status value. It is an argument in FORTRAN and the function return value in C. Any value other than SUCCESS indicates that the value given for HOST was invalid.

6.2.2 x/zvpixsize—Size of a pixel in bytes given the data type and host representation

```
status = xvpixsize(pixsize, type, ihost, rhost)
status = zvpixsize(pixsize, type, ihost, rhost);
```

Returns the size of a pixel in bytes given the data type and host representation. One of the **pixsize** routines should be used to figure out the size of a pixel. Do *not* assume any particular size, like 4 bytes for a REAL. It may be different on other machines. It is valid to use `sizeof()` in C to get the size of a pixel in the native representation *only*, but the **pixsize** routines are the only valid way to get the size of a pixel on any other hosts.

Arguments:

- **PIXSIZE**: integer, output
  Returns the size of a pixel in bytes. If an error occurs (such as an invalid data type), PIXSIZE is returned as 0.
- **TYPE**: string, input
  TYPE is the data type of the pixel. It corresponds to the FORMAT label item in a file. It may be one of the standard VICAR data types: “BYTE”, “HALF”, “FULL”, “REAL”,”DOUB”, or “COMP”. The types “WORD”, “LONG”, and “COMPLEX” are also accepted, but are obsolete and should not be used.
- **IHOST**: string, input
  IHOST is the integral host representation for the pixel. It corresponds to the INTFMT label item in a file. It may be any of the supported integer data types, which are listed. See Table 2: Valid VICAR Integer Formats (page 11). It may also be “NATIVE” or “LOCAL”, both of which mean the native host INTFMT. IHOST should be given even if you are dealing with floating-point data types.
- **RHOST**: string, input
  RHOST is the floating-point host representation for the pixel. It corresponds to the REALFMT label item in a file. It may be any of the supported floating-point data types, which are listed. See Table 3: Valid VICAR Real Number Formats (page 11). It may also be “NATIVE” or “LOCAL”, both of which mean the native host REALFMT. RHOST should be given even if you are dealing with integral datatypes.

6.2.3 x/zvpixsizeb—Size of a binary label value in bytes from a file

```
status = xvpixsizeb(pixsize, type, unit)
status = zvpixsizeb(pixsize, type, unit);
```

Return the size of a binary label value in bytes from a file. This routine is exactly like **x/zvpixsize** except that the IHOST and RHOST values are obtained for binary labels from the file specified by UNIT, which must be open. It is provided merely as a shortcut to get the size of a binary label value for a file.

Arguments:

- **PIXSIZE**: integer, output
  Returns the size of a pixel in bytes. If an error occurs (such as an invalid data type), PIXSIZE is returned as 0.
- **TYPE**: string, input

TYPE is the data type of the binary label value. It corresponds to the FORMAT label item in a file, although binary label values are not restricted to FORMAT and maybe any data type. TYPE may be one of the standard VICAR data types: “BYTE”, “HALF”,”FULL”, “REAL”, “DOUB”, or “COMP”. The types “WORD”, “LONG”, and “COMPLEX” are also accepted, but are obsolete and should not be used.

- **UNIT**: integer, input
  UNIT is the unit number of an open file, which is used to obtain the source BINTFMT and INTFMT. The values obtained from the file are used exactly like the **x/zvpixsize** IHOST and RHOST.

6.2.4 x/zvpixsizeu—Size of a pixel in bytes from a file

```
status = xvpixsizeu(pixsize, type, unit)
status = zvpixsizeu(pixsize, type,unit);
```

Return the size of a pixel in bytes from a file. This routine is exactly like **x/zvpixsize** except that the IHOST and RHOST values are obtained from the file specified by UNIT, which must be open. It is provided merely as a shortcut to get the pixel size of a file.

Arguments:

- **PIXSIZE**: integer, output
  Returns the size of a pixel in bytes. If an error occurs (such as an invalid data type), PIXSIZE is returned as 0.
- **TYPE**: string, input
  TYPE is the data type of the pixel. It corresponds to the FORMAT label item in a file. It may be one of the standard VICAR data types: “BYTE”, “HALF”, “FULL”, “REAL”,”DOUB”, or “COMP”. The types “WORD”, “LONG”, and “COMPLEX” are also accepted, but are obsolete and should not be used.
- **UNIT**: integer, input
  UNIT is the unit number of an open file, which is used to obtain the source INTFMT and REALFMT. The values obtained from the file are used exactly like the **x/zvpixsize** IHOST and RHOST.

6.2.5 x/zvtrans—Translate pixels from one format to another

```
call xvtrans(buf, source, dest, npix)
zvtrans(buf, source, dest, npix);
```

Translate pixels from one format to another. One of the translation setup routines must have been called first to set up the translation buffer. This routine is the only standard way to translate data in the VICAR system, both between host representations (e. g. VAX to IEEE) and between data types (e.g. integer to real).

This routine is coded to be very efficient, so it may be called inside a tight loop with very little performance penalty**.**

Arguments:

- **BUF**: integer-array(12), input
  BUF is the translation buffer that describes the translation to be performed. It is initialized by one of the translation setup routines. The internals of this buffer are unknown to the application program,

with one exception, described below. *Any* other access to the internals of the buffer may cause your program to break in the future, as the structure may change without notice.
If the first integer in the translation buffer is NULL (0), then **x/zvtrans** merely moves the data from the source to the destination, without any conversion. This can happen often when reading a file, where you don't know ahead of time what host representation the input data is in. If it turns out to be the native representation, no translation is necessary, and the first integer of the buffer will be 0. This fact can sometimes be used to avoid copying the data, making the program more efficient.

- **SOURCE**: pixel-buffer, input
  SOURCE is the source data buffer.
- **DEST**: pixel-buffer, output
  DEST is the destination data buffer. It may *not* be the same as SOURCE, i.e. you can't translate in place or with overlapping buffers.
- **NPIX**: integer, input
  NPIX is the number of pixels (not bytes!) to translate. Both SOURCE and DEST must be large enough to hold NPIX pixels; no checking is done.

6.2.6 x/zvtrans_in—Create translation buffer for input

```
call xvtrans_in(buf, stype, dtype, sihost, srhost, status)
status = zvtrans_in(buf, stype, dtype, sihost, srhost);
```

Create translation buffer for input. The data will be converted from a host representation of (SIHOST, SRHOST) and data type of STYPE into the machine's native representation and data type DTYPE. So, it converts from foreign to local format. Since all processing must be done in native format on the machine the program is running on, this translation is most often needed for input from a file**.**

Arguments:

- **BUF**: integer-array(12), output
  BUF is the translation buffer that this routine will setup, describing the translation to be performed.
- **STYPE**: string, input
  STYPE is the source data type. It corresponds to the FORMAT label item in a file. It may be one of the standard VICAR data types: “BYTE”, “HALF”, “FULL”, “REAL”,”DOUB”, or “COMP”. The types “WORD”, “LONG”, and “COMPLEX” are also accepted, but are obsolete and should not be used.
- **DTYPE**: string, input
  DTYPE is the desired destination data type. It corresponds to the FORMAT label item in a file. It may be one of the standard VICAR data types: “BYTE”, “HALF”, “FULL”, “REAL”, “DOUB”, or “COMP”. The types “WORD”, “LONG”, and “COMPLEX” are also accepted, but are obsolete and should not be used.
- **SIHOST**: string, input
  SIHOST is the host representation for the source of integral data types. It corresponds to the INTFMT label item in a file. It may be any of the supported integer data types, which are listed. See Table 2: Valid VICAR Integer Formats (page 11). It may also be “ NATIVE” or “LOCAL”, both of which mean the native host INTFMT. SIHOST should be given even if you are dealing only with floating-point data types. See also **x/zvhost**.

- **SRHOST**: string, input
  SRHOST is the host representation for the source of floating-point data types. It corresponds to the REALFMT label item in a file. It may be any of the supported floating-point data types, which are listed. See Table 3: Valid VICAR Real Number Formats (page 11). It may also be "NATIVE" or "LOCAL", both of which mean the native host REALFMT. SRHOST should be given even if you are dealing only with integral data types. See also **x/zvhost**.

- **STATUS**: integer, output
  The returned status value. It is an argument in FORTRAN and the function return value in C. Any value other than SUCCESS indicates that the translation is invalid for some reason, and the translation buffer should not be used.

6.2.7 x/zvtrans_inb—Create translation buffer for input from binary labels of a file

```
call xvtrans_inb(buf, stype, dtype, unit, status)
status = zvtrans_inb(buf,stype, dtype, unit);
```

Create translation buffer for input from the binary labels of a file. This routine is exactly like **x/zvtrans_in** except that the SIHOST and SRHOST values are obtained for binary labels from the file specified by UNIT, which must be open. It is provided merely as a shortcut for the common case of reading binary label data from a labeled file**.**

Arguments:

- **BUF**: integer-array(12), output
  BUF is the translation buffer that this routine will setup, describing the translation to be performed.

- **STYPE**: string, input
  STYPE is the source data type. It corresponds to the FORMAT label item in a file, although binary label values are not restricted to FORMAT and may be of any datatype. STYPE may be one of the standard VICAR data types: "BYTE", "HALF", "FULL", "REAL","DOUB", or "COMP". The types "WORD", "LONG", and "COMPLEX" are also accepted, but are obsolete and should not be used.

- **DTYPE**: string, input
  DTYPE is the desired destination data type. It corresponds to the FORMAT label item in a file, although binary label values are not restricted to FORMAT and may be of any data type. DTYPE may be one of the standard VICAR data types: "BYTE", "HALF", "FULL", "REAL", "DOUB", or "COMP". The types "WORD", "LONG", and "COMPLEX" are also accepted, but are obsolete and should not be used.

- **UNIT**: integer, input
  UNIT is the unit number of an open file, which is used to obtain the source BINTFMT and INTFMT. The values obtained from the file are used exactly like the **x/zvtrans_in** SIHOST and SRHOST.

- **STATUS**: integer, output
  The returned status value. It is an argument in FORTRAN and the function return value in C. Any value other than SUCCESS indicates that the translation is invalid for some reason, and the translation buffer should not be used.

6.2.8 x/zvtrans_inu—Create translation buffer for input from a file

```
call xvtrans_inu(buf, stype, dtype, unit, status)
status = zvtrans_in u(buf,stype, dtype, unit);
```

Create translation buffer for input from a file. This routine is exactly like **x/zvtrans_in** except that the SIHOST and SRHOST values are obtained from the file specified by UNIT, which must be open. It is provided as a shortcut for the common case of reading image data from a labeled file**.**

Arguments:

- **BUF**: integer-array(12), output
  BUF is the translation buffer that this routine will setup, describing the translation to be performed.
- **STYPE**: string, input
  STYPE is the source data type. It corresponds to the FORMAT label item in a file. It may be one of the standard VICAR data types: " BYTE", "HALF", "FULL", "REAL","DOUB", or "COMP". The types "WORD", "LONG", and "COMPLEX" are also accepted, but are obsolete and should not be used.
- **DTYPE**: string, input
  DTYPE is the desired destination data type. It corresponds to the FORMAT label item in a file. It may be one of the standard VICAR data types: "BYTE", "HALF", "FULL", "REAL", "DOUB", or "COMP". The types "WORD", "LONG", and "COMPLEX" are also accepted, but are obsolete and should not be used.
- **UNIT**: integer, input
  UNIT is the unit number of an open file, which is used to obtain the source INTFMT and REALFMT. The values obtained from the file are used exactly like the **x/zvtrans_in** SIHOST and SRHOST.
- **STATUS**: integer, output
  The returned status value. It is an argument in FORTRAN and the function return value in C. Any value other than SUCCESS indicates that the translation is invalid for some reason, and the translation buffer should not be used.

6.2.9 x/zvtrans_out—Create translation buffer for output

```
call xvtrans_out(buf, stype, dtype, dihost, drhost, status)
status =zvtrans_out(buf, stype, dtype, dihost, drhost);
```

Create translation buffer for output. The data will be converted from the machine's native representation and data type of STYPE into a host representation of (DIHOST, DRHOST) and data type of DTYPE. So, it converts from local to foreign format. Since all processing must be done in native format on the machine the program is running on, this translation is most often needed for output to a file.

This routine is less commonly used than the input routines. The general rule for applications is to read any format, but write the native format. Translation on output is not needed in this case. However, **x/zvtrans_out** is provided for special cases where the data must be written in a different host representation**.**

Arguments:

- **BUF**: integer-array(12), output

BUF is the translation buffer that this routine will create, describing the translation to be performed.

- **STYPE**: string, input
  STYPE is the source data type. It corresponds to the FORMAT label item in a file. It may be one of the standard VICAR data types: “BYTE”, “HALF”, “FULL”, “REAL”,”DOUB”, or “COMP”. The types “WORD”, “LONG”, and “COMPLEX” are also accepted, but are obsolete and should not be used.
- **DTYPE**: string, input
  DTYPE is the desired destination data type. It corresponds to the FORMAT label item in a file. It may be one of the standard VICAR data types: “BYTE”, “HALF”, “FULL”, “REAL”, “DOUB”, or “COMP”. The types “WORD”, “LONG”, and “COMPLEX” are also accepted, but are obsolete and should not be used.
- **DIHOST**: string, input
  DIHOST is the host representation for the destination of integral data types. It corresponds to the INTFMT label item in a file. It may be any of the supported integer data types, which are listed. See Table 2: Valid VICAR Integer Formats (page 11). It may also be “NATIVE” or “LOCAL”, both of which mean the native host INTFMT. DIHOST should be given even if you are dealing only with floating-point data types. See also **x/zvhost**.
- **DRHOST**: string, input
  DRHOST is the host representation for the destination of floating-point data types. It corresponds to the REALFMT label item in a file. It may be any of the supported floating-point data types, which are listed. See Table 3: Valid VICAR Real Number Formats (page 11). It may also be “NATIVE” or “LOCAL”, both of which mean the native host REALFMT. DRHOST should be given even if you are dealing only with integral data types. See also **x/zvhost**.
- **STATUS**: integer, output
  The returned status value. An argument in FORTRAN or the function return value in C. Any value other than SUCCESS indicates that the translation is invalid, and the translation buffer should not be used.

6.2.10 x/zvtrans_set—Create translation buffer for data types only

```
call xvtrans_set(buf, stype, dtype, status)
status = zvtrans_set(buf, stype, dtype);
```

Create translation buffer for data types only. Both the source and the destination must be in the native host representation. It is useful for converting internal buffers from one data type to another. Don't use it with data direct from a file, however, as files are not guaranteed to be in the native host representation**.**

Arguments:

- **BUF**: integer-array(12), output
  BUF is the translation buffer that this routine will setup, describing the translation to be performed.
- **STYPE**: string, input
  STYPE is the source data type. It corresponds to the FORMAT label item in a file. It may be one of the standard VICAR data types: “BYTE”, “HALF”, “FULL”, “REAL”,”DOUB”, or “COMP”. The types “WORD”, “LONG”, and “COMPLEX” are also accepted, but are obsolete and should not be used.

- **DTYPE**: string, input
  DTYPE is the desired destination data type. It corresponds to the FORMAT label item in a file. It may be one of the standard VICAR data types: “BYTE”, “HALF”, “FULL”, “REAL”, “DOUB”, or “COMP”. The types “WORD”, “LONG”, and “COMPLEX” are also accepted, but are obsolete and should not be used.
- **STATUS**: integer, output
  The returned status value. It is an argument in FORTRAN and the function return value in C. Any value other than SUCCESS indicates that the translation is invalid for some reason, and the translation buffer should not be used.

7. FORTRAN String Conversion Routines

7.1 Introduction

These routines allow C-language subroutines to use character strings passed in from a FORTRAN routine (CHARACTER*n data type). FORTRAN-callable subroutines written in C must use these routines to handle character strings. The passing of strings between FORTRAN and C varies widely among different machine architectures. Attempting to do your own without using these routines practically guarantees that your code will not be portable.

These routines are most useful in the FORTRAN interface to SUBLIB routines, but they could be useful within a single application program if it uses both languages.

All of these routines are callable from C only. Do not} attempt to call them from FORTRAN. Writing a FORTRAN routine that accepts C strings is much more difficult. See 2.6 Mixing FORTRAN and C (page 27).

These routines have much in common in their calling sequences, so the common features and rules are described only once in the Common Features section below.

Two of the routines apply only to strings being sent out of a C routine, back to the FORTRAN caller. They are marked “output” below. The other four apply only to strings being passed in to a C routine, from a FORTRAN caller. They are marked “input".

7.1.1 Common Features: Rules and arguments common to all string routines

All the string conversion routines place great emphasis on the argument list of the C routine that is directly called by FORTRAN. Certain rules apply to that argument list, and many of the common parameters reference it. All of these apply to the routine called directly} by FORTRAN. Suppose a FORTRAN routine calls routine a(), which then calls routine b(), and b() wants to call one of these string conversion routines. All of the parameters and rules will apply only to a()'s argument list. Routine b() will need much of the information passed in from a() in order to call the string conversion routines, but all the information is relative to a()'s argument list. In general, it's usually easier to have a() do all the conversion and let b() deal only with C strings, but sometimes that is impractical.

7.1.1.1 Include file

In order to use any of the routines, you must include the file “ftnbridge.h”. As with other RTL includes, you must include xvmaininc.h first, the names must be enclosed in double quotes and end in the “.h” extension (you don't need xvmaininc.h if “vicmain_c” is included). Example:

```
#include "xvmaininc.h"
#include "ftnbridge.h"
```

7.1.1.2 Imakefile

The flag FTN_STRING must be defined in the imakefile for the program unit if any C routine accepts FORTRAN strings. This applies to both the direct-called routine, and the routine that ultimately calls one of the conversion routines. The FTN_STRING flag causes the compiler to use a lower level of optimization, which is required on some machines in order to access the argument list.

7.1.1.3 FORSTR_PARAM and FORSTR_DEF macros

In most routines (see below for exception), include the macro FORSTR_PARAM in the argument list of the directly called routine, FORSTR_DEF at the end of the formal parameter declaration list, just before the opening brace of the procedure and FORSTR_BLOCK immediately after the opening

brace of the function with no other declarations or statements before it.

FORSTR_DEF and FORSTR_BLOCK should not have semicolons after them, as the semicolon (if needed) is included in the macro definition. These macros are no-ops on many machines, but are required on some in order to get at the FORTRAN string lengths.

The exception is routines that use the <varargs.h> variable argument mechanism. User subroutines should not normally use varargs, but it is used fairly extensively inside the RTL to handle the keyword-value argument pairs. If you use <varargs.h>, follow all the standard C rules for that mechanism, and you should not use the FORSTR_PARAM, FORSTR_BLOCK or FORSTR_DEF macros. For example:

```
FTN_NAME(text)(inbuf, inchr, line, outbuf, size, dn, FORSTR_PARAM)
char *inbuf;
int *inchr, *line;
unsigned char outbuf[];
int *size, *dn;
FORSTR_DEF
{
   FORSTR_BLOCK
   char *c_string;
   int  length,i;
   ...
```

7.1.1.4 Argument restrictions

All arguments to the direct-called routine must be the size of a generic pointer. Since FORTRAN passes everything by reference anyway, all of your arguments will be pointers, so this should not cause a problem.

7.1.1.5 Arguments to FORTRAN string conversion routines

The arguments to the FORTRAN string conversion routines all apply to the argument list of the routine that is directly called by FORTRAN, not to any intermediary routines.

- **FOR_STRING: pointer to char, input/output**
  FOR_STRING should be the name of the argument that contains the FORTRAN string. It should be declared as type “char *” in the argument list. Simply pass in the name of the argument, without any extra \&'s or *'s or anything. The FORTRAN routine must declare the string as CHARACTER*n. String arrays (handled with the routines **sc2for_array**} and **sfor2c**_array} must be declared as single-dimension arrays of CHARACTER*n, but are still declared as “char *” in the C routine.

- **ARGPTR: generic pointer, input**
  ARGPTR should be the address of the first argument, whatever it is. It does not matter what the data type of the first argument is, or if it is already a pointer to something else. Simply put an ampersand (\&) followed by the name of the first argument in the argument list.

- **NARGS: integer, input**
  NARGS is simply the total number of arguments passed in to the routine. Do not} count FORSTR_PARAM as one of the arguments. If you are using <varargs.h>, pass in the actual number of arguments sent to the routine.

- **ARGNO: integer, input**
  ARGNO is the number of the argument that contains the string to be converted. If the string is the first argument in the list, ARGNO would be 1. If it is the fifth argument, ARGNO would be 5.

- **STRNO: integer, input**
  STRNO is the string count for the string to be converted. It is similar to ARGNO, but counts only strings. So, if the string you are converting is the third argument, but only the second string in the argument list (because the first one is an integer), then ARGNO would be 3 but STRNO would be 2.

Some examples may help to clarify things. The routine **sfor2c** is used in these examples, but the principles apply to all the string conversion routines. The calling sequence for **sfor2c**} is **sfor2c**(c_string, max_length, for_string, argptr, nargs, argno, strno).

```
int constargs(a, s1, s2, b, FORSTR_PARAM)
  int *a, *b;
  char *s1, *s2;
  FORSTR_DEF
{ char cs1[11], cs2[20];
  sfor2c(cs1, 10, s1, &a, 4, 2, 1);
  sfor2c(cs2, 19, s2, &a, 4, 3, 2);
}

int constargs2(s1, s2, a, s3, b, s4, FORSTR_PARAM)
  char *s1, *s2, *s3, *s4;
  int *a, *b;
{ char cs1[101], cs2[31], cs3[80], cs4[5];
  sfor2c(cs1, 100, s1, &s1, 6, 1, 1);
  sfor2c(cs2, 30,  s2, &s1, 6, 2, 2);
  sfor2c(cs3, 79,  s3, &s1, 6, 4, 3);
  sfor2c(cs4, 4,   s4, &s1, 6, 6, 4);
}

int varargs(va_alist)
  va_dcl
{ char cs[11];
 /* nargs = number of arguments, argno=argument #, and strno=string #
*/
 /* which all come from knowing what to expect in the argument list */
  forstr = va_arg(ap, char *);
  sfor2c(cs, 10, forstr, &va_alist, nargs, argno, strno);
}
```

7.2 String Conversion API

7.2.1 sc2for—C null-terminated string to an output FORTRAN string

```
sc2for(c_string, max_length, for_string, argptr, nargs, argno, strno);
```

This routine converts a standard C null-terminated string to an output FORTRAN string. It is used to send strings back to a FORTRAN caller.

Arguments:

- **C_STRING**: string, input
  String to convert in standard null-terminated C format.
- **MAX_LENGTH**: integer, input.
  Alternate maximum length of the FORTRAN string. Normally the maximum length is obtained from the output FORTRAN string itself (the “n” in the CHARACTER*n declaration. If MAX_LENGTH

is passed in as 0, then this natural length is used. MAX_LENGTH is an alternate maximum string length in case one is provided as a parameter to the routine. The actual maximum FORTRAN length used is the minimum of the passed in MAX_LENGTH (if not 0) and the natural FORTRAN string length. The output string will be truncated if the FORTRAN string is not long enough. Any extra space at the end of the FORTRAN string will be padded with blanks in the standard FORTRAN style.

- **FOR_STRING, ARGPTR, NARGS, ARGNO, STRNO**
  See 7.1.1 Common Features: Rules and arguments common to all string routines (page 102) above.

7.2.2 sc2for_array—C null-terminated array of strings to FORTRAN string array

```
sc2for_array(c_string,len,nelements,for_string,max_length,argptr,nargs,
argno,strno);
```

This routine converts a standard C null-terminated array of strings into an output FORTRAN string array. The C string array must be a two-dimensional array of characters, not} an array of pointers to strings. The FORTRAN string should be declared as a single-dimensional array of CHARACTER*n in the calling routine.

Arguments:

- **C_STRING**: string array, input
  String array to convert in standard null-terminated C format. It must be a two-dimensional array of char, not an array of pointers to strings. Each string should have its own null terminator.
- **LEN**: integer, input
  LEN is the size of the inner dimension of the C string array. If the array is declared as “char x[10][81];+” (10 strings of 80 characters each plus terminator), then LEN would be 81.
- **NELEMENTS**: integer, input
  NELEMENTS is the number of strings in the array to convert.
- **MAX_LENGTH**: integer, input/output
  On input, MAX_LENGTH is the alternate maximum length of each FORTRAN string. Normally the maximum length is obtained from the output FORTRAN string array itself (the “n” in a CHARACTER*n declaration). If MAX_LENGTH is passed in as 0, then this natural length is used. MAX_LENGTH is an alternate maximum string length in case one is provided as a parameter to the routine. The actual maximum FORTRAN length used is the minimum of the passed in MAX_LENGTH (if not 0) and the natural FORTRAN string length. The output string will be truncated if the FORTRAN string is not long enough. Any extra space at the end of the FORTRAN string will be padded with blanks in the standard FORTRAN style. MAX_LENGTH should normally be passed in as 0, as it makes little sense to override the natural FORTRAN string length. However, it is possible, and might be useful in some unusual cases.
  On output, MAX_LENGTH returns the actual FORTRAN string length used by the routine.
- **FOR_STRING, ARGPTR, NARGS, ARGNO, STRNO**
  See 7.1.1 Common Features: Rules and arguments common to all string routines (page 102) above.

7.2.3 sfor2c—FORTRAN input string to a standard C null-terminated string

```
sfor2c(c_string, len, for_string, argptr, nargs, argno, strno);
```

This routine converts FORTRAN input string to a standard C null-terminated string. It is used to receive string parameters from a FORTRAN caller.

Arguments:

- **C_STRING**: string, output

Buffer to hold the output C string. The string will be truncated if the buffer is not big enough. It will always be null terminated, even if it was truncated. Any trailing blanks in the FORTRAN string will be removed.

- **LEN**: integer, input
  Maximum length of the output C string. This parameter defines the size of the C string buffer. It is expressed in terms of the maximum length} of the string, which means it does not include the terminator byte. The buffer should actually be declared to be one byte larger than LEN to allow room for the null terminator. So, if the declaration is "char buffer[80];", then LEN should be 79.
- **FOR_STRING, ARGPTR, NARGS, ARGNO, STRNO**
  See 7.1.1 Common Features: Rules and arguments common to all string routines (page 102) above.

7.2.4 sfor2c_array—FORTRAN string array to C null-terminated array of strings

```
sfor2c_array(c_string, max_length, nelements, for_string, argptr,
nargs, argno, strno);
```

This routine converts a FORTRAN string array to a standard C null-terminated array of strings. The returned C string array is a two-dimensional array of characters, not} an array of pointers to strings. The FORTRAN string should be declared as a single-dimensional array of CHARACTER*n in the calling routine.

This routine is somewhat unusual in that it actually allocates the memory for the C string for you. You pass in the address of a character pointer, not the address of a buffer for the characters. **sfor2c**_array calls malloc()}to allocate the required memory, and returns the address of that memory in the pointer. It is your responsibility to call free()to free up that memory when you are done with it.

Arguments:

- C_STRING: pointer to string array, output
  C_STRING is the address of a pointer that will be filled in to point at the string array. The returned array will be a two-dimensional array of characters, not an array of pointers to strings. Each string will have its own null terminator.

This routine actually allocates the memory for the C string for you. C_STRING is the address of a character pointer, not the address of a buffer for the characters. **sfor2c**_array} calls sub malloc() to allocate the required memory, and returns the address of that memory in C_STRING. It is your responsibility to call free() to free up that memory when you are done with it.

The inner dimension of the array is returned via the MAX_LENGTH parameter. Since you don't know this size at compile time, you can't access the strings like a normal two-dimensional array. It is easy enough to do your own addressing, however. For example:

```
char *array;
int maxlen=0;
...
sfor2c_array(&array, &maxlen, ...);
...
process_string(array+(i*maxlen));       /* to get at the i'th string */
```

- MAX_LENGTH: integer, input/output
  On input, MAX_LENGTH is the alternate maximum length of each FORTRAN string. Normally the maximum length is obtained from the input FORTRAN string array itself (the "n" in a CHARACTER*n declaration). If MAX_LENGTH is passed in as 0, then this natural length is used. MAX_LENGTH is an alternate maximum string length in case one is provided as a parameter to the routine. The actual maximum FORTRAN length used is the minimum of the passed in MAX_LENGTH (if not 0) and the natural FORTRAN string length. MAX_LENGTH should almost always be passed in as 0, as it makes little sense to override the natural FORTRAN string length, especially on an array. However, it is possible, and might be useful in some unusual cases.

On output, MAX_LENGTH returns the size of the inner dimension of the C string array that was allocated by **sfor2c**_array}. To access the i'th string in the array, simply add i*maxlen to the returned array pointer.

- NELEMENTS: integer, input
  The number of strings in the array to convert.
- FOR_STRING, ARGPTR, NARGS, ARGNO, STRNO
  See 7.1.1 Common Features: Rules and arguments common to all string routines (page 102) above.

7.2.5 sfor2len—Length of a FORTRAN string

```
length = sfor2len(for_string, argptr, nargs, argno, strno);
```

This routine returns the length of a FORTRAN string. It does not get a pointer to the characters, nor does it convert them to a C string. It is most useful to get the length of a string in order to allocate a buffer for it before calling **sfor2c**. Note that the length returned is the "n" in the CHARACTER*n declaration, not the number of characters currently in the string.

Arguments:

- FOR_STRING, ARGPTR, NARGS, ARGNO, STRNO
  See 7.1.1 Common Features: Rules and arguments common to all string routines (page 102) above.

7.2.6 sfor2ptr—Pointer to actual characters in FORTRAN string

```
ptr = sfor2ptr(for_string);
```

This routine returns a pointer to the actual characters in an input FORTRAN string. It does not get the FORTRAN string length, nor does it copy the string to an output C string. It merely returns a pointer to the characters. No guarantee is made that any of the characters are valid, since that depends on the FORTRAN string length. You can be sure that there will not be a null terminator. Some machines may have one, but you may not depend on a null terminator being there.

This routine should be used sparingly; use **sfor2c** for most FORTRAN string conversion. **sfor2ptr** is mainly intended for use in scanning a variable-length argument list to find the end-of-list marker. It is used extensively inside the RTL for this purpose. It should only rarely if ever be used in application code.

Note that only the FOR_STRING standard argument is required. This is because the FORTRAN string length is ignored. All the other parameters are used to find the length.

Arguments:

- **FOR_STRING**
  See 7.1.1 Common Features: Rules and arguments common to all string routines (page 102) above.

8. Utility Routines

8.1 Introduction

This Section describes the miscellaneous utility subroutines and functions available in the VICAR run-time library. The subroutines described in this Section may not conform to all the software standards established in the rest of the manual. In particular, some routines require either FORTRAN or C formatted strings. Those requirements are indicated on a routine by routine basis.

The routines **x/zvfilpos**, **x/zvtpinfo**, **x/zvtpmode**, and **x/zvtpset** are used to directly manipulate 9-track tapes from within a program. The use of tapes in this way directly from a program is problematic under UNIX, so it is discouraged. While this method of access is allowed, the underlying implementation is not portable. It currently only runs on Sun-4s with one brand of tape drive. All processing should be done from disk files, with only operating system or special-purpose utilities

accessing the tape (these utilities would only transfer files to/from tape, with no processing).

8.2 Utility API

8.2.1 abend/zabend—Terminate processing abnormally

```
call abend()
zabend()
```

abend issues a message indicating that **abend** was called, and then calls **x/zvend** with a status of zero to indicate a program failure. Any open VICAR files are closed and write buffers flushed before termination.

8.2.2 x/zmove—Move bytes from one buffer to another

```
call xmove(from, to, len)
zmove(from, to, len);
```

Move bytes from one buffer to another. Overlapping moves are handled correctly, unlike the C routine `memcpy()`.

Arguments:

- **FROM**: pixel-array, input
  FROM is the buffer to move the bytes from (the source). It may *not* be a FORTRAN CHARACTER*n variable.
- **TO**: pixel-array, input
  TO is the buffer to move the bytes to (the destination). It may *not* be a FORTRAN CHARACTER*n variable.
- **LEN**: integer, input
  The number of bytes to move. There is no restriction on how many bytes can be moved, except available memory. LEN is measured in *bytes*, not *pixels*.

8.2.3 x/zvbands—Return band usage information

```
call xvbands(sb, nb, nbi)
zvbands(sb, nb, nbi);
```

x/zvbands is analogous to **x/zvsize** in its usage. It returns information from the command line given in either the BANDS or the SB, NB parameters, or, if the parameters were not specified, from the input file size.

Arguments:

- **SB**: output, integer
  The starting band for processing. If the BANDS parameter (BANDS = (SB, NB))was given, the first element of BANDS is used. If BANDS was not given, and the SB parameter was given, SB is used.

Otherwise, the default is 1.

- **NB**: output, integer
  The number of bands for processing. If the BANDS parameter was given, the second element of BANDS is used. If BANDS was not given, and the NB parameter was given, NB is used. Otherwise, the default value is the number of bands in the primary input.

- **NBI**: output, integer
  The number of bands in the primary input file.

8.2.4 x/zvcmdout—Sends a command string to TAE to be executed

```
call xvcmdout(command, status)
status = zvcmdout(command);
```

Sends a command string to TAE to be executed, and returns output values in the interactive parblock, accessible by the **x/zviparm** family of routines. The command must be a TAE intrinsic command, or a procedure PDF that uses only intrinsic commands, i.e. no processes, no DCL or shell. This is mainly intended for running VIDS procedures from within a program, but it may have other applications as well.

This routine is quite similar to **x/zvcommand**. The only difference is that **x/zvcmdout** makes any output variables accessible. This is most useful with the VIDS JGET command, which returns values to the caller in TAE variables. These variables are placed in the interactive parblock.

Arguments:

- **COMMAND**: string, input
  The command string to execute. It may be a TAE intrinsic command or a procedure PDF which uses intrinsic commands only (this includes almost all VIDS PDF's). All potential outputs from the executed command will be in the form of NAME parameters. The values you wish returned should be declared as local variables in the PDF for the program (the one you're writing, not the one being called). These local variables should then be passed in the COMMAND string as values for the NAME parameters in the executed command. The variables can then be accessed from the interactive parblock via the **x/zviparm** family of routines.

- **STATUS**: integer, output
  The returned status value. It is an argument in FORTRAN and the function return value in C. Any value other than SUCCESS indicates that the command didn't execute or the outputs didn't get returned correctly, and the returned values should not be used.

8.2.5 x/zvcommand—Execute a VICAR command string

```
call xvcommand(command,status)
status =zvcommand(command,status)
```

x/zvcommand passes a command string on to the VICAR supervisor (TAE) for execution.

The command may be a VICAR intrinsic command or a procedure PDF that uses only VICAR intrinsic commands (or other procedure PDF's). Process PDF's and DCL commands are not allowed. Examples of commands that are allowed are VIDS commands and tape mounting commands (but NOT tape initialization since it uses DCL commands).

Arguments:

- **COMMAND**: input, string
  The command string is passed in exactly as the user would type it on the command line. It may be either a FORTRAN or a C string.

- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error code may be XVCOMMAND_ERROR (meaning an internal error with interprocess communications) or it will be a success/failure indicator (either it's equal to SUCCESS or it's not). The actual error code from

the command may be returned for some types of commands, but all you can count on is a success/fail indication.
If an error occurs, the action specified by **x/zveaction** is performed. Even if 'S' is not specified in the **x/zveaction** call (which tells VICAR not to print system messages), the executed program will still print error messages from the executed command. The only message that 'S' controls is the message from VICAR saying that **x/zvcommand** failed.

8.2.6 x/zvfilename—Returns a filename suitable for use with a system open() call

```
call xvfilename(in_name, out_name, status)
status = zvfilename(in_name,out_name, out_len);
```

Given a filename as input by the user, this function returns a filename suitable for use with a system `open()` call or other non-VICAR file operation. For UNIX, this means that environment variables and `~username` are expanded. For VMS, this means the old-style temporary filename suffix. Z *xx* is added. For both systems, the "`+` " form of temporary file is expanded.

This function does not need to be called if the RTL is used for I/O (it is called internally by **x/zvopen**). But, any program that gets a filename from the user for use in non-RTL I/O should make use of this function.

Because this function is only intended for use with non-VICAR I/O, only diskfile names are supported. Names specifying tape files or memory files will be treated as if they were disk files, which could provide surprising results. "`*`" as a wildcard character is legal in this function, and is passed through unchanged (so the output has a "`*`" in it).

For UNIX, the expansions are as follows:

- **`$var`**:Expand environment variable "`var`".
- **`${var}`**:Expand environment variable "`var`".
- **`~user`**: Expand to home directory of user"`user`".
- **`~`**: Expand to home directory of current user (`$HOME`).
- **`$$`**: Insert a single `$` (no environment variable).
- **`+`**: Expand to translation of `"$VTMP/`" for temporary files.

For VMS, the expansions are as follows:

- **`+`**: Expand to "`vtmp:`" (if subdirectory present) or "`vtmp:[000000]`"(if no subdirectory) for temporary files.
- **no + and no suffix**: Append ". Z *xx*" (where *xx* comes from `v2$pidcode`) for old-style temporary names.

Under VMS, both expansions may occur on the same name.

The temporary filename locations (`$VTMP` and `vtmp`) are set up in vicset2.

Arguments:

- **IN_NAME:** string, input
  IN_NAME is the input filename that you want converted.
- **OUT_NAME:** string, output
  The resultant converted string is returned in OUT_NAME.
- **OUT_LEN:** integer, input
  This argument specifies the length of the output string buffer OUT_NAME (to avoid overflow). A length of 0 means the buffer is unlimited, and it is the caller's responsibility to make sure there is no

overflow. OUT_LEN is only present in the C interface.

- **STATUS**: integer, output
  The returned status value. It is an argument in FORTRAN and the function return value in C. Any value other than SUCCESS indicates that the filename did not translate properly, and the returned value should not be used.

8.2.7 x/zvfilpos—Return the current tape position

```
POSITION = x/zvfilpos(UNIT)
```

x/zvfilpos looks up the current file position of the given unit if it is a tape, and returns it in POSITION. If the file is not on tape, -1 is returned.

This routine assumes that file UNIT has already been opened with **x/zvopen**.

Arguments:

- **UNIT**: input, integer
  The unit number of the tape from **x/zvunit**. UNIT must have already been opened with a call to **x/zvopen**.

8.2.8 x/zvmessage—Log a user message

```
call xvmessage(message, key)
zvmessage(message, key)
```

x/zvmessage is the primary subroutine to be used to print messages on the user terminal or in log files. It will work whether or not the program is under the VICAR supervisor, and allows the specification of a "key" for help on the message.

Arguments:

- **MESSAGE**: input, string
  The message to be logged. May be either a C or FORTRAN string.

- **KEY**: input, string
  KEY is an optional argument used to give a *message key*. A message key is used to look up help information for an error message. It consists of a string of the form "*facility-key*", where *facility* is the same for all error messages described in one package, and *key* is a unique identifier for that error messages.
  For example, a program named COPY could have a series of error messages, all of which would begin with the facility name "COPY", using keys such as "COPY-NO INPUT" or "COPY-END OFFILE". The error messages would then be stored in a file called COPYFAC. MSG in the standard TAE error message file format. The user could then use HELP-MESSAGE or the "?" command to get help on any error messages received. Details on the use of message files can be found in the *TAE Application Programmer's Reference Manual*.
  A null value (empty string) or a single blank given for KEY is equivalent to not specifying the argument.

8.2.9 x/zvselpi—Selects the file to use as the primary input

```
call xvselpi(instance)
zvselpi(instance);
```

Selects the file to use as the primary input. The primary input is normally the first file given in the INP parameter. It is used for several purposes. If file attributes like size, type, etc. aren't specified when opening a file, defaults are taken from the primary input. When creating a new output file, history and property labels for it are copied from the primary input, in order to maintain the processing history and file attributes.

In the rare cases where the first INP file is not appropriate for the primary input, calling **x/zvselpi**

allows you to change the file that is used as the primary input, or to disable the primary input altogether. The file selected must still be one of the files in the INP parameter. For example, you might be taking *n* input files and creating *n* output files after doing the same processing to each. You would want to set the primary input for each output file to the corresponding input file, in order to preserve the history labels. Or, you may want to create a file with no history labels whatsoever (except for the current task).

This routine should be called before the **x/zvopen** of the output file you want associated with the input. The only routines that use the primary input directly are **x/zvopen**, **x/zvsize**, and **x/zvbands**. The primary input in effect at the time each of these routines is called determines which input file is used. You may change **x/zvselpi** after the **x/zvopen** statement, even if you do more processing to the file. It will still use the primary input in effect at the time the file was opened.

It is slightly more efficient to call **x/zvselpi** before you open the primary input file for other reasons, because it avoids an extra file open. However, this should not have a big impact.

There is no status return from this routine.

Arguments:

- **INSTANCE**: integer, input
  Determines which instance of the INP parameter to use as the primary input. Numbering starts at one, so you may restore the default behavior by calling **x/zvselpi** with an INSTANCE of 1. An INSTANCE of 4 would mean the fourth item in the INP parameter, etc.
  If INSTANCE is zero, then the primary input is disabled. Provide all necessary file size and type parameters to **x/zvopen**, since there are no defaults. The history labels also will not be copied, so the current task will be the first (and only) task in the new file's label. This is the primary reason for disabling the primary input.

8.2.10 x/zvselpiu—Selects the file to use as primary input

```
call xvselpiu(unit)
zvselpiu(unit);
```

Selects the file to use as the primary input. The primary input is normally the first file given in the INP parameter. It is used for several purposes. If file attributes like size, type, etc. aren't specified when opening a file, defaults are taken from the primary input. When creating a new output file, history and property labels for it are copied from the primary input, in order to maintain the processing history and file attributes.

In the rare cases where the first INP file is not appropriate for the primary input, calling **x/zvselpiu** allows you to change the file that is used as the primary input. This unit then becomes the primary input. This allows any file, not just one associated with the INP parameter, to be the primary input. The file associated with the unit may be open or closed,but you should not free the unit (CLOS_ACT, FREE) as long as it is the primary input. A primary input unit of 0 is valid. To disable the primary input completely, call **x/zvselpi**(0); do not use **x/zvselpiu**() for this case.

This routine should be called before the **x/zvopen** of the output file you want associated with the input. The only routines that use the primary input directly are **x/zvopen**, **x/zvsize**, and **x/zvbands**. The primary input in effect at the time each of these routines is called determines which input file is used. You may change **x/zvselpiu** after the **x/zvopen** statement, even if you do more processing to the file. It will still use the primary input in effect at the time the file was opened.

It is slightly more efficient to call **x/zvselpiu** before you open the primary input file for other reasons, because it avoids an extra file open. However, this should not have a big impact.

There is no status return from this routine.

Arguments:

- **UNIT**: input, integer
  Unit number of file from **x/zvunit**.

8.2.11 x/zvsize—Return image size values

```
call xvsize(sl, ss, nl, ns, nli, nsi)
zvsize(sl, ss, nl, ns, nli, nsi);
```

x/zvsize searches the user parameters and the first input file for image size values. The values are returned as described below under "arguments". If there is an input file, it must be opened before calling **x/zvsize**.

For 3 dimensional files **x/zvsize** must be supplimented with **x/zvbands**, which returns infomation about bands..

Arguments:

- **SL**: output, integer
  SL is the starting line which should be used when reading the input. If the SIZE parameter was given, SL is taken from the first element in the size field. If SIZE was not given, but the SL parameter was, SL is taken from the SL parameter. Otherwise, SL defaults to a value of one.
- **SS**: output, integer
  SS is the starting sample which should be used when reading the input. If the SIZE parameter was given, SS is taken from the second element in the size field. If SIZE was not given, but the SS parameter was, SS is taken from the SS parameter. Otherwise, SS defaults to a value of one.
- **NL**: output, integer
  NL is the number of lines which should be read from the input and written to the output. If the SIZE parameter was given, NL is taken from the third element in the size field. If SIZE was not given, but the NL parameter was, NL is taken from the NL parameter. Otherwise, NL defaults to the size of the primary input.
  a value of zero for NL is treated as if no value was given.
- **NS**: output, integer
  NS is the number of samples per line which should be read from the input and written to the output. If the SIZE parameter was given, NS is taken from the fourth element in the size field. If SIZE was not given, but the NS parameter was, NS is taken from the NS parameter. Otherwise, NS defaults to the size of the primary input. A value of zero for NS is treated as if no value was given.
- **NLI**: output, integer
  NLI is the number of lines in the primary input. It is obtained from the NL optional to **x/zvget**.
- **NSI**: output, integer
  NSI is the number of samples per line in the primary input. It is obtained from the NS optional to **x/zvget**.

9. Appendix A: Summary of Calling Sequences

This section contains a brief synopsis of the calling sequences for all the RTL routines, in alphabetical order.

For routines that take optional arguments (keyword-value pairs), the allowed keywords are listed. You must have an argument list terminator on every routine that lists optional arguments, even if you don't use them. Some of the keywords listed are not implemented or not useful, but they are allowed by the parameter parsing mechanism.

The actual data type declarations for the data types below are listed. See Table 7: FORTRAN declarations for Run-Time Library arguments (page 25). Also See Table 5: C Declarations for Run-Time Library Arguments (page 23). Keep in mind that a value listed as "output" or "in/out" in a C call must be passed by address, not value.

abend/zabend

```
call abend
zabend();
```

qprint/zqprint

```
call qprint(message, length)
zqprint(message, length);
   message      input     string
   length       input     integer
```

sc2for

```
sc2for(c_string, max_length, for_string, argptr, nargs, argno, strno);
   c_string     input     string
   max_length   input     integer
   for_string   output    fortran string
   argptr       input     void pointer
   nargs        input     integer
   argno        input     integer
   strno        input     integer
```

sc2for_array

```
sc2for_array(c_string, len, nelements, for_string, max_length, argptr,
nargs, argno, strno);
   c_string     input     string array, size nelements
   len          input     integer
   nelements    input     integer
   for_string   output    fortran string array, size nelements
   max_length   in/out    integer
   argptr       input     void pointer
   nargs        input     integer
   argno        input     integer
   strno        input     integer
```

sfor2c

```
sfor2c(c_string, len, for_string, argptr, nargs, argno, strno);
   c_string     output    string
   len          input     integer
   for_string   input     fortran string
   argptr       input     void pointer
   nargs        input     integer
   argno        input     integer
   strno        input     integer
```

sfor2c_array

```
sfor2c_array(c_string, max_length, nelements, for_string, argptr,
nargs, argno, strno);
   c_string     output    pointer to string array, size nelements
   max_length   in/out    integer
   nelements    input     integer
   for_string   input     fortran string array, size nelements
   argptr       input     void pointer
   nargs        input     integer
```

```
   argno         input     integer
   strno         input     integer
```

sfor2len

```
sfor2len(for_string, argptr, nargs, argno, strno);
   for_string    input     fortran string
   argptr        input     void pointer
   nargs         input     integer
   argno         input     integer
   strno         input     integer
```

sfor2ptr

```
ptr = sfor2ptr(for_string);
   for_string    input     fortran string
   ptr           output    string pointer
```

xladd/zladd

```
call xladd(unit, type, key, value, status, <optionals>, ' ')
status = zladd(unit, type, key, value, <optionals>, 0);
   unit          input     integer
   type          input     string
   key           input     string
   value         input     value array, size NELEMENT
   status        output    integer
Optionals allowed:
   ELEMENT       input     integer
   ERR_ACT       input     string
   ERR_MESS      input     string
   FORMAT        input     string
   HIST          input     string
   INSTANCE      input     integer
   LEVEL         input     integer
   MODE          input     string
   NELEMENT      input     integer
   PROPERTY      input     string
   ULEN          input     integer
```

xldel/zldel

```
call xldel(unit, type, key, status, <optionals>, ' ')
status = zldel(unit, type, key, <optionals>, 0);
   unit          input     integer
   type          input     string
   key           input     string
   status        output    integer
Optionals allowed:
   ELEMENT       input     integer
   ERR_ACT       input     string
   ERR_MESS      input     string
   HIST          input     string
   INSTANCE      input     integer
   NELEMENT      input     integer
   NRET          output    integer
   PROPERTY      input     string
```

xlget/zlget

```
call xlget(unit, type, key, value, status, <optionals>, ' ')
status = zlget(unit, type, key, value, <optionals>, 0);
   unit          input     integer
   type          input     string
   key           input     string
   value         output    value array, size NELEMENT
   status        output    integer
Optionals allowed:
   ELEMENT       input     integer
   ERR_ACT       input     string
   ERR_MESS      input     string
   FORMAT        input     string
   HIST          input     string
   INSTANCE      input     integer
   LENGTH        output    integer
   LEVEL         output    integer
   NELEMENT      input     integer
   NRET          output    integer
   PROPERTY      input     string
   ULEN          input     integer
```

xlgetlabel/zlgetlabel

```
call xlgetlabel(unit, buf, bufsize, status)
status = zlgetlabel(unit, buf, bufsize);
   unit          input     integer
   buf           output    string
   bufsize       in/out    integer
   status        output    integer
```

xlhinfo/zlhinfo

```
call xlhinfo(unit, tasks, instances, nhist, status, <optionals>, ' ')
status = zlhinfo(unit, tasks, instances, nhist, <optionals>, 0);
   unit          input     integer
   tasks         output    string array, size nhist
   instances     output    integer array, size nhist
   nhist         in/out    integer
   status        output    integer
Optionals allowed:
   ERR_ACT       input     string
   ERR_MESS      input     string
   NRET          output    integer
   ULEN          input     integer
```

xlinfo/zlinfo

```
call xlinfo(unit,type,key,format,maxlen,nelement,status,<optionals>,'
')
status = zlinfo(unit,type,key,format,maxlen,nelement,<optionals>, 0);
   unit          input     integer
   type          input     string
   key           input     string
   format        output    string
   maxlen        output    integer
   nelement      output    integer
```

```
   status        output    integer
Optionals allowed:
   ERR_ACT       input     string
   ERR_MESS      input     string
   HIST          input     string
   INSTANCE      input     integer
   MOD           output    integer
   PROPERTY      input     string
   STRLEN        output    integer
```

xlninfo/zlninfo

```
call xlninfo(unit, key, format, maxlength, nelement, status,
<optionals>, ' ')
status = zlninfo(unit, key, format, maxlength, nelement,<optionals>,
0);
   unit          input     integer
   key           output    string
   format        output    string
   maxlength     output    integer
   nelement      output    integer
   status        output    integer
Optionals allowed:
   ERR_ACT       input     string
   ERR_MESS      input     string
   MOD           output    integer
   STRLEN        output    integer
```

xlpinfo/zlpinfo

```
call xlpinfo(unit, properties, nprop, status, <optionals>, ' ')
status = zlpinfo(unit, properties, nprop, <optionals>, 0);
   unit          input     integer
   properties    output    string array, size nprop
   nprop         in/out    integer
   status        output    integer
Optionals allowed:
   ERR_ACT       input     string
   ERR_MESS      input     string
   NRET          output    integer
   ULEN          input     integer
   INST_NUM      input     integer array
```

xmove/zmove

```
call xmove(from, to, len)
zmove(from, to, len);
   from          input     pixel buffer, size len (bytes)
   to            output    pixel buffer, size len (bytes)
   len           input     integer
```

xvadd/zvadd

```
call xvadd(unit, status, <optionals>, ' ')
status = zvadd(unit, <optionals>, 0);
   unit          input     integer
   status        output    integer
Optionals allowed:
```

```
   BHOST         input     string
   BIN_CVT       input     string
   BINTFMT       input     string
   BLTYPE        input     string
   BREALFMT      input     string
   CLOS_ACT      input     string
   COND          input     string
   CONVERT       input     string
   FORMAT        input     string
   HOST          input     string
   I_FORMAT      input     string
   INTFMT        input     string
   IO_ACT        input     string
   IO_MESS       input     string
   LAB_ACT       input     string
   LAB_MESS      input     string
   METHOD        input     string
   OP            input     string
   O_FORMAT      input     string
   OPEN_ACT      input     string
   OPEN_MES      input     string
   REALFMT       input     string
   TYPE          input     string
   U_DIM         input     integer
   U_FILE        input     integer
   U_FORMAT      input     string
   U_NB          input     integer
   U_NBB         input     integer
   U_NL          input     integer
   U_NLB         input     integer
   U_NS          input     integer
   U_N1          input     integer
   U_N2          input     integer
   U_N3          input     integer
   U_ORG         input     string
   UPD_HIST      input     string
```

xvbands/zvbands

```
call xvbands(sb, nb, nbi)
zvbands(sb, nb, nbi);
   sb            output    integer
   nb            output    integer
   nbi           output    integer
```

xvclose/zvclose

```
call xvclose(unit, status, <optionals>, ' ')
status = zvclose(unit, <optionals>, 0);
   unit          input     integer
   status        output    integer
Optionals allowed:
   CLOS_ACT      input     string
```

xvcmdout/zvcmdout

```
call xvcmdout(command, status)
status = zvcmdout(command);
```

```
   command       input     string
   status        output    integer
```

xvcommand/zvcommand

```
call xvcommand(command, status)
status = zvcommand(command);
   command       input     string
   status        output    integer
```

xveaction/zveaction

```
status = xveaction(action, message)
status = zveaction(action, message);
   action        input     string
   message       input     string
   status        output    integer
```

xvend/zvend

```
call xvend(status)
zvend(status);
   status        input     integer
```

xvfilpos/zvfilpos

```
position = xvfilpos(unit)
position = zvfilpos(unit);
   unit          input     integer
   position      output    integer
```

xvget/zvget

```
call xvget(unit, status, <optionals>, ' ')
status = zvget(unit, <optionals>, 0);
   unit          input     integer
   status        output    integer
Optionals allowed:
   BHOST         output    string
   BINTFMT       output    string
   BLTYPE        output    string
   BREALFMT      output    string
   BUFSIZ        output    integer
   DIM           output    integer
   FLAGS         output    integer
   FORMAT        output    string
   HOST          output    string
   IMG_REC       output    integer
   INTFMT        output    string
   LBLSIZE       output    integer
   NAME          output    string
   NB            output    integer
   NBB           output    integer
   NL            output    integer
   NLB           output    integer
   NS            output    integer
   N1            output    integer
   N2            output    integer
```

```
   N3            output    integer
   ORG           output    string
   PIX_SIZE      output    integer
   REALFMT       output    string
   RECSIZE       output    integer
   TYPE          output    string
   VARSIZE       output    integer
   UPD_HIST      input     string
```

xvhost/zvhost

```
call xvhost(host, intfmt, realfmt, status)
status = zvhost(host, intfmt, realfmt);
   host          input     string
   intfmt        output    string
   realfmt       output    string
   status        output    integer
```

xvintract/zvintract

```
call xvintract(subcmd, prompt)
zvintract(subcmd, prompt);
   subcmd        input     string
   prompt        input     string
```

xvip/zvip

```
call xvip(name, value, count)
status = zvip(name, value, count);
   name          input     string
   value         output    value array, size count
   count         output    integer
   status        output    integer
```

xviparm/zviparm

```
call xviparm(name, value, count, def, maxcnt)
status = zviparm(name, value, count, def, maxcnt, length);
   name          input     string
   value         output    value array, size count
   count         output    integer
   def           output    integer
   maxcnt        input     integer
   length        input     integer
   status        output    integer
```

xviparmd/zviparmd

```
call xviparmd(name, value, count, def, maxcnt)
status = zviparmd(name, value, count, def, maxcnt, length);
   name          input     string
   value         output    value array, size count
   count         output    integer
   def           output    integer
   maxcnt        input     integer
   length        input     integer
   status        output    integer
```

xvipcnt/zvipcnt

```
call xvipcnt(name, count)
status = zvipcnt(name, count);
   name          input     string
   count         output    integer
   status        output    integer
```

xvipone/zvipone

```
status = xvipone(name, value, instance, maxlen)
status = zvipone(name, value, instance, maxlen);
   name          input     string
   value         output    value
   instance      input     integer
   maxlen        input     integer
```

xvipstat/zvipstat

```
call xvipstat(name, count, def, maxlen, type)
status = zvipstat(name, count, def, maxlen, type);
   name          input     string
   count         output    integer
   def           output    integer
   maxlen        output    integer
   type          output    string
   status        output    integer
```

xviptst/zviptst

```
GIVEN = xviptst(key)
GIVEN = zviptst(key);
   key           input     string
```

xvmessage/zvmessage

```
call xvmessage(message, key)
zvmessage(message, key);
   message       input     string
   key           input     string
```

xvopen/zvopen

```
call xvopen(unit, status, <optionals>, ' ')
status = zvopen(unit, <optionals>, 0);
   unit          input     integer
   status        output    integer
Optionals allowed:
   ADDRESS       output    pixel pointer
   BHOST         input     string
   BIN_CVT       input     string
   BINTFMT       input     string
   BLTYPE        input     string
   BREALFMT      input     string
   CLOS_ACT      input     string
   COND          input     string
   CONVERT       input     string
   FORMAT        input     string
```

```
   HOST          input     string
   I_FORMAT      input     string
   INTFMT        input     string
   IO_ACT        input     string
   IO_MESS       input     string
   LAB_ACT       input     string
   LAB_MESS      input     string
   METHOD        input     string
   OP            input     string
   O_FORMAT      input     string
   OPEN_ACT      input     string
   OPEN_MES      input     string
   REALFMT       input     string
   TYPE          input     string
   U_DIM         input     integer
   U_FILE        input     integer
   U_FORMAT      input     string
   U_NB          input     integer
   U_NBB         input     integer
   U_NL          input     integer
   U_NLB         input     integer
   U_NS          input     integer
   U_N1          input     integer
   U_N2          input     integer
   U_N3          input     integer
   U_ORG         input     string
   UPD_HIST      input     string
```

xvp/zvp

```
call xvp(name, value, count)
status = zvp(name, value, count);
   name          input     string
   value         output    value array, size count
   count         output    integer
   status        output    integer
```

xvparm/zvparm

```
call xvparm(name, value, count, def, maxcnt)
status = zvparm(name, value, count, def, maxcnt, length);
   name          input     string
   value         output    value array, size count
   count         output    integer
   def           output    integer
   maxcnt        input     integer
   length        input     integer
   status        output    integer
```

xvparmd/zvparmd

```
call xvparmd(name, value, count, def, maxcnt)
status = zvparmd(name, value, count, def, maxcnt, length);
   name          input     string
   value         output    value array, size count
   count         output    integer
   def           output    integer
   maxcnt        input     integer
```

```
   length        input     integer
   status        output    integer
```

xvpblk/zvpblk

```
call xvpblk(parblk)
zvpblk(parblk);
   parblk        output    void pointer
```

xvpclose/zvpclose

```
call xvpclose(status)
status = zvpclose();
   status        output    integer
```

xvpcnt/zvpcnt

```
call xvpcnt(name, count)
status = zvpcnt(name, count);
   name          input     string
   count         output    integer
   status        output    integer
```

xvpixsize/zvpixsize

```
status = xvpixsize(pixsize, type, ihost, rhost)
status = zvpixsize(pixsize, type, ihost, rhost);
   pixsize       output    integer
   type          input     string
   ihost         input     string
   rhost         input     string
   status        output    integer
```

xvpixsizeb/zvpixsizeb

```
status = xvpixsizeb(pixsize, type, unit)
status = zvpixsizeb(pixsize, type, unit);
   pixsize       output    integer
   type          input     string
   unit          input     integer
   status        output    integer
```

xvpixsizeu/zvpixsizeu

```
status = xvpixsizeu(pixsize, type, unit)
status = zvpixsizeu(pixsize, type, unit);
   pixsize       output    integer
   type          input     string
   unit          input     integer
   status        output    integer
```

xvpone/zvpone

```
status = xvpone(name, value, instance, maxlen)
status = zvpone(name, value, instance, maxlen);
   name          input     string
   value         output    value
   instance      input     integer
```

```
   maxlen        input     integer
```

xvpopen/zvpopen

```
call xvpopen(status, n_par, max_parm_size, filename, error_act, unit)
status = zvpopen(filename, error_act, unit);
   status        output    integer
   n_par         input     integer (obsolete)
   max_parm_size input     integer (obsolete)
   filename      input     string
   error_act     input     string
   unit          output    integer
```

xvpout/zvpout

```
call xvpout(status, name, value, format, count)
status = zvpout(name, value, format, count, length);
   status        output    integer
   name          input     string
   value         input     value array, size count
   format        input     string
   count         input     integer
   length        input     integer
```

xvpstat/zvpstat

```
call xvpstat(name, count, def, maxlen, type)
status = zvpstat(name, count, def, maxlen, type);
   name          input     string
   count         output    integer
   def           output    integer
   maxlen        output    integer
   type          output    string
   status        output    integer
```

xvptst/zvptst

```
GIVEN = xvptst(key)
GIVEN = zvptst(key);
   key           input     string
```

xvread/zvread

```
call xvread(unit, buffer, status, <optionals>, ' ')
status = zvread(unit, buffer, <optionals>, 0);
   unit          input     integer
   buffer        output    pixel buffer
   status        output    integer
Optionals allowed:
   BAND          input     integer
   IO_MESS       input     string
   LINE          input     integer
   METHOD        input     string
   NBANDS        input     integer
   NLINES        input     integer
   NSAMPS        input     integer
   OP            input     string
   OPEN_ACT      input     string
```

```
   SAMP          input     integer
   U_FORMAT      input     string
```

xvselpi/zvselpi

```
call xvselpi(instance)
zvselpi(instance);
   instance      input     integer
```

xvsfile/zvsfile

```
status = xvsfile(unit, file)
status = zvsfile(unit, file);
   unit          input     integer
   file          input     integer
   status        output    integer
```

xvsignal/zvsignal

```
call xvsignal(unit, status, abend_flag)
zvsignal(unit, status, abend_flag);
   unit          input     integer
   status        input     integer
   abend_flag    input     integer
```

xvsize/zvsize

```
call xvsize(sl, ss, nl, ns, nli, nsi)
zvsize(sl, ss, nl, ns, nli, nsi);
   sl            output    integer
   ss            output    integer
   nl            output    integer
   ns            output    integer
   nli           output    integer
   nsi           output    integer
```

xvsptr/zvsptr

```
call xvsptr(value, count, offsets, lengths)
zvsptr(value, count, offsets, lengths);
   value         input     string
   count         input     integer
   offsets       output    integer array, size count
   lengths       output    integer array, size count
```

xvtpinfo/zvtpinfo

```
status = xvtpinfo(sym_name, dev_name, tfile, trec)
status = zvtpinfo(sym_name, dev_name, tfile, trec);
   sym_name      input     string
   dev_name      output    string
   tfile         output    integer
   trec          output    integer
   status        output    integer
```

xvtpmode/zvtpmode

```
call xvtpmode(unit, istape)
```

```
istape = zvtpmode(unit);
   unit          input     integer
   istape        output    integer
```

xvtpset/zvtpset

```
status = xvtpset(name, tfile, trec)
status = zvtpset(name, tfile, trec);
   name          input     string
   tfile         input     integer
   trec          input     integer
   status        output    integer
```

xvtrans/zvtrans

```
call xvtrans(buf, source, dest, npix)
zvtrans(buf, source, dest, npix);
   buf           input     integer array, size 12
   source        input     pixel buffer, size npix
   dest          output    pixel buffer, size npix
   npix          input     integer
```

xvtrans_in/zvtrans_in

```
call xvtrans_in(buf, stype, dtype, sihost, srhost, status)
status = zvtrans_in(buf, stype, dtype, sihost, srhost);
   buf           output    integer array, size 12
   stype         input     string
   dtype         input     string
   sihost        input     string
   srhost        input     string
   status        output    integer
```

xvtrans_inb/zvtrans_inb

```
call xvtrans_inb(buf, stype, dtype, unit, status)
status = zvtrans_inb(buf, stype, dtype, unit);
   buf           output    integer array, size 12
   stype         input     string
   dtype         input     string
   unit          input     integer
   status        output    integer
```

xvtrans_inu/zvtrans_inu

```
call xvtrans_inu(buf, stype, dtype, unit, status)
status = zvtrans_inu(buf, stype, dtype, unit);
   buf           output    integer array, size 12
   stype         input     string
   dtype         input     string
   unit          input     integer
   status        output    integer
```

xvtrans_out/zvtrans_out

```
call xvtrans_out(buf, stype, dtype, dihost, drhost, status)
status = zvtrans_out(buf, stype, dtype, dihost, drhost);
   buf           output    integer array, size 12
```

```
   stype         input     string
   dtype         input     string
   dihost        input     string
   drhost        input     string
   status        output    integer
```

xvtrans_set/zvtrans_set

```
call xvtrans_set(buf, stype, dtype, status)
status = zvtrans_set(buf, stype, dtype);
   buf           output    integer array, size 12
   stype         input     string
   dtype         input     string
   status        output    integer
```

xvunit/zvunit

```
call xvunit(unit, name, instance, status, <optionals>, ' ')
status = zvunit(unit, name, instance, <optionals>, 0);
   unit          output    integer
   name          input     string
   instance      input     integer
   status        output    integer
Optionals allowed:
   U_NAME        input     string
```

xvwrit/zvwrit

```
call xvwrit(unit, buffer, status, <optionals>, ' ')
status = zvwrit(unit, buffer, <optionals>, 0);
   unit          input     integer
   buffer        input     pixel buffer
   status        output    integer
Optionals allowed:
   BAND          input     integer
   LINE          input     integer
   NBANDS        input     integer
   NLINES        input     integer
   NSAMPS        input     integer
   SAMP          input     integer
   U_NL          input     integer
```

xvzinit

```
call xvzinit(lun, flag, debug)
   lun           input     integer
   flag          output    integer
   debug         output    integer
```

zvpinit

```
zvpinit(parb);
   parb          input     void pointer
```

zv_rtl_init

```
status = zv_rtl_init();
   status        output    integer
```

10. Appendix B: Error Messages

10.1 Error message format

This Section describes the meaning of the VICAR error messages. VICAR error messages are given in the following form:

[VIC2-—*key*] *message*

where VIC2 indicates that the message was issued from the VICAR2 package, and *key* is the specific key or identifier for the message given. The message key may be used to ask for help with the HELP-MESSAGE command in the VICAR supervisor or to look up a message in this Section. In addition, the key is stored internally by the supervisor, so that by simply typing a question mark (?) to the prompt, help on the last error message given is received.

10.2 Messages by key

This Section lists VICAR2 error messages in alphabetical order by key. The accompanying message, the numerical value, and the symbolic name by which the error may be referenced in a program are given, followed by a detailed description of what the message means, and the action required to correct the error.

- **ALROPN Symbolic Name**: FILE_IS_ALREADY_OPEN
  [VIC2-ALROPN] Attempt to open an open file; program error
  Explanation: **x/zvopen** has been called on a file which is already open. Program error. User action: Please notify the cognizant programmer.

- **BADBAND Symbolic Name**: IMPROPER_BAND_SIZE_PARAM
  [VIC2-BADBAND] Improper band size parameter; program error
  Explanation: A band size argument (BAND, NBANDS, U_NB) was provided to the indicated routine and was not in an allowable range (usually less than zero).
  User action: Check all the parameters which could have led to the bad sample size. If they are good, it is probably a program error and the cognizant programmer should be consulted.

- **BADBINSIZ Symbolic Name**: IMPROPER_ BINARY _SIZE_PARAM
  [VIC2-BADBINSIZ] Improper binary size parameter; program error
  Explanation: A binary size argument (U_NBB or U_NLB) was provided to the indicated routine and was not in an allowable range (usually less than zero).
  User action: Check all the parameters which could have led to the bad binary size. If they are good, it is probably a program error and the cognizant programmer should be consulted.

- **BADDIM Symbolic Name**: BAD_DIM_NAME
  [VIC2-BADDIM] Bad name for an image dimension (DIM1etc.).
  Explanation:
  A bad name was given for one of the dimensions in a multi-dimensional file. This error is the result of a call to **x/zvopen** or **x/zvadd** in which the DIM1.. DIM4 or the U_DIM1.. U_DIM4 optional arguments were given with a value which was not one of: SAMP, LINE, BAND, or TIME.
  User Action: If you specified the name for the dimension on the command line, re-specify it with one of the above names. If not, this error represents a program error, so notify the cognizant programmer. If there is a need to add a new name to the list, notify the MIPL system engineer.

- **BADELEM Symbolic Name**: IMPROPER_ELEMENT_NUMBER
  [VIC2-BADELEM] Improper element number; program error
  Explanation: A value for ELEMENT or NELEMENTS was provided to a label routine that was not in the allowable range (usually less than zero).
  User action: Check all the parameters which could have led to the bad element number. If they are good, it is probably a program error and the cognizant programmer should be consulted.

- **BADFILE Symbolic Name**: IMPROPER_FILE_NUMBER

[VIC2-BADFILE] Bad file number for tape
Explanation: The U_FILE optional is less than zero.
User action: this is a program error. Please consult the cognizant programmer. Programmer action: Check the value of U_FILE.

- **BADFOR Symbolic Name**: IMPROPER_FORMAT_STRING
[VIC2-BADFOR] Improper FORMAT string; program error
Explanation: The pixel format being passed to the indicated routine is not one of the valid values. This is a program error.
User action: Please consult the cognizant programmer to verify that a valid format string is being passed. Programmer action: make sure that the FORMAT argument to the indicated routine is one of following values: BYTE, HALF, FULL, REAL, DOUB, or COMP.

- **BADINST Symbolic Name**: ILLEGAL_INSTANCE
[VIC2-BADINST] Illegal instance; program error
Explanation: In a call to **x/zvunit**, the value given to the INSTANCE argument is greater than the actual number of files given. User action: Verify that all the proper parameters have been input. If they have then this message indicates that a program error has occurred and the cognizant programmer should be consulted. Programmer action: Modify the PDF to require that the user input the appropriate number of files.

- **BADLBL Symbolic Name**: BAD_ INPUT_LABEL
[VIC2-BADLBL] Bad input label; check file contents
Explanation: An error occurredwhile parsing the input label. Either the file has no valid VICAR2 label, or the label is corrupted. User action: If the file has no label, LABEL-CREATE may be used to create one. If the file has a label but this error still occurs, then the label has probably been corrupted, which could indicate an executive error. In the latter case, consult the system programmer.

- **BADLBLTP Symbolic Name**: BAD_LABEL_TYPE
[VIC2-BADLBLTP] Bad label type; check file contents
Explanation: A label type other than HISTORY or SYSTEM has been input to the indicated routine, or neither has been specified. User action: The validity of the label type must be checked. Please consult the cognizant programmer
Programmer action: Verify that the call to the indicated routine contains a SYSTEM or HISTORY label, or that a valid label type was specified.

- **BADLEN Symbolic Name**: IMPROPER_LENGTH
[VIC2-BADLEN] Improper length; program error
Explanation:
ULEN was either required for the indicated routine and not given, or it was given an invalid value.
User action: The size and existence of the value given to ULEN must be checked. Please consult the cognizant programmer.
Programmer action:
The value of ULEN must be positive and less than the maximum string size. It is required to be given for multi-valued string items passed from C, and all string items (multi-valued or single value) passed from FORTRAN declared with a BYTE or LOGICAL*1. Strings passed from FORTRAN declared asCHARACTER do not require the ULEN optional.

- **BADLINE Symbolic Name**:IMPROPER_LINE_SIZE_PARAM
[VIC2-BADLINE] Improper line size parameter; program error
Explanation: A line size argument (LINE, NLINES, U_NL) was provided to the indicated routine and was not in an allowable range (usually less than zero). User action: Check all the parameters which could have led to the bad sample size. If they are good, it is probably a program error and the cognizant programmer should be consulted.

- **BADLINST Symbolic Name**:IMPROPER_LABEL_INSTANCE
[VIC2-BADLINST] Improper label instance number; program error

Explanation: An INSTANCE number was provided to a label routine that was not in the allowable range (usually less than zero). User action: Check all the parameters whichcould have led to the bad instance. If they are good, it is probably a program error and the cognizant programmer should be consulted.

- **BADMETH Symbolic Name**: IMPROPER_METHOD_STRING
  [VIC2-BADMETH] Improper METHOD string; program error
  Explanation: The METHOD optionalargument contained an invalid value for the routine indicated. This is a program error. The valid values for METHOD are RANDOM, SEQ. User action: Please consult the cognizant programmer so that the value of METHOD can be checked.

- **BADMODESTR Symbolic Name**:IMPROPER_MODE_STRING
  [VIC2-BADMODESTR] Improper MODE string; program error
  E xplanation: The MODE optional parameter had an invalid value associated with it in the indicated routine. This is a program error.
  User action: Please consult the cognizant programmer so that the value of MODE can be checked.
  Programmer action: The only valid values for MODE are “ADD”, “INSERT”, and “REPLACE”. Check that the call to **x/zvopen** uses one of these values.

- **BADNAM Symbolic Name**: BAD_FILE_PARAM_NAME
  [VIC2-BADNAM] Bad file parameter name; program error
  Explanation: An error was encountered when **x/zvunit** tried to look up a file name in the command line. This error probably indicates a mismatch between the program and the PDF file associated withit. User action: Please consult the cognizant programmer so that any disparity between the program and the PDF can be corrected.

- **BADOPR Symbolic Name**: IMPROPER_OPERATION
  [VIC2-BADOPR] Operation conflicts with open attributes; program error
  Explanation:Improper operation. The program tried to perform an operation on the indicated file which is forbidden in the mode in which the file has been opened. For instance, trying to writeto a file which has been opened for READ. User action: this is generally a program error. Consult the cognizant programmer for the program in question.

- **BADOPSTR Symbolic Name**: IMPROPER_OP_STRING
  [VIC2-BADOPSTR] Improper OP string; program error
  Explanation: The OP optional parameter had an invalid value associated with it in the indicated routine. This is a program error. User action: Please consult the cognizant programmer so that the value of OP can be checked. Programmer action: The only valid values for OP are “READ”,”WRITE”, and “UPDATE”. Check that the call to **x/zvopen** uses one of these values.

- **BADORG Symbolic Name**: BAD_ORG
  [VIC2-BADORG]ORG key word (file organization) is not valid.
  Explanation:
  The file organizationkey word-ORG -passed to the indicated routine is not one of the valid values. The valid file organizations are BSQ (band sequential), BIL (Band interleaved by line), and BIP(Band interleaved by pixel). User Action: Notify the cognizant programmer of the program which failed or MIPL.

- **BADSAMP Symbolic Name**:IMPROPER_SAMP_SIZE_PARAM
  [VIC2-BADSAMP] Improper sample size parameter; program error
  Explanation: A sample size argument (SAMP, NSAMPS, U_NS) was provided to the indicated routine and was not in an allowable range (usually less than zero). User action: Check all the parameters which could have led to the bad sample size. If they are good, it is probably a program error and the cognizant programmer should be consulted.

- **BADSIZ Symbolic Name**:IMPROPER_IMAGE_SIZE_PARAM
  [VIC2-BADSIZ] Improper image size parameter; program error
  Explanation: A size argument (U_N1, U_N2, U_N3, or U_N4) was provided to the indicated routine and was not in an allowable range (usually less than zero). User action: Check all the parameters

which could have led to the bad size. If they are good, it is probably a program error and the cognizant programmer should be consulted.

- **BADTRANS Symbolic Name**:INVALID_FORMAT_TRANSLATION
  [VIC2-BADTRANS] Invalid format translation
  Explanation:
  The data format translation specified with U_FORMAT is not legal. Currently no translations are allowed for DOUB data formats. User action: make sure you specified the correct file.
  Programmer action: Don't allow conversions to/from DOUBle format.

- **BUG Symbolic Name**: INTERNAL_ERROR
  [VIC2-BUG]Internal VICAR bug check failure —Notify system programmer
  Explanation: An internal bug check in the VICAR run-time library failed, indicating a bug in the run-time library. User Action: this error should never occur, hence please notify the VICAR system programmer immediately.

- **CONVERR Symbolic Name**: CONVERSION_ERROR
  [VIC2-CONVERR] Conversion error; program error
  Explanation: The data item passed to the indicated routine is incompatible with the FORMAT parameter. For example, an integer is expected and a real is passed. User action: If the parameter is user specified check to see that the data item being passed to the routine is in the expected format. If the problem is still not obvious, consult the cognizant programmer. Programmer action: Check the value of the item passed to the the indicated routine.

- **DUPKEY Symbolic Name**: DUPLICATE_KEY
  [VIC2-DUPKEY] Duplicate key; program error
  Explanation: An attempt has been made to add a label item with the label routine, **x/zladd**, for a key that already exists. A duplicate item cannot be added under the same HISTORY subset. User action: If the item is user specified a new subset must be created to accept the item. If the item is not user specified and this message is displayed consult the cognizant programmer.

- **END OFVOL Symbolic Name**: END_OF_VOLUME
  [VIC2-END OFVOL] End of volume (double tape mark) reached
  Explanation: The end of volume mark (double tape mark or double end of file) was hit when trying to open a file on an input tape. User action: Scan the tape to determine the actual number of files on it, and make sure that the program does not try to access a file beyond that number.

- **EOF Symbolic Name**: END_OF_FILE
  [VIC2-EOF] End of file
  Explanation: The end of file was reached. For input disk files, this error probably indicates an attempt to read beyond the number of lines in the image. Otherwise, this error indicates that the physical end of file was reached on an I/O operation. User action: this could be either a user or a programmer error. Verify that all the parameters to the program in question are good values, and if so, consult the cognizant programmer.

- **EOLAB Symbolic Name**: END_OF_LABEL
  [VIC2-EOLAB] End of label
  Explanation: you have reached the end of the label on an **x/zlninfo**. This is an informational message which is used as a flag within a routine. User action: If this message is displayed consult the cognizant programmer.

- **ERRACT Symbolic Name**: BAD_ERR_ACT_ VALUE
  [VIC2-ERRACT] Bad IO_ACT, OPEN_ACT, or ERR_ACT; program error
  Explanation: Arguments IO_ACT, OPEN_ACT, and ERR_ACT must contain the characters: 'U', 'A', or 'S', or any combination thereof. User action: The contents of the arguments must be checked. Please consult the cognizant programmer. Programmer action: Verify that the arguments IO_ACT, ERR_ACT, and OPEN_ACT contain the characters 'U', 'S', 'A', alone or in any combination.

- **FILETYPE Symbolic Name**: BAD_FILE_TYPE

[VIC2-FILETYPE] Invalid file type
Explanation:
The file type (the TYPE optionalto the indicated routine) was not a supported type. Currently the types supported are'IMAGE', 'PARM', and 'PARAM', and the IBIS types 'GRAPH1', 'GRAPH2', 'GRAPH3', and'TABULAR'. User action: If the file type was specified on the command line (eg, you tried to use LABEL-REPLACE to change it), then use one of the valid types. Otherwise, it may be a program error. Programmer action: Use only the valid types listed above.

- **FNDKEY Symbolic Name**: CANNOT_FIND_KEY
[VIC2-FNDKEY] Cannot find key; program error
Explanation: The indicated routine is unable to find the specified key in the label.
User action: If the key is user specified, check the label to verify that the label item exists. If, after checking, the problem is not obvious, consult the cognizant programmer.

- **FORREQ Symbolic Name**: FORMAT_ OPTIONAL_REQUIRED
[VIC2-FORREQ] The FORMAT optional is required with xladd
Explanation: The FORMAT optional cannot be defaulted.
User action: Enter the desired FORMAT.

- **HSTNTASC Symbolic Name**: HIST_NAME_HAS_NON_ASCII_CHAR
[VIC2-HSTNTASC] History name has non-ASCII characters;program error
Explanation: HIST name has a non-ASCII character. The HIST optional for the label processing routines was given a name which contains a character which is not a valid ASCII character. The HIST item (TASK name) must be a valid ASCII string, less than 8 characters long. User action: this is most likely a program error. Please consult the cognizant programmer.

- **ILLFOREQ Symbolic Name**: ILLEGAL_FORMAT_REQUEST
[VIC2-ILLFOREQ] Illegal format request; program error
Explanation: Type is not STRING, INT, or REAL. User action: Consult the cognizant programmer so that the validity of the type specified can be checked.

- **INSUFMEM Symbolic Name**: INSUFFICIENT_MEMORY
[VIC2-INSUFMEM] Insufficient memory; consult system programmer
Explanation: Insufficient memory for operation. VICAR2 was not able to allocate sufficient memory foran internal function. User action: this error probably indicates a memory quota was exceeded. See the system manager or system programmer to determine the exact cause and perhaps increase the amount of memory available.

- **LBLIO Symbolic Name**: LABEL_IO_ERROR
[VIC2-LBLIO] Label I/O error; check file contents
Explanation: Usually a system error or VICAR2 bug. User action: Inform VICAR2 system programmer.

- **LNGACT Symbolic Name**: ACT_STRING_TOO_LONG
[VIC2-LNGACT] ACT string too long; program error
Explanation: The argument CLOS_ACT is longer than the maximum number of allowed characters.
User action: The size of, CLOS_ACT must be checked. Please consult the cognizant programmer.
Programmer action: Verify that the string passed is the proper length.

- **LNGCOND Symbolic Name**: COND_STRING_TOO_LONG
[VIC2-LNGCOND] COND string too long; program error
Explanation: COND string has exceeded maximum number of allowable characters. User action: The size of the string, COND, must be checked. Please consult the cognizant programmer.
Programmer action: Check COND argument in indicated routine and verify that it is within the allowable stringlength.

- **LNGHST Symbolic Name**: HISTORY_NAME_TOO_LONG
[VIC2-LNGHST] History name too long; program error
Explanation: The history NAME is too long. The HIST optional for the label processing routines

was given a name which contains more than eight characters. The HIST item (TASK name) must be a valid ASCII string, less than 8 characters long. User action: Most likely a program error. Consult the cognizant programmer.

- **LNGMES Symbolic Name**: ERROR_MESS_TOO_LONG
  [VIC2-LNGMES] Error message too long; program error. Not used.
- **LNGMESS Symbolic Name**: ERR_MESS_TOO_LONG
  [VIC2-LNGMESS] Error message too long; program error
  Explanation: Error message has exceeded the maximum string size allowed (132 characters). User action: Consult the cognizant programmer.
- **MODINPLBL Symbolic Name**:CAN_NOT_MODIFY_AN_ INPUT_LABEL
  [VIC2-MODINPLBL] Attempt to modify input label; program error
  Explanation: An attempt was made to modify a read only file. User action:Consult the cognizant programmer.
- **MODOPNUN Symbolic Name**: CANNOT_MOD_OPEN_UNIT
  [VIC2-MODOPNUN] An open unit cannot be modified; call **x/zvclose** first
  Explanation: An attempt to call the routine **x/zvadd** was made after a unit was already open. In order to prevent the internal table from being corrupted, this operation is defined as illegal. User action: Consult the cognizant programmer. Programmer action: Prevent the calling of **x/zvadd** after a unit has been opened. NL may be updated by using only **x/zladd**, and other fields may not be modified after a unit is opened.
- **MULTPARMFILE Symbolic Name**:MULTIPLE_PARAMETER_FILES
  [VIC2-MULTPARMFILE] Multiple parameter files cannot be open at once
  Explanation: An attempt was made to open a parameter file before the previous one was closed. Only one parameter file can be open at a time. User action: this is a program error. Consult the cognizant programmer. Programmer action: Finish processing one parameter file before using another. Make sure that all parameter files are closed using **x/zvpclose**. **x/zvclose** will not properly close a parameter file.
- **NOARRAY Symbolic Name**: ARRAY_IO_NOT_ALLOWED
  [VIC2-NOARRAY] Array I/O not allowed to non-disk device
  Explanation: An attempt was made to open a non-disk file for array I/O. The program being used was designed only to work on disk files. User action: Use a disk file for the file on which the program failed.
- **NOBINTRAN Symbolic Name**:NO_TRANSLATION_WITH_ BINARY
  [VIC2-NOBINTRAN] binary labels not allowed with dataformat conversions
  Explanation: binary labels may not be accessed (via'COND',' BINARY ') when data format translation is needed. User action: make sure you specified the correct file. Programmer action: Don't use 'COND',' BINARY' unless you are reading from the file in its “natural” format. If extract the labels (binary header or binary prefix), open it once with binary access, get the labels, then close it and re-open it with a different U_FORMAT.
- **NOEFLG Symbolic Name**:UNABLE_TO_ACQUIRE_AN_EVENT_FLAG
  [VIC2-NOEFLG] Unable to get EF; re-try or consultsystem programmer
  Explanation: Unable to acquire an event flag. VICAR2 I/O needs event flags to control the different file units. This error generally means that the system provided event flags are all being used.
  User action: this is probably a program error, caused by having too many files open at once or by using event flags for some other purpose. Consult the cognizant programmer.
- **NOFREUN Symbolic Name**: NO_FREE_UNITS
  [VIC2-NOFREUN] No free units available
  No free units. Explanation: The maximum number of units has been reached. This error generally means that too many files are opened at once. User action: Consult the cognizant programmer for the progra m in which the error occurred and determine whether or not it is a programe rror. If not,

consult the system programmer.

- **NOKEY Symbolic Name**: NO_SUCH_KEY
  [VIC2-NOKEY]No such key in the indicated task
  Explanation: The indicated routine is unable to find the specified key in the label. This can be either a user or programmer error. User action: If the key is user specified, check the label to verify that the label item exists. If, after checking, the pr oblem is not obvious, consult the cognizant programmer.

- **NOLAB Symbolic Name**: FILE_HAS_NO_LABEL
  [VIC2-NOLAB] File has no label; check file contents
  Explanation: input file does not contain a valid VICAR2 label. User action: Check file to verify that a label does exists. LABEL-CREATE may be used to create a label.

- **NOLBL Symbolic Name**: NO_SYSTEM_LABEL
  [VIC2-NOLBL] No system label; check file contents
  Explanation: Unlabeled file. User action: Use the program LABEL to create a label for the file.

- **NOMEM Symbolic Name**:NO_MEMORY_FOR_LABEL_PROCESS
  [VIC2-NOMEM] No memory for label process; consult system programmer
  Explanation: Memory required for label processing is dynamically allocated. There is insufficient memory for operation. User action: this is a system error which probably indicates a memory quot a was exceeded. See the system manager or system programmer to determine the exact cause and perhaps increase the amount of memory available.

- **NONASC Symbolic Name**:STRING_HAS_NON_ASCII_CHARS
  [VIC2-NONASC] String has non-ASCII characters
  Explanation: A string was passed to the indicated routine which contained characters which are not valid ASCII characters. This generally indicates a program error. User action: The contents of the string should be checked. Please consult the cognizant programmer. Programmer action: Use the debugger or a print statement to determine what the indicated string contains.

- **NONSEQWRITSymbolic Name**:NON_SEQUENTIAL_WRITE
  [VIC2-NONSEQWRIT] A non-sequential write was attempted on a sequential—only device
  Explanation: Some devices, such as tapes, only allow sequential access on writes. The program attempted to either back up and re-write a recordor write only part of a record to a file on a sequential device. User action: If the program requires a file, use a disk or memory file. If the problem is not obvious consult the cognizant programmer.

- **NOSCHTSK Symbolic Name**: NO_SUCH_TASK
  [VIC2-NOSCHTSK] No such task in label
  Explanation: The requested task was not in the history label of the image file. This could be either a program or user error. User action: Use “LABEL-LIST 'TASK” on the image file to determine the tasks contained in the label. If a task name is being given on the command line, check the detailed help for the program being used to verify that the correct syntax is being used. If the problem is still not evident, consult the cognizant programmer.

- **NOSCHUN Symbolic Name**: NO_SUCH_UNIT
  [VIC2-NOSCHUN] No such unit; probable error in unit number
  Explanation: An action was requested on a nonexistent unit. This error generally indicates that the program failed tocall **x/zvunit** before using a unit number, or that the variable which contains the unit number has been inadvertantly written over. User action: Verify that the unit number being used is valid. If this is not the problem then consult the cognizant programmer.

- **NOT ASK Symbolic Name**: NO_TASKS_IN_LABEL
  [VIC2-NOT ASK] No tasks in label;check file contents or create new label
  Explanation:The history label of the indicated file does not contain any history subsets, denoted bythe TASK key word. This situation is usually caused by the user deleting the history label. User action: Use the program LABEL or the DCL DUMP facility to examine the label of the image file in question. If the label contains any TASK key words, there is probably an executive bug, so consult

the system programmer. If not, and a history label is desired, use LABEL-REMOVE to remove the system label, and LABEL-CREATE to create a new one.

- **NOT AVAIL Symbolic Name**: NOT_IMPLEMENTED
  [VIC2-NOT AVAIL] Function is not yet implemented; Program error.
  Explanation: The indicated routine was called with an optional argument which although listed in the programmer's reference manual has not yet been implemented. User Action: this indicates a program error, so consult the cognizant programmer or inform MIPL of the bug. Programmer Action: Find another way to accomplish the same function if possible. If the function is not indicated as being unimplemented in the VICAR RTL programmer reference manual, inform MIPL of the problem.

- **NOTERM Symbolic Name**: NO_IO_TO_TERMINAL
  [VIC2-NOTERM] Terminal not allowed for file name, use another name
  Explanation: Terminal not allowed for I/O operation. The file name given to a program points to the user's terminal (for example the logical name TT). Data can not be written to the terminal. User action: Use another file name.

- **NOTEXTEND Symbolic Name**:CANNOT_EXTEND_ARRAY_FILE
  [VIC2-NOTEXTEND] Cannot extend array file, consult cognizant programmer
  Explanation: An attempt was made by the executive to extend an array file. Array files currently may not be extended. User action: this is an executive error and the system programmer should be consulted.

- **NOTEXTMEM Symbolic Name**:CANNOT_EXTEND_MEMORY_FILE
  [VIC2-NOTEXTMEM] Cannot extend memory file, consult cognizant programmer
  Explanation: An attempt was made by the executive to extend a memory file. Memory files currently may not be extended. User action: this is an executive error and the system programmer should be consulted.

- **NOTMOUNTED Symbolic Name**: DEVICE_NOT_MOUNTED
  [VIC2-NOTMOUNTED] A file open was attempted on a tape device that is not mounted
  Explanation: Magnetic tape devices must be mounted prior to use with the MOUNT command.
  User action: Mount the tape with the MOUNT command, and try opening the file again.

- **NOTMULT Symbolic Name**: BUF_NOT_MULT_OF_REC
  [VIC2-NOTMULT] Tape blocksize is not an integral number of records
  Explanation: For a tape which is being blocked, the block size for the tape (usually given by BLOCKINGin the mount command) is not an integral number of records. This case is not allowed because it makes reading the tape difficult and puts da ta in a non-standard format. Asecond possibility is that the record size is greater than 65,534 bytes. If this is thecase, default the block size on the mount, and not use the NOBLOCK option in the **x/zvopen** call, or you will get this error.
  User action: Verify that block size you gave on the MOUNT command is a multiple number of records. The record size for a band-sequential (BSQ) or band-interleaved by line (BIL) image is NS *(bytes per pixel), and for a band interleaved by pixel (BIP) image is NB * (bytes per pixel). If your record size is greater than 65,534 bytes, then do not specify the blocksize on the MOUNT command, and do not use the NOBLOCK option from **x/zvopen**.

- **NOTOPN Symbolic Name**: FILE_NOT_OPEN
  [VIC2-NOTOPN] File not open; program error
  Explanation: The indicated routine triedto operate on an unopened file. User action: this indicates a program error. See the cognizant programmer. Programmer action: make sure that x/zvopen is called prior to the indicated operation. If so, make sure that the program checks the status of the open. The status checking can be achieved automatically with the OPEN_ACT option in **x/zvopen**.

- **NOTPARMFILE Symbolic Name**: NOT_PARAMETER_FILE
  [VIC2-NOTPARMFILE] File specified in PARMS is not a parameter file
  Explanation: The file given with PARMS must be a parameter file created by another program with **x/zvpopen**, **x/zvpout**, etc. and not an image file.

User action: Check that the filename you gave is correct. If it is, then consult the cognizant programmer for the program that created the file. Programmer action: Parameter files must be created using **x/zvpopen**, **x/zvpout**, and **x/zvpclose**. Normal image files cannot be used as parameter files.

- **NULLREQ Symbolic Name**: NULL_REQUEST
  [VIC2-NULLREQ] Null request; program error
  Explanation: The NHIST argument passed to the routine **x/zlhinfo** is either zero or negative. No information is being requested. User action: The value of the argument passed to the indicated routine must be checked. Please consult the cognizant programmer. Programmer action: Check the argument NHIST being passed to the routine **x/zlhinfo**.

- **ODDOPT Symbolic Name**: ODD_NUMBER_OF_ OPTIONALS
  [VIC2-ODDOPT] Unpaired optionals; program error
  Explanation: the number of optional arguments to the indicated routine is odd. Since the optional arguments are given in pairsas KEY WORD, VALUE, an odd number is not allowed. User action: Please consult the cognizant programmer to check that the calling sequence is valid. Programmer action: Check the indicated routine call and verify that the calling sequence is legitimate.

- **OPNINP Symbolic Name**:UNABLE_TO_OPEN_PRIMARY_ INPUT
  [VIC2-OPNINP] Unable to open primary input; check filespecification
  Explanation: In the course of opening a file other that the primary input, an attempt was made to open the primary input to get label/control information which failed. User action: Check the file specification of the primary input. Also, if it is on tape, make sure that it is not on the same tape as the output file which is being processed.

- **ORGMSMTCH Symbolic Name**: ORG_MISMATCH
  [VIC2-ORGMSMTCH] File organization is not that required by this program
  Explanation: The program being used requires a specific file organization (BSQ, BIL, or BIP), and the image file in question is not in that organization. User Action: Check the documentation of the program in question to find out what organization the file should be in, and convert the file to the proper organization.

- **OUTLBL Symbolic Name**:UNABLE_TO_CREATE_OUTPUT_LABEL
  [VIC2-OUTLBL] Unable to create output label; consult system programmer
  Explanation: An error occurred while trying to create the system label for the output file indicated. User action: Consult the system programmer.

- **PARBLKERR Symbolic Name**: PARBLK_ERROR
  [VIC2-PARBLKERR] Internal error in GET_PARM
  Explanation: An invalid type was found in a variable in the parblk passed from TAE.
  User action: Notify the VICAR system programmer.
  Programmer action: The VARIABLE structure for the parameter requested in a call to GET_PARM had an invalid type. It was not V_INTEGER, V_REAL, or V_STRING. Check the parblk passed from TAE for vailidity.

- **PARNOTFND Symbolic Name**: PARAM_NOT_FOUND
  [VIC2-PARNOTFND] A program parameter was not found in the PDF
  Explanation: A call to **x/zvparm** or a related routine requested a parameter that is not in the PDF. User action: this is a program error. Consult the cognizant programmer. Programmer action: A parameter in an **x/zvparm** call (or another parameter routine such as **x/zvp**) was not found in the par block passed to the program from TAE. Check both the **x/zvparm** call and the PDF to make sure the parameter exists in both places and that the spellings match.

- **QUALNOTFND Symbolic Name**: QUAL_NOT_FOUND
  [VIC2-QUALNOTFND] A parameter qualifier was not found in the PDF
  Explanation: A call to **x/zvparm** or a related routine requested a parameter qualifier that is not in the PDF. User action: this is a program error. Consult the cognizant programmer.
  Programmer action: A parameter qualifier in an **x/zvparm** call (or another parameter routine such as

x/zvp) was not found in the par blockpassed to the program from TAE. Check both the **x/zvparm** call and the PDF to make sure the qualifier exists in both places and that the spellings match.

- **RECPARFAIL Symbolic Name**: **x/zvrecpar**_FAIL
  [VIC2-RECPARFAIL] **x/zvrecpar** could not retrieve thepar block from TAE
  Explanation: **x/zvrecpar** was unable to get thepar block from TAE that contains the RECVAR parameters. User action: this is most likely a program error. Make sure your inputs to the program were correct, then notify the cognizant programmer. Programmer action: Check that you executed a RECVAR statement before calling **x/zvrecpar**. If you did, it is most likely a communication problem between TAE and the application. Check your quotas (especially BYTLIM) and if they are ok, then consult the VICAR system programmer.

- **SECDEL Symbolic Name**: CANNOT_DELETE_SECTION
  [VIC2-SECDEL] Unable to free array file, consult system programmer
  Explanation: The area of memory allocated for an array file, called a mapped Section, cannot be deleted. User action: this is an executive error. Consult the cognizant programmer for the VICAR executive.

- **SIZREQ Symbolic Name**: IMAGE_SIZE_REQUIRED
  [VIC2-SIZREQ] Image size required ; re-enter command
  Explanation: **x/zvopen** was asked to process a file with incomplete information, such as creating an output filewhere no input file is present and no size information was supplied by the user. User action: Use size field to work around the problem, and notify the cognizant programmer of the failure of the program to either calculate the size or require its input.

- **STOROPT Symbolic Name**:UNABLE_TO_STORE_ OPTIONAL
  [VIC2-STOROPT] Unable to store optional; consult system programmer
  Explanation: An internal error occurred, preventing the storage of an optional argument to the routine indicated. User action: this error probably indicates an internal error. Please consult the system programmer.

- **STRTELEM Symbolic Name**: BAD_STARTING_ELEMENT
  [VIC2-STRTELEM] Bad starting element; program error
  Explanation: Cannot delete a multi-valued element starting out of range. User action: Check to see if the parameter could have tried to delete a label item. If the problem is still not obvious, consult the cognizant programmer. Programmer action: Verify that the proper values are being passed to the routine **x/zldel**.

- **STRTREC Symbolic Name**: START_REC_ERROR
  [VIC2-STRTREC] Bad starting record for read or write operation; program error.
  Explanation: The starting record for a read or write operation, given by the LINE, BAND, or SAMP optional arguments, is only partially specified. This conditi on a rises whenan image file has a third dimension which is greater than one unit in size (for example, a band-sequential image with more than one band), and the starting value in either thesecond or third dimension was defaulted (for example, you have a band-sequential image and you specify BAND but not LINE). The error condition was raised because the record being requested is ambiguous. User Action: this is a program error. Notify the cognizant programmer. Programmer Action: Either completely specify the record in the call to **x/zvread** or **x/zvwrit**(for band-sequential images, giveboth LINE and BAND), or completely default it (give neither).

- **TAPMETH Symbolic Name**: ILLEGAL_TAPE_METHOD
  [VIC2-TAPMETH] Tape cannot be opened for random access.
  Explanation: An attempt has been made to open a tape file for random access. User action: If the program requires a file, use a disk or memory file. If the problem is not obvious consult the cognizant programmer.

- **TAPOPR Symbolic Name**: ILLEGAL_TAPE_OPERATION
  [VIC2-TAPOPR] Tape cannot be opened for update.
  Explanation: An attempt has been made to open a tape file for update. User action: If the program

requires a file, use a disk ormemory file. If the problem is not obvious consult the cognizant programmer.

- **TAPPOS Symbolic Name**: TAPE_POSITIONING_ERROR
  [VIC2-TAPPOS] Tape positioning error; check drive status
  Explanation: Bad device status. Failed to position tape. User action: Check to see that the tape drive isfunctioning properly.
- **TOOLATE Symbolic Name**: TOO_LATE
  [VIC2-TOOLATE] Attempt to modify tape label after write; program error
  Explanation: Occurs when label processing to a tape file happens after first tape I/O. User action: Program may require file to be on disk. Write desired file to disk. If the problem is still not obvious consult the cognizant programmer.
- **UNDEFOPT Symbolic Name**: UNDEFINED_ OPTIONAL
  [VIC2-UNDEFOPT] Undefined optional argument; program error
  Undefined optional. Explanation: An optional argument given to a VICAR2 routine is not recognized as a valid optional argument. User action: this error is usually a program error and the cognizant programmer should be notified. Programmer action: Check the appropriate document to determine what optional parameters are allowed for the subroutine call in question, ornotify the system programmer.
- **VARREC Symbolic Name**: VARREC_ERROR
  [VIC2-VARREC] COND=VARREC must have NOLABELS, NOBLOCK and tape
  Explanation: If the vARREC option is given to COND, NOLABELS, NOBLOCK, and a tape file must be specified as well. User action: this is a program error. Please consult the cognizant program mer. Programmer action: If VARREC is specified, verify that NOLABELS, NOBLOCK, and a tape file is also specified.
- **WAITFL Symbolic Name**: IO_WAIT_FAIL
  [VIC2-WAITFL] I/O wait fail; consult system programmer
  Explanation: I/O wait fail. A wait for asynchronous I/O returned a bad status. User action: this is an executive error and the system programmer should be notified.
- **x/zvcmderr Symbolic Name**: **x/zvcommand**_ERROR
  [VIC2-XVCMDERR] Internal error in **x/zvcommand**
  Explanation: An internal error occurred in **x/zvcommand** having to do withmessage-passing to the TAE host. Either the PARBLK could not be built, or it could not besent or received. User action: Notify the VICAR system programmer.
- **x/zvcmdfail SymbolicName**: **x/zvcommand**_FAIL
  [VIC2-XVCMDFAIL] The command submitted via **x/zvcommand** had an error
  Explanation: The command submitted with a call to **x/zvcommand** returned an error status.
  User action: this is most likely a program error. Make sure your inputs to the program were correct, then notify the cognizant programmer. Programmer action:
  Check the command submitted with **x/zvcommand**. Only intrinsic commands and procedures using intrinsic commands may be used with **x/zvcommand**, i.e. no processes or DCL commands. The failure may be a syntax error, an error in the parameters, or an execution error. The commandshould have printed an error message; use this to find the problem.

11. Appendix C: Deprecated and Obsolete Subroutines

11.1.1 qprint/zqprint—(Obsolete) Print a message to the terminal

11.1.2 vic1lab—(Obsolete) Return IBM VICAR72 byte labels in a buffer x/zvpblk—Return the address of the parameter block. FOR SPECIAL APPLICATIONS ONLY.

The following twelve subroutines are listed only for reference. They should not be used in new programs.

The PARMS files created by **x\zvpopen**, **x\zvpout**, and **x\zvpclose** should be avoided if possible. PARMS files are used to communicate parameters between programs (the receiver has a PARMS parameter to receive the file). They work and are portable, but the internal implementation is necessarily non-standard. Other communication methods, such as TCL variables or IBIS interface files, are recommended instead.

11.1 qprint/zqprint—(Obsolete) Print a message to the terminal

```
call qprint(message, nbytes)
zqprint(message, nbytes)
```

Obsolete. Use **x/zvmessage**. **qprint/zqprint** will print a message to the standard output, with some carriage control. If the program is run interactively, then the output will appear on the user terminal, and any session logs which are active. Otherwise, it will appear in the job log.

For new VICAR software, **x/zvmessage** is to be preferred to **qprint/zqprint** because it uses a standard message format, allowing the user to build help for error messages, and it works both with and without the VICAR supervisor. QPRINT will function only within the VICAR supervisor.

qprint/zqprint uses the first character in the MESSAGE buffer for FORTRAN-style carriage control. Read the description of the MESSAGE argument carefully to understand how this first character is used.

qprint/zqprint calls the subroutine PRINT with editing turned on, so that non-printable characters are replaced with the character "|".

Arguments:

- **MESSAGE**: input, string, maximum length 132
  MESSAGE contains the message to be given to the user. It may be passed by descriptor (FORTRAN character type) or reference (C char array). If the string is passed by descriptor or is a NULL terminated C-type string, the NBYTES argument is optional.
  The first character in MESSAGE is used to perform a limited FORTRAN style carriage control, controlling output as follows. If the leading character is a
  - 0, then a blank line is generated before the message is printed.
  - 1, then a form-feed is generated if the job is not interactive, and a blank line is inserted if the job is interactive.
  - blank, then normal output starts on the next line.

  If the leading character is one of the three above, then after the carriage control is performed, printing starts with the second character of the buffer. Otherwise, printing starts with the first character of the buffer.

- **NBYTES**: input, integer
  NBYTES is the number of bytes to be printed. If MESSAGE is of type CHARACTER, then the smaller of the NBYTES and the declared length of the string is used. Likewise, if a NULL (binary 0) character is encountered before NBYTES characters are printed, printing terminates at the NULL. If the string is not NULL terminated or a FORTRAN CHARACTER type, then NBYTES is required.

11.2 vic1lab—(Obsolete) Return IBM VICAR72 byte labels in a buffer x/zvpblk—Return the address of the parameter block. FOR SPECIAL APPLICATIONS ONLY.

```
call xvpblk(parb)
status = zvpblk(parb)
```

Not for general use.

The routine **xvpblk** is not portable and should not be used. It has been moved to the P2 library. It is retained only for backwards compatibility. The reason it is not portable is the same as the reason array I/O is not portable —the lack of pointers in FORTRAN. The C routine, **zvpblk,** is portable and can be used. It could be used in a FORTRAN program in a manner similar to that used for array I/O.

x/zvpblk is not needed by most VICAR applications. It is used in the MAIN portion of the application program to obtain the address of the parameter block containing all of the parameters received from the supervisor portion of the executive. The parameter block is a special structure, and should never be looked at directly by the application. Use **x/zvparm** to fetch values out of the parameter block.

Arguments:

- **PARB**: output, pointer to integer
  The address of the parameter block.

11.3 x/zlgetlabel—(Obsolete) Read labels into local memory

```
call xlgetlabel (unit,buf,bufsize,status)
status = zvlgetlabel (unit,buf,bufsize,status)
```

Replace with explicit calls to the other label routines to retrieve only the labels of interest. Reading the entire label buffer and searching through it for a key word (the common usage of **xlgetlabel**) is prone to error as the given key word might exist as part of some other, unrelated label. It is permissible (if the label contents are set up this way) to read a set of labels into a buffer and search that buffer, but do not use **xlgetlabel** for this.

x/zlgetlabel returns the entire VICAR label in a single buffer as it exists in the file, except that the LBLSIZE key word for the part of the label which exists at the end of the file (if the label is in two parts) is not written into the buffer.

The BUFSIZE argument puts a ceiling on the number of bytes returned. That is, the actual number of bytes written into the buffer is the smaller of the entire label size and BUFSIZE. If BUFSIZE is zero on input, then nothing is copied into the buffer, but the en tire label size will be return ed in BUFSIZE (instead of 0).

The user can thus use a BUFSIZE of zero to find out how much space is needed before calling to get the actual label.

Arguments:

- **UNIT**: input, integer
  Unit of associated file. A valid VICAR unit number (returned from **x/zvunit**) of an open file.
- **BUF**: output, byte array

Buffer to contain labels. The buffer in which the VICAR label is to be returned. Should be large enough to hold the entire label, although BUFSIZE may be used to inhibit too large a transfer. Type HELP ARGUMENTS BUFSIZE for further information.

- **BUFSIZ**: input/output, integer
  The size of the internal buffer being used for the I/O operations. If the file is a tape, BUFSIZ will be equivalent to the block size of the tape in bytes.
  On input, BUFSIZE specifies the maximum number of bytes to be returned in the buffer. On output, it gives the actual number of bytes returned.
  If BUFSIZE is zero on input, then nothing is copied into BUFFER, and the total size of the label is returned (instead of zero).

- **STATUS**: output, integer
  Error status code. A value of one indicates success. The error codes are listed in 10 Appendix B: Error Messages (page 107).

11.4 x/zvend—(Do Not Use) Terminate processing

```
call xvend(status)
zvend(status)
```

Use **abend/zabend** instead. **x/zvend** is used to terminate a program. It may be called from any location in the program, making it possible to terminate processing from subroutines, similar to the FORTRAN STOP statement.

Use of **x/zvend** should normally be avoided, for the following reasons:

- When processing terminates because of an unrecoverable error, **abend/zabend** should be used, because **abend/zabend** has certain checks which **x/zvend** does not supply, and **abend/zabend** prints out a standard message which is familiar to the users.

- When terminating processing normally, it is a better practice to return control systematically back to the main portion of the program (in our case to subroutine MAIN44) in order to make the software more easily maintainable.

A FORTRAN “STOP” statement or a C ”exit” call should never be used in a VICAR program.

Arguments:

- **STATUS**: input, integer
  The status with which processing should be terminated. Only two values are allowed: a status of 1 means successful completion, while a status of 0 means abnormal termination. The status is passed back up to the supervisor portion of the executive where it may be entered into logs, etc.

11.5 x/zvpclose—Close parameter data set NOT RECOMMENDED

```
call xvpclose(status)
status =zvpclose()
```

x/zvpclose gets the unit number of the parameter data set which is currently open and closes it with a call to **x/zvclose**.

Arguments:

- **STATUS**: output, integer
  VICAR status indicator. See 10 Appendix B: Error Messages (page 128) for values.

11.6 x/zvpopen—Open a parameter data set for output. NOT RECOMMENDED

```
call xvpopen( status, npar, max_len, filenam,error_act, unit )
status = zvpopen( npar, max_len, filenam, error_act,unit )
```

x/zvpopen opens a new parameter data set for output. The file created is a labeled VICAR file of type PARM, with fixed length 512 byte records.

x/zvpopen uses **x/zvopen** to open the file. The file unit number is maintained internally for reference by **x/zvpout** and **x/zvpclose**. Only one parameter data set may be opened for writing at a time.

Arguments:

- **STATUS**: output, integer
  VICAR return status. Possible values are listed in Appendix B: Error Messages (page 128).
- **NPAR**: input, integer
  this parameter is ignored, and is kept only to maintain compatibility with old programs.
- **MAX_LEN:** input, integer
  this parameter is ignored, and is kept only to maintain compatibility with old programs.
- **FILENAM**: input, string
  The name of the parameter file to be opened.
- **ERROR_ACT:** input, string
  The action to be taken if an error occurs either in the opening of the file or in a subsequent call to **x/zvpout** or **x/zvpclose**. The string is a combination of any of three letters: A, S, and U. The appearance of one of these letters in the string causes the following action to take place in the event of an error:
  - **A**: Abort the program
  - **S**: Issue a system error message
  - **U**: Issue a user message

  If ERROR_ACT is not given, the error action will default to the action specified by **x/zveaction**.
- **UNIT**: output, integer
  If given, UNIT will contain the VICAR I/O unit number being used for the parameter data set. It may then be used in conjunction with other VICAR I/O routines for operations such as adding label information (**x/zladd**).
  Non-label I/O (i.e., **x/zvread** or **x/zvwrit**) should not be performed on a parameter data set, as it is possible to destroy the data contained in it.

11.7 x/zvpout—Write parameter to parameter file. NOT RECOMMENDED.

```
call xvpout( status, name, value, format, count, length )
status = zvpout( name, value, format, count, length )
```

x/zvpout takes a parameter which exists as a valid TAE parameter in some other program and writes it to a parameter file for passing to that program. The information given is formatted and written out to the parameter file, which has already been opened with a call to **x/zvpopen**.

Arguments:

- **STATUS**: output, integer
  VICAR return status indicator. Possible values are listed in Appendix B: Error Messages (page 128).
- **NAME**: input, string
  Name of the parameter in the PDF of the receiving program.
- **VALUE**: input, any type
  value of parameter to be written out. The type of argument used for VALUE depends on the value of FORMAT given. The two *must* match.

- **FORMAT**: input, string
  Data format of VALUE. Valid values are STRING, INT, REAL, REAL8.
- **COUNT**: input, integer
  The actual count of the parameter. This value may exceed the allowable count in the receiving PDF.
- **LENGTH**: input, integer
  zvpout only. For string parameters, the maximum string length in bytes that occurs in the parameter value vector. For the other formats, this argument is ignored. Since the string length can be obtained from the FORTRAN string itself, it is not needed for the FORTRAN **xvpout** call.

11.8 x/zvsfile—Skip files on a tape. USE x/zvadd INSTEAD.

```
call xvsfile(unit, number)
status =zvsfile(unit, number)
```

This routine is not needed. Call **x/zvadd** with the U_FILE optional instead, which is completely equivalent.

x/zvsfile will skip to the file number specified on a tape. If the file number is 0, it skips the tape to the next file.

x/zvsfile requires that the tape file be closed, but it still requires a unit number. If the file has not yet been opened, the unit number can be obtained from **x/zvunit**. If the file has been opened, it must be closed (with **x/zvclose**) before **x/zvsfile** is called, and re-opened afterwards. The unit number obtained from the first call to **x/zvunit** should be used.

x/zvsfile sets the U_FILE entry in the control block for the specified unit number. The same effect may be achieved by passing the U_FILE optional to **x/zvopen** directly.

Arguments:

- **UNIT**: input, integer
  The unit number from **x/zvunit** of the tape. The file may not be open.
- **NUMBER**: input, integer
  The file number. The same rules apply for NUMBER as for the U_FILE optional to **x/zvopen** or **x/zvadd**. If NUMBER=0, the tape is advanced to the next file. If NUMBER > 0, the tape is moved to the absolute file specified by NUMBER (the first file is file number 1). NUMBER may not be less than zero.

11.9 x/zvsptr—String parameter processing subroutine.

```
call xvsptr( val, count, ptr, len )
status = zvsptr( val, count, ptr, len )
```

Deprecated. The parameter processing routines (**xvparm** *et al*) will, in the absence of string length information, return multi valued string arrays in a packed format which must be unpacked using **x/zvsptr**. This packed format (and **x/zvsptr**) should not be used. Provide the parameter processing routine with a CHARACTER*n array in FORTRAN, or a two-dimensional array of characters with a non-zero LENGTH parameter in C, instead. A normal array of strings will then be returned, and **x/zvsptr** will not be needed.

x/zvsptr returns pointers to multiple strings returned by **x/zvparm**. This function is needed when a multi-valued string parameter has been declared to be a BYTE array or CHARACTER (non-array) constant in the application program.

On return from **x/zvsptr**, PTR(I) contains the sequence number of the first byte of the Ith string. (i.e., PTR(1) should always be 1.)

If specified, LEN(I) will contain the length of the Ith string. The indices in PTR are FORTRAN type; that is, the first element is 1.

Arguments:

- **VAL**: input, string
  The array of string values returned from **x/zvparm**.
- **COUNT**: input, integer
  COUNT is the number of strings to search for.
- **PTR**: output, integer array, size count
  PTR is an array in which to return the indices to the strings found. The indices are FORTRAN type; i.e., the first element is element 1.
- **LEN**: output, integer array, size count
  LEN is an array in which to return the lengths of the strings found.

11.10 x/zvtpinfo—Return tape drive information

```
call xvtpinfo(name,device,file,rec)
status= zvtpinfo(name,device,file,rec)
```

x/zvtpinfo is not for use by normal VICAR applications. It is used by certain specialized applications which need to access the tape drive directly without using the VICAR I/O.

Given the symbolic name of the tape drive from the VICAR MOUNT command, **x/zvtpinfo** will return the physical device name and the current position on the tape.

Arguments:

- **NAME**: input, string
  NAME is the symbolic name which was provided to VICAR when the MOUNT or ALLOC command was issued (the NAME parameter).
- **DEVICE**: output, string
  DEVICE is the physical name of the tape drive as supplied to the MOUNT or ALLOC command via the DEVICE parameter. This name may be used in calls to host routines (such as VMS system services) to manipulate the drive.
- **FILE**: output, integer
  FILE is the current file position on the tape, as stored in the VICAR global $TFILE. If **x/zvtpinfo** is used with non-VICAR I/O to manipulate the drive, after the program is done **x/zvtpset** should be called to ensure that the file number and record number are accurate.
- **REC**: output, integer
  REC is the current record number of the current file on the tape, as stored in the VICAR global $TREC. If **x/zvtpinfo** is used with non-VICAR I/O to manipulate the drive, after the program is finished with its task **x/zvtpset** should be called to ensure that the file number and record number are accurate.

11.11 x/zvtpmode—Indicate whether an image file is on tape

```
call xvtpmode(unit,ind)
status =zvtpmode(unit,ind)
```

x/zvtpmode is used to indicate to the calling program whether or not a given unit number is actually on tape. It should be used by programs which absolutely require a file to be on tape to verify that it is. Normally the device a file resides on should not matter.

The given unit must have been opened with a call to **x/zvopen** before **x/zvtpmode** may be called.

Arguments:

- **UNIT**: input, integer
  The VICAR unit number of the file. The unit must have already been opened with a call to

x/zvopen.

- **IND**: output, integer
  If the unit is not a tape, IND will be set to zero, or FALSE. If the output is a tape, then IND should indicate the current density of the drive (800, 1600, or 6250). If an error occurs determining the drive density, then IND will be set to one.

11.12 x/zvtpset—Set tape drive position globals

```
call xvtpset(name,file,rec)
status =zvtpset(name,file,rec)
```

x/zvtpset is a special purpose routine which should not be used by normal VICAR applications. It is intended to be used ONLY as a companion to the non-standard routine **x/zvtpinfo**.

x/zvtpset will save the tape position given by the FILE and REC arguments for the drive NAME in the VICAR supervisor global variables $TFILE and $TREC so that the next program to execute will know the proper tape position.

Arguments:

- **NAME**: input, string
  NAME is the symbolic name which was used in the VICAR MOUNT command to mount the tape. It is the same as the NAME parameter for **x/zvtpinfo**.
- **FILE**: input, integer
  FILE is the current file number of the tape (the first file is file 1). If the true file number is not passed in for FILE, then the next VICAR program will most likely read the wrong file on the tape.
- **REC**: input, integer
  REC is the current record number in the current file on the tape (the first record is record 1).

12. Apprendix D: Unavailable Optional Arguments

There are a number of "unavailable" optional arguments. These are in the optional argument table, but the validation routine returns a NOT_IMPLEMENTED error. They are:

- **NLINES**: input, integer
  Indicates the number of lines to be accessed in a single operation. Defaults to 1.
- **SLICE1**: input, integer
  Indicates the starting pixel in the first dimension of the image. Thus this argument may refer to the LINE, SAMP, or BAND dimension depending on the file organization. It defaults to a starting value of 1and increments per read by 1. (NOT YET AVAILABLE)
- **SLICE2**: input, integer
  As in SLICE1 but for the second file dimension. It defaults to 1 and remains at 1 unless explicitly changed. (NOT YET AVAILABLE)
- **SLICE3**: input, integer
  As in SLICE2 but for the third file dimension. (NOT YET AVAILABLE)
- **NSLICE1**: input, integer
  Indicates the number of pixels to be accessed in the first dimension of the file. Defaults to the length of the first file dimension. (NOT YET AVAILABLE)
- **NSLICE2**: input, integer
  As in NSLICE1 except for the second file dimension. Defaults, however, to 1. (NOT YET AVAILABLE)
- **NSLICE3**: input, integer

As in NSLICE2 except for the third file dimension. (NOT YET AVAILABLE)

- **U_N4**: input, integer
  Indicate that the Executive is to ensure that the output file will have its fourth dimension specified by this value This argument is useful for the program which does not care about the file organization, and simply wants to deal with records and pixels.

- **N4**: output, integer
  The number of pixels in the fourth dimension of the image. The value returned refers to the physical dimensions of the file and is independent of file organization.

See the table in $V2TOP/rtl/source/global.c for details.

There are more of these arguments that were (are?) planned for but were never implemented. The SLICE things should give you access to dimensions 1,2,3,(4) regardless of the order (BSQ/BIL/BIP). Right now, you have to jump through some hoops for order-independent reading. And there were apparently plans to have 4-dimensional files, but they were never realized. NLINES was for multi-line reads (e.g. read a tile), which would be quite useful but has not been implemented either.

13. Appendix E: About This Document

This manual combines material from two previous manuals: "VICAR Run-Time Library Reference Manual" JPL D-4311, 1988 and "VICAR Porting Guide" JPL D-9395, 1994, http://www-mipl.jpl.nasa.gov/portguide/portguide.html. Both were written by Robert G. Deen, Robert.G.Deen@jpl.nasa.gov.

13.1 Document Source

This document was written in the Microsoft Word program. Any changes or additions to it must be made in the original Word document. This is available in two versions: Word native (binary): http://www-mipl.jpl.nasa.gov/RTL/RTL_Manual.wrd and Microsoft RTF (Rich Text Format): http://www-mipl.jpl.nasa.gov/RTL/RTL_Manual.rtf. RTF documents are ASCII text with embedded formatting commands, and can be imported by many word processing programs.

An Adobe PDF version of this document at: http://www-mipl.jpl.nasa.gov/RTL/RTL_Manual.pdf is available for easy printing or viewing using the Adobe Acrobat Reader from: http://www.adobe.com/prodindex/acrobat/readstep.html.

13.2 Generating HTML Version

The HTML version of this manual is generated automatically from the RTF version by the RTFtoHTML filter program, available from: http://www.sunpack.com/RTF/. You must have a modified version of the html-trn parameter file: http://rushmore.JPL.NASA.GOV/RTL/html-trn in the directory containing the RTFtoHTML program.

Use this command line to generate the HTML version of the manual:

```
./rtftohtml -x -c -h2 -T "VICAR RTL Manual" RTL_Manual_Draft.rtf
```

13.3 Changing or Adding to this Document

Any editing done in this document may result in page numbers or section numbers changing. These changes are made automatically only if you use cross-references. If you don't undertand how to use cross-references, please consult the manual or the online help.

The HTML generation step can only be successful if you use one of a limited number of styles RTFtoHTML understands.The RTFtoHTML manual: http://www.sunpack.com/RTF/guide.htm has complete information.

Since page number references are not useful in HTML, all references of the form: (page 233)

must be formatted with the small caps attribute. The html-trn file is modified so that RTFtoHTML will not translate any text in small caps.

13.3.1 Styles used in this Document

Below is a list of styles used in this manual. If you use any other styles, please consult the RTFtoHTML manual: http://www.sunpack.com/RTF/guide.htm and examine the html-trn parameter file to make sure the additional styles will not break the HTML generation process.

- Heading 1 —top level heading
- Heading 2 —second level heading
- Heading 3 —third level heading
- Normal —text in Times font without first-line indent
- Normal Indent —text in Times font with first-line indent, no indent in HTML
- bullet list —first level item list, not numbered
- bullet list 1 —second level item list, not numbered
- pre —preformatted text in Courier font, used for code or computer file names
- glossary —acronym or definition lists
- Caption —table caption

13.3.2 Formatting Hints and Kinks

The html-trn is modified for HTML generation to:

- Discard any text that is has the text format “small caps”. This can be found in the Fonts menu item in the Format menu.
- Add extra line breaks (
) after paragraphs and list items.