Printed September 6, 1994

Multimission Image Processing Laboratory

VICAR Porting Guide
Final Version

B. Deen

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL D–9395

Contents

List of Tables

1 Introduction

This document describes in detail how to port a VICAR application or subroutine to the Unix operating system. VICAR (Video Information Communication And Retrieval) is the image processing system developed and used at JPL's Multimission Image Processing Laboratory (MIPL), that supports both flight project imaging and general instrument data processing tasks.

The VICAR system dates back to the 1960's, when it ran on IBM mainframe computers. In the early 1980's, the system was converted from IBM to run on a DEC VAX/VMS system, and IBM support was dropped. The resulting VMS system is in use at the present time. Now, VICAR is being ported to run on a variety of Unix platforms. This port differs from the earlier conversion in that support for the VMS system is being maintained. The portable VICAR system will run under both VMS and Unix.

1.1 About This Guide

The intended audience for this document is software developers who are involved in porting a VICAR application or in writing a new portable application. It is assumed that the reader is familiar with programming in the current VMS-only VICAR system.

This document discusses in detail the changes in the VICAR Run-Time Library (abbreviated RTL in the rest of the document) that have been made in order to make it portable. This document is not a complete reference for the RTL. It is a supplement to the *VICAR Run-Time Library Reference Manual*. Where the two documents conflict, this guide takes precedence. Features of the RTL that are not described in this guide are the same as documented in the Reference Manual.

This document is broken up into several major parts. The first section describes some of the problems involved in writing portable VICAR code. The next two sections describe the changes in the RTL. The next six sections describe writing portable C, Fortran, and TCL code in the VICAR context. The next three sections describe putting all the pieces together to make a deliverable VICAR program or subroutine, the directory structure, and examples. The final two sections contain summaries of the major portability rules and all RTL calling sequences.

If you as a programmer doing a port come across any portability issues that aren't discussed here, or any problems with what is discussed, please bring it to the attention of the VICAR system programmer (the author of this guide) immediately. It will be incorporated into the next release of this document.

1.2 What is a Port

The emphasis in the whole VICAR porting activity is the word *portable*. All VICAR code, including programs, procedures, and subroutines, must be able to run on many different computers and operating systems. This activity is not a *conversion*, which implies the old version is being thrown out. This happened when VICAR was converted from the IBM to VMS. It is truly a *port*, meaning that VICAR will run on the old VMS system as well as Unix systems.

The portable RTL has been set up to allow the current, unportable, VMS code to continue to run on VMS. This helps to smooth the transition between a VMS-specific system and a portable system. However, you must be careful to maintain the distinction between ported and unported code. After a program is ported, it still runs on VMS, as before, but it will look quite different internally.

MIPL will only officially support VICAR on a few computer platforms. All VICAR software must run on all of these platforms (specific waivers may be granted for special hardware-dependent code). These platforms depend to a large extent on the hardware that is present at MIPL. The VAX running VMS is obviously one of the supported platforms. A DEC Alpha running VMS is also now officially supported. Currently supported Unix platforms are Sun SPARC (SunOS and Solaris 2), Silicon Graphics, and Hewlett-Packard 9000/700 series computers (note: the HP may be removed from the supported list due to funding constraints). VICAR code is only required to run on the machines MIPL officially supports. However, whenever possible, keep in mind general portability rules. It is possible that MIPL may add more supported machine types in the future. The more portably that application code is written, the easier the job of adding a machine will be.

It is important to realize that different computers run slightly different versions of Unix. It is not enough to say your program is portable if it runs on the VMS system and one Unix system. It must run on all the versions of Unix that MIPL supports. The RTL is designed to run on many more machines than will likely be supported, as you can tell from the HOST listings in Table 6. This allows parts of VICAR to be tested on machines that are not fully supported. Applications do not need to support all of these machines, but they should support as many as are practical.

1.3 Reasons for Porting

There are several reasons for porting VICAR to Unix. The most important reason is that Unix is an industry standard operating system, unlike VMS. A wide variety of computer vendors support Unix. Although each vendor's flavor of Unix is slightly different, they are fundamentally similar. The available hardware for Unix systems is typically much faster and cheaper than VAX hardware. So, the port removes the reliance of VICAR on one computer vendor, and allows for a wide range of options. Not only is the computer itself cheaper, the various peripherals and display devices are often cheaper as well.

Another reason for performing the port is that many scientists and experimenters are moving towards Unix workstations at their facilities. In order for them to use VICAR at their sites, it must run on their systems.

The Posix standard for operating systems closely mirrors many of the Unix system calls. Porting VICAR to Unix makes it possible to run on the Posix standard eventually. While VMS will also be Posix-compliant in the future, the system calls to do so will be the Unix-style system calls.

Finally, porting VICAR to Unix allows an opportunity to clean up and modernize the VICAR system. While most of the applications will be ported straight over (without any significant new features), the executive and system-level VICAR code has been significantly updated. Performing the port will ease the transition into a more modern, window-oriented user interface.

1.4 Philosophy of Port

When porting an application or subroutine, keep portability uppermost in your mind. In some cases, this will mean sacrificing some performance to achieve portable code. As long as the performance sacrifice isn't too great, this is acceptable. CPUs are getting faster and faster all the time. A typical Unix workstation has several times the CPU performance of the VAX 8650s that MIPL currently uses. Some pretty big performance hits could be taken before they would be noticed.

On the other hand, in the near term, MIPL must still use the VAX 8650s. Since code you port goes immediately into the development system when you deliver it, and eventually becomes operational, it will be used on the same hardware as it is currently. Slowdowns here will be noticed. Some VICAR subsystems have performance requirements; these must continue to be met. This may be modified somewhat based on circumstances, but try to keep performance in mind. If you cannot bring performance up to the required level, then you might need to consider two separate versions, one for VMS only and one that's portable. This is highly discouraged, as it makes maintenance twice as hard, but it may be an option in some cases. Eventually, when MIPL gets faster VMS machines and more use is made of Unix machines, the performance hit of the portable version will be acceptable and the VMS-only version will be removed. Note, however, that even if you do create a VMS-only version of the program, it still must be able to read files written on foreign machines, as all programs must. Therefore, you cannot always just use the current VICAR code as the VMS-specific version.

A secondary goal of the port is to improve the maintainability of VICAR programs, including the cleaning out of the SUBLIB subroutine library. A little time spent making a program more maintainable will pay off many times over in reduced maintenance costs in the future. Don't just throw away old code gratuitously. There is much to be said for keeping code that works and has been tested over 15 years. However, if a genuine opportunity arises to improve the code, please do so.

There are many routines in the current SUBLIB that are not really needed, or that almost duplicate other routines. For this reason, a wholesale conversion of all SUBLIB routines is not being performed. Rather, subroutines will be ported as they are needed during the conversion of application programs. That way, SUBLIB will get cleaned out in a relatively painless way.

1.5 TBD List

There are some issues related to the porting activity that have not yet been fully resolved. These TBD items are listed here for reference. This list covers unresolved issues affecting program code, as distinct from minor features that have been decided upon but not yet implemented (such as property label instances). It is expected that these issues will be resolved in the near future, and supplements to this guide will be published.

- Use of C++ in VICAR
- Documentation and help files in HTML format

1.6 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)

SPARC Scalable Processor ARChitecture

SPICE Spacecraft, Planet, Instrument, C-kernel, Event

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

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

2 Portability Constraints

This section attempts to explain some of the reasons behind the new VICAR portability rules. While the rest of this document concentrates on the *what* and *how*, this section goes into the *why* behind the major design decisions. Most of the changes in VICAR and the RTL are a direct consequence of portability constraints.

2.1 Variable Arguments

On most machines, there is no way for a subroutine to tell how many arguments are actually passed into it at run time. For that reason, routines that have optional arguments are not allowed. Optional arguments are quite common in VMS VICAR because the VAX (and Alpha) are two of the few machines that pass the number of arguments to a subroutine. These subroutines and the programs that call them will all have to be changed. The preferred solution is to give all arguments on every call to a subroutine. An alternate solution is to have several subroutine names, each with a different number of arguments. That should only be done in very unusual circumstances, however.

There is a standard way to do variable arguments in C and that is to use the <varargs> interface (see any C reference for details). That is the way `printf()` is done, for example. The VICAR Run-Time Library uses this mechanism to handle the "keyword","value" style of optional arguments. The key to <varargs> is that the subroutine must be able to tell how many and what type of arguments are passed in by looking at the previous arguments. For example, `printf()` knows what arguments to expect by examining the format string. The RTL knows what type of argument to expect by the keyword, but it has to know how many arguments to expect. This is handled by adding a terminator as the last keyword in the argument list. See Section 3, RTL Calling Conventions, for details. VICAR application code may use the <varargs> interface if absolutely required; however, it should generally be avoided.

There is no portable way at all to do variable arguments in Fortran. Fortran can call <varargs> routines (such as the RTL), but routines that accept variable arguments cannot be written in Fortran. In addition, code should not be written that accepts variable arguments from a Fortran routine. The RTL does this, but it uses some special tricks that are difficult to port. VICAR application code should not use these tricks.

2.2 Data Types

Different machines have different ways of representing data. VMS machines represent integers in a 2's complement format with the low-order byte first ("little endian"). Most Unix machines represent integers in a 2's complement format with the high-order byte first ("big endian"). In addition, most Unix machines use IEEE floating-point format, while VMS uses its own floating-point format. The DECstation uses IEEE format with the bytes reversed.

All these data representation differences add up to major porting headaches. How does an application read a file that was written on a different machine? How does an application convert between data types? Fortunately, most data format conversions during I/O are handled automatically by the RTL. However, there are several areas to watch out for in application programs. See Section 5, Data Types and Host Representations, for details. The use of EQUIVALENCE statements in Fortran code for type conversion is a particular problem; see Section 7.3, No EQUIVALENCE for Type Conversion, for details.

2.3 Language Differences

Not all compilers are created equal. Typical VICAR Fortran code uses a lot of VMS Fortran extensions that are not in the Fortran 77 standard. Some of these extensions are useful enough and are available widely enough to make them permissible in VICAR programs. Others, however, are not and will have to be changed. See Section 7, Porting Fortran, for details.

With C, the situation is much better, but there are still some porting issues. See Section 6, Porting C, for details.

VICAR fully supports both ANSI and K&R (Kernighan and Ritchie) C, and Fortran. ANSI C is generally preferred over K&R C. For details on using ANSI C with VICAR, see Section 6.5, ANSI C. There is a good chance that C++ will be supported in the future, but that is still TBD.

2.4 Mixed-Language Interface

The interface between routines in Fortran and C is a particular porting problem. The Fortran and C language interfaces are fundamentally different. Fortran strings are especially difficult, since there are now six different ways of passing them among the machines the RTL currently supports. Therefore, any subroutine that accepts character strings must have separate interfaces for Fortran and C (called a bridge routine, or a language binding). In addition, the standard calling sequence even for non-string arguments differs in Fortran (pass by reference) and C (pass by value). Although a Fortran-style calling sequence can be simulated in C (which is the way most VICAR routines currently work), it is unwieldy and difficult to manage. Plus, there is no guarantee that simulating the Fortran interface in C will work across all machine architectures. Therefore, all subroutines must have separate C and Fortran calling sequences, with different names. The name issue will be discussed below. Of course, some subroutines only make sense when called from one language, in which case the alternate language binding would not be needed.

2.5 Subroutine Names

Given that separate Fortran and C-callable subroutines are needed, is there any way of using the same subroutine name for both? Unfortunately, the answer is no. On many Unix machines, such as Suns and DECstations, the Fortran compiler automatically adds an underscore ("`_`") to the end of every external symbol either defined or referenced. This is transparent if you are using only Fortran, but becomes an issue when mixing with C. For example, if your C-callable routine is named `xyz()`, the Fortran-callable routine (in C) must be named `xyz_()` in order for the Fortran routine to call it.

If adding an underscore was universally accepted, then the same routine name could be used for both the Fortran and C interfaces. The Fortran binding would simply append an underscore. However, this is not the case. Several machines, notably VMS and the HP 700 series, do not append an underscore and so use the same name for both Fortran and C. To accomodate these machines, there must be a different name for the Fortran and C versions of subroutines. For example, the RTL uses the convention of starting the name with "`x`" for Fortran (e.g. **xvread**, **xlget**) and starting the name with "`z`" for C (e.g. **zvread**, **zlget**). There has been no firm naming convention established for the VICAR subroutine library. Many subroutines prepend a "`z`" to the name, similar to the RTL. Others use a mixed-case/underscore convention, for example `mpLabelRead` or `IBISColumnWrite` for C and `mp_label_read` or `ibis_column_write` for Fortran. Use whatever seems appropriate when creating a new subroutine.

You may be wondering how to append the underscore. There is a macro provided to do this for you in C, called FTN_NAME in "ftnbridge.h". See Section 9.2, Naming Subroutines, for details.

The naming problem provides another incentive for having separate Fortran and C interfaces in the first place. Imagine a subroutine written in Fortran, named "`xyz`". Since some machines add an underscore, the C compiler will see the routine name as "`xyz_`". In order to call this routine, the application programmer would have to add an underscore after the routine name, which would not be portable, or use a macro, which would be terribly inconvenient. The application programmer would have to know whether a routine was written in Fortran or C to know whether or not to use the macro. If the subroutine was converted to the other language at some point, all the calls in all the applications would have to be changed, which would be a nightmare. To solve this problem, the burden is placed on the subroutine writer. The application programmer merely calls the subroutine; the subroutine is then responsible for taking care of language differences. In this case, there would be a bridge routine named `zyz()`, written in C, that would in turn call the Fortran routine `xyz()` using the FTN_NAME macro after converting the arguments appropriately.

Since a Fortran subroutine called from C must have a separate bridge routine anyway, it makes sense to require *all* subroutines to have a separate bridge routine, with a different name. Besides solving the problems listed above, it allows much more freedom in choosing a calling sequence that makes sense for each language.

3 RTL Calling Conventions

This section describes the calling conventions for the Run-Time Library, where they differ from the old standards as documented in the *VICAR Run-Time Library Reference Manual*.

3.1 Terminator

The most immediately obvious change in the RTL calling sequence is the addition of an argument list terminator for keyword-value routines. There are two basic types of RTL routines: those that take a constant number of arguments, and those that take optional arguments of the form "keyword","value", "keyword","value", etc. The terminator applies *only* to the keyword-value routines. *All* routines that accept keyword-value pairs, whether or not the pairs are used in any particular call, must have the terminator. The terminator is a language-dependent argument that goes at the end of the argument list, where the next keyword 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, ' ')
```

Note that **x/zvclose** requires a terminator, since it allows a (seldom-used) keyword-value pair. Since the routine allows keyword-value pairs, it must have a terminator, even though the keyword-value pairs are not used in this particular call.

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);
```

3.2 No Optional Arguments

Another change in the RTL calling sequence is the elimination of "pure" optional arguments, which are arguments at the end of the list that formerly could be omitted. Examples of this are **xvparm** and **xvmessage**. The *only* optional arguments allowed in the VICAR system are keyword-value style arguments such as the ones in some of the RTL routines. They can be handled in a portable way, but "pure" optional arguments cannot.

In general, all arguments must be given. This will require the addition of arguments to many of the calls to the affected routines. For each argument that now must be included, there is a value to pass in that has the same meaning as the argument not being there, typically 0 or an empty string. See the documentation for the individual routines for details.

The affected routines are listed in Table 1, with the number of arguments that are required in the Fortran and C calling sequences. The minimum number of arguments in the old RTL are listed as well.

Note that for **xvparm** and **xviparm**, the last argument has been removed, while the last two have been removed from **xvpcnt** and **xvipcnt**. The function of the "r8flag" parameter of **xvparm** and **xviparm** has been replaced with the routines **x/zvparmd** and **x/zviparmd**, which return double-precision floating point values. The function of the "def" parameter on **xvpcnt**

Routine	Fortran Arg Count	C Arg Count	Old Minimum Arg Count
qprint/zqprint	2	2	1
x/zvbands	3	3	2
x/zvmessage	2	2	1
x/zvparm	5	6	4*
x/zviparm	5	6	4*
x/zvpcnt	2	2	2*
x/zvipcnt	2	2	2*
x/zvpopen	6	3	4
x/zvpout	5	5	5*
x/zvsize	6	6	4
x/zvsptr	4	4	3

* See "Note" paragraphs under Section 3.2, No Optional Arguments.

Table 1: Argument counts for RTL routines formerly using optional arguments

and **xvipcnt** has been replaced with the "def" parameter on the new routines **x/zvpstat** and **x/zvipstat**, while the "nbytes" parameter is no longer useful. All of these parameters were rarely if ever used in existing code. Note that the "default" parameter semantics should not be used; use the count of the parameter instead (because once set in TAE tutor, a parameter can not be returned to the default state).

Note also that the last argument to **xvpout** (the maximum string length) has been removed. It was never used in existing code, and is not needed for the Fortran call. The length argument is, however, required for **zvpout**.

3.3 Fortran Calling Sequence

The Run-Time Library now 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. They use the same routine names as in the previous RTL.

The Fortran routines are pretty much the same as the old RTL routines. In fact, under VMS, they can be called exactly the same as the old RTL, including being called from C. However, this is *only* for backwards compatibility — portable routines must never call the Fortran entry points from C, and Fortran routines must use all the new calling conventions. Violating these rules will likely result in a program crash on any machine other than a VAX.

3.3.1 Character Strings

Most RTL routines take characters strings as arguments. They appear in parameter names, keywords, 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 now accept character strings *only* in Fortran CHARACTER*n format, which is the standard way of doing strings in Fortran. CHARACTER*n variables and constants have a length associated with them (the "n"), which the RTL uses to find the end

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 2: 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 in Table 2
pixel pointer	N/A

Note that 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 3: Fortran declarations for Run-Time Library arguments

of the string, and to make sure that buffer overflow does not occur on output strings. Byte arrays will not work, since there is no Fortran string length information, thus no standard way for the RTL to find the end of the string. Also, arrays of CHARACTER*1 will not work — the RTL will think the string being passed in is one character long.

This change will be most apparent in the **qprint** and **xvmessage** routines, which are often passed byte arrays in the current code. This will not work. Use the new SUBLIB routines `mvcl` or `mvlc` to convert between byte arrays and CHARACTER*n variables . A better approach (and highly recommended) is to change the program to use CHARACTER*n throughout. Output formatting is actually much easier with CHARACTER*n variables. See Section 7.5, READ & WRITE to Strings, for details.

3.3.2 Fortran Data Types

The standard Fortran definitions for each of the VICAR pixel data types are listed in Table 2. The data types that may be passed in to the RTL, and the Fortran declarations for each type, are listed in Table 3. These RTL data types are referred to in Section 15, Summary of Calling Sequences.

3.4 C Calling Sequence

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

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

3.4.1 Differences from Fortran

There are several differences in the C calling sequence from the Fortran calling sequence. The name change has been discussed already. Also, the use of a 0 or NULL terminator for a keyword-value argument list has been discussed above. There are two major differences that will come up in porting programs, however. They both have to do with making the interface conform to standard C calling conventions. First, most arguments are now passed by value. The old RTL routines expected all arguments to be passed by reference, since that's how Fortran does it. This leads to all sorts of strange constructs like "`&1`" in C, which is not valid on most machines. The rule is: if it's an input value, it's passed by value. If it's an output value, it of course must still be passed by reference (as a pointer). For example, "unit" is a parameter to most of the RTL routines. Instead of calling `zvread(&unit, ...);`, you call `zvread(unit, ...);` (where "unit" is an integer). The routine **zvunit**, however, returns "unit" as an output, so it would still have to be called "`zvunit(&unit, ...);`". Most values in keyword-value pairs are input, so the "`&`"s would go away. On **zvget**, however, most values are output, so they are still passed as pointers. Arrays of items, as well as strings, are still passed by reference as per standard C practice. It sounds rather confusing, but it makes sense once you work with it a little.

There is a pitfall to be aware of regarding **zladd**. Since **zladd** accepts as input the value to which to set the label, you might expect it to be passed by value. However, **zladd** can accept arrays of items, since a label may be multi-valued. Therefore, the "value" parameter to **zladd** is always passed by reference. This can be confusing if only one value is being added. This is not an issue with **zlget**, since "value" is an output from **zlget**. You would expect "value" to be passed by reference in that case.

Second, many routines had a "status" argument, which returned the status of the command. These arguments have been removed. The status is now the return value of the function. So, instead of "`zvread(unit,&status,...);`", it is now "`status = zvread(unit,...);`". If you don't need the status value, it can be ignored entirely, as in "`zvread(unit,...);`".

Also, the handling of arrays of strings as arguments has been improved and made consistent in the C interface. Only a few routines use string arrays as arguments, but the calls to those routines should be examined carefully. These routines are **zladd**, **zlget**, **zlhinfo**, **zvpout**, **zvparm**, **zviparm**, **zvparmd**, and **zviparmd**. See below for the rules on passing string arrays.

3.4.2 C Data Types

The standard C definitions for each of the VICAR pixel data types are listed in Table 4. The data types that may be passed in to the RTL, and the C declarations for each type, are listed in Table 5. These RTL data types are referred to in Section 15, Summary of Calling Sequences.

Special attention should be paid to the passing of string arrays. Since labels and parameters can have more than one value, the routines that deal with them must be able to accept (or

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 in Table 4
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 in Table 4
pointer to string array	char **x

Table 5: C declarations for Run-Time Library arguments

output) arrays of strings. The RTL convention is that all arrays of strings are expected to be two-dimensional arrays of "char". This does not mean an array of pointers to strings; be very careful of that. It does mean 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. Note that the length is the size of the inner dimension; it is not the length of any particular string. In other words, the length includes space for the null terminator at the end of a string. If you know the maximum size of any string is 10 characters, make sure the length is at least 11 to include room for the null terminator.

3.5 Backwards Compatibility with VMS

All RTL routines were designed to be backwards compatible with the old VMS version. So, programs that follow the old RTL interface rules will continue to work on VMS. This includes no need for a terminator, use of optional arguments, calling the Fortran routines from C, and even the inclusion of arguments that have been removed in the new version.

While the old ways still work, they work *only* under VMS. Any program that does not follow the new rules for calling the RTL will not work on any other machine. This may cause problems for people trying to develop portable code under VMS, until they fully understand the new rules, because an invalid program will work under VMS. The way around this is to try the program frequently on a Unix-based machine to make sure you are writing truly portable code.

4 RTL Routine Changes

This section describes the changes that have been made to the RTL in the conversion, as well as a couple of routines that were added previously but not documented yet. A summary of the calling sequences of all RTL routines is presented in Section 15, Summary of Calling Sequences.

4.1 New Optional Keywords

Several new optional keywords have been added to some of the RTL routines. These are listed and described below. They are valid in both the Fortran and C calling sequences.

- 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 Section 5, Data Types and Host Representations, for more information. HOST is available in the routines **x/zvadd**, **x/zvopen**, and **x/zvget**. For output files, the HOST optional to **x/zvopen** or **x/zvadd** unconditionally sets the output's HOST label. For input files, the HOST optional to **x/zvopen** or **x/zvadd** has no effect unless the input file is unlabeled (i.e. the HOST in the input file overrides).

  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 in Table 6. 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 data types, such as Array I/O or CONVERT OFF, you must pay attention to the INTFMT label (via **x/zvget**) and use the **x/zvtrans** representation (for binary labels see BINTFMT below). See Section 5, Data Types and Host Representations, for more information. INTFMT is available in the routines **x/zvadd**, **x/zvopen**, and **x/zvget**. For output files, the INTFMT optional to **x/zvopen** or **x/zvadd** unconditionally sets the output's INTFMT label. For input files, the INTFMT optional to **x/zvopen** or **x/zvadd** has no effect unless the input file is unlabeled (i.e. the INTFMT in the input file overrides).

  The valid values of INTFMT may change as the RTL is ported to new machines. However, the currently valid values are listed in Table 7. In addition, the following values are also accepted:

  NATIVE : The integer format 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 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 data types, such as Array I/O or CONVERT OFF, you must 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 BREALFMT below). See Section 5, Data Types and Host Representations, for more information. REALFMT is available in the routines **x/zvadd**, **x/zvopen**, and **x/zvget**. For output files, the REALFMT optional to **x/zvopen** or **x/zvadd** unconditionally sets the output's REALFMT label. For input files, the REALFMT optional to **x/zvopen** or **x/zvadd** has no effect unless the input file is unlabeled (i.e. the REALFMT in the input file overrides).

  The valid values of REALFMT may change as the RTL is ported to new machines. However, the currently valid values are listed in Table 8. In addition, the following values are also accepted:

  NATIVE : The floating point format 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 BREALFMT 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 BHOST 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 BIN_CVT is OFF (if not present the file's BHOST comes from the primary input, or defaults to VAX-VMS). If BIN_CVT is ON, the BHOST optional is ignored since the file's BHOST gets set to the native host. See Section 5, Data Types and Host Representations, for more information. BHOST is available in the routines **x/zvadd**, **x/zvopen**, and **x/zvget**.

  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 please 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 BIN_CVT is OFF (if not present the file's BINTFMT comes from the primary input, or defaults to the integer format for VAX-VMS). If BIN_CVT is ON, the BINTFMT optional is ignored since the file's BINTFMT gets set to the native host integer format. See Section 5, Data Types and Host Representations, for more information. BINTFMT is available in the routines **x/zvadd**, **x/zvopen**, and **x/zvget**.

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 please change all three. BREALFMT 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 BIN_CVT is OFF (if not present the file's BREALFMT comes from the primary input, or defaults to the floating-point format for VAX-VMS). If BIN_CVT is ON, the BREALFMT optional is ignored since the file's BREALFMT gets set to the native host floating-point format. See Section 5, Data Types and Host Representations, for more information. BREALFMT is available in the routines **x/zvadd**, **x/zvopen**, and **x/zvget**.

  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 data type, 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 BIN_CVT setting). If the optional is not present, the file's BLTYPE comes from the primary input. See Section 5, Data Types and Host Representations, for more information. BLTYPE is available in the routines **x/zvadd**, **x/zvopen**, and **x/zvget**.

  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 Section 5.6.3, Binary Label Type, 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/zvtrans** family of routines. See Section 5, Data Types and Host Representations, for details. CONVERT is available in the routines **x/zvadd** and **x/zvopen**.

- 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 *only* if BIN_CVT is OFF (they are ignored if BIN_CVT is on).

  BIN_CVT is available in the routines **x/zvopen** and **x/zvadd**. BIN_CVT is ignored for input files. See Section 5.6, Dealing with Binary Labels, for more information.

- PROPERTY, string: The name of the property set that the label routine is accessing. PROPERTY is available on **x/zladd**, **x/zldel**, **x/zlget**, and **x/zlinfo**. The PROPERTY optional argument is only valid if the TYPE argument of the above routines is set to "PROPERTY". See Section 4.6, Property Labels, for more information.

Several previously existing optionals for the label routines are now valid in other routines. The new combinations are:

- NRET is now available in **x/zlhinfo**, where it 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 is now available in **x/zlhinfo**, where it specifies the length of each element in the TASKS array. Formerly, TASKS was constrained to be 8 characters and only 8 characters. If a task name was exactly 8 characters long, there would not be a null terminator in the TASKS array. This has been fixed, so now 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). You are *strongly* urged to use a ULEN of at least 9 from C, in order to get all 8 characters with a null terminator.

- ERR_ACT is now available in **x/zlhinfo** and **x/zlninfo**, where it works just like all the other ERR_ACT's.

- ERR_MESS is also now available in **x/zlhinfo** and **x/zlninfo**, where it works just like all the other ERR_MESS's.
- STRLEN is now available in **x/zlninfo**, where it works just like STRLEN does in **x/zlinfo**.
- MOD is now available as an optional argument to **x/zlninfo** as well as **x/zlinfo**, but it is still not implemented.

4.2 New Routines

This section documents the new RTL routines. They are broken down by functional areas.

4.2.1 Parameter Routines

The parameter routines are grouped into two major areas. The first deal with parameters on the command line, and the second deal with parameters from an interactive request (via **x/zvcommand**). The new commands in a nutshell are:

- **x/zvip**: Interactive version of **x/zvp**.
- **x/zviparmd**: Interactive version of **x/zvparmd**.
- **x/zvipone**: Interactive version of **x/zvpone**.
- **x/zvipstat**: Interactive version of **x/zvpstat**.
- **x/zvparmd**: Like **x/zvparm** except floating point values are returned as double precision.
- **x/zvpone**: Return a single value from a multivalued parameter.
- **x/zvpstat**: Get info about a parameter but not its value.

4.2.1.1 x/zvip

```
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).

  Note that 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 multivalued string with **zvip**, it is returned in the **zvsptr** packed format, which is obsolete and should not be used. Under 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.

4.2.1.2 x/zviparmd

```
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 **xviparm**.

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.

4.2.1.3 x/zvipone

```
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 multivalued 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 **xvipone** 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.

4.2.1.4 x/zvipstat

```
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 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".

4.2.1.5 x/zvparmd

```
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 **xvparm**.

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.

4.2.1.6 x/zvpone

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

This routine returns a single value from a multivalued 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 **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.

4.2.1.7 x/zvpstat

```
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".

4.2.2 Translation Routines

The translation routines allow conversion of data between different data types and different host representations. See Section 5, Data Types and Host Representations, 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.

The translation routines are:

- **x/zvtrans**: Translate data types and host representations.
- **x/zvtrans_in**: Set up a translation buffer for input from a (potentially) foreign host.
- **x/zvtrans_in**: Set up a translation buffer for input from the binary labels of a file.
- **x/zvtrans_inu**: Set up a translation buffer for input from a file.
- **x/zvtrans_out**: Set up a translation buffer for output to a (potentially) foreign host.
- **x/zvtrans_set**: Set up a translation buffer for native data type conversion.

4.2.2.1 x/zvtrans

```
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). Any other routines to do translations should be merely syntactic sugar for this routine.

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. Note that 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 to translate. Both SOURCE and DEST must be large enough to hold this many pixels; no checking is done. Note that this is the number of *pixels*, not the number of *bytes*, to translate.

4.2.2.2 x/zvtrans_in

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

Set up a 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 set up, 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 in Table 7. It may also be "NATIVE" or "LOCAL", both of which mean the native host INTFMT. Note that 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 in Table 8. It may also be "NATIVE" or "LOCAL", both of which mean the native host REALFMT. Note that 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.

4.2.2.3 x/zvtrans_inb

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

Set up a 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 set up, 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 data type. 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 BREALFMT. 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.

4.2.2.4 x/zvtrans_inu

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

Set up a translation buffe 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 merely 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 set up, 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.

4.2.2.5 x/zvtrans_out

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

Set up a 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 this translation is most often needed for output to a file.

This routine will be much 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 set up, 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 in Table 7. It may also be "NATIVE" or "LOCAL", both of which mean the native host INTFMT. Note that 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 in Table 8. It may also be "NATIVE" or "LOCAL", both of which mean the native host REALFMT. Note that 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. 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.

4.2.2.6 x/zvtrans_set

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

Set up a 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 set up, 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.

4.2.3 Miscellaneous Routines

The miscellaneous routines don't fit in any general category. They are:

- **x/zlpinfo**: Returns the names of property subsets in the given file.
- **x/zmove**: Move data from one place to another.
- **x/zvcmdout**: Version of **x/zvcommand** that can return values.
- **x/zvfilename**: Expand a filename to a form suitable for `open()`.
- **x/zvhost**: Return values for INTFMT and REALFMT given a host type name.
- **x/zvpixsize**: Return the size of a pixel given the type and host.
- **x/zvpixsizeb**: Return the size of a binary label value given the type and a file.
- **x/zvpixsizeu**: Return the size of a pixel given the type and a file.
- **x/zvselpi**: Selects the file to use as the primary input.

4.2.3.1 x/zlpinfo

```
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 **x/zlhinfo**, 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 **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 of course 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: 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 routine failed for some reason, and the returned values may not be valid.

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.

  Note that 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 **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.

4.2.3.2 x/zmove

```
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 of course available memory. Note that LEN is measured in *bytes*, not *pixels*.

4.2.3.3 x/zvcmdout

```
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.

4.2.3.4 x/zvfilename

```
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 disk filenames are supported. Names specifying tape files or memory files will be treated as if they were disk files, which could provide surprising results. Note that "`*`" 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.

Please note that 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.

4.2.3.5 x/zvhost

```
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 in Table 6. 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 xvmaininc.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 in Table 7.

  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 in Table 8.

  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.

4.2.3.6 x/zvpixsize

```
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 in Table 7. It may also be "NATIVE" or "LOCAL", both of which mean the native host INTFMT. Note that 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 in Table 8. It may also be "NATIVE" or "LOCAL", both of which mean the native host REALFMT. Note that RHOST should be given even if you are dealing with integral data types.

4.2.3.7 x/zvpixsizeb

```
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 may be 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 BREALFMT. The values obtained from the file are used exactly like the **x/zvpixsize** IHOST and RHOST.

4.2.3.8 x/zvpixsizeu

```
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.

4.2.3.9 x/zvselpi

```
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 a couple of things. If you don't give file attributes like size, type, etc. when you open a file, the defaults are taken from the primary input. Also, when you create a new output file, the history and property labels for the new file 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 of course).

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. You must 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.

4.2.4 Fortran String Conversion Routines

These routines allow C-language subroutines to use character strings passed in from a Fortran routine (CHARACTER*n data type). *All* 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 Section 9, Mixing Fortran and C, for details.

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".

- Common Features: Rules and arguments common to all string routines
- **sc2for**: Convert a C string to Fortran (output)
- **sc2for_array**: Convert an array of C strings to Fortran (output)
- **sfor2c**: Convert a Fortran string to C (input)
- **sfor2c_array**: Convert an array of Fortran strings to C (input)
- **sfor2len**: Get the length of a Fortran string (input)
- **sfor2ptr**: Get a pointer to the characters in a Fortran string (input)

4.2.4.1 Common Features

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.

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, and 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"
```

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. See Section 11.1, `vimake`, for details on the imakefile.

FORSTR_PARAM and FORSTR_DEF macros

In most routines (see below for the exception), you must include the macro FORSTR_PARAM in the argument list of the directly called routine, and you must put FORSTR_DEF at the end of the formal parameter declaration list, just before the opening brace of the procedure. FORSTR_DEF should not have a semicolon after it, 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 or FORSTR_DEF macros.

Examples:

```
int constargs(a, s1, s2, b, FORSTR_PARAM)
  int *a, *b;
  char *s1, *s2;
  FORSTR_DEF
{ ...
}

int varargs(va_alist)
  va_dcl
{ ...
}
```

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.

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];
  /* set 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);
}
```

4.2.4.2 sc2for

```
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 the "Common Features" section above.

4.2.4.3 sc2for_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 the "Common Features" section above.

4.2.4.4 sfor2c

```
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 the "Common Features" section above.

4.2.4.5 sfor2c_array

```
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 `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
  NELEMENTS is the number of strings in the array to convert.
- FOR_STRING, ARGPTR, NARGS, ARGNO, STRNO
  See the "Common Features" section above.

4.2.4.6 sfor2len

```
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 the "Common Features" section above.

4.2.4.7 sfor2ptr

```
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 the "Common Features" section above.

4.2.5 System Internal Routines

These three routines should *never* be called by application code. They are called by the "vicmain_c" and "VICMAIN_FOR" include files to initialize the system. They are listed here because technically they are externally visible routines. However, since only the main include files may ever call them, detailed documentation will not be provided.

- **xvzinit**: Do most of the setup for a Fortran main program
- **zvpinit**: Initialize parameters from a PARMS parameter file
- **zv_rtl_init** : Initialize the Run-Time Library

4.3 New Features in Old Routines

This section lists new features and potential incompatibilities of the new RTL that aren't discussed elsewhere or need to be re-emphasized.

- **xvp**

  The COUNT parameter on **zvp**, **xvip**, and **zvip** behaves like you would expect. It returns the number of parameters given, and 0 if it is a null parameter. However, in **xvp**, COUNT behaves differently. 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. See the discussion of COUNT under **x/zvip** above for more details.

- **xvparm** and **xviparm** R8FLAG parameter

  The R8FLAG parameter of **xvparm** and **xviparm** has been removed. To return double precision real values, use **x/zvparmd** or **x/zviparmd**. The R8FLAG parameter does still exist in the VMS version for backward compatibility, however, it is *only* in the VMS version and may disappear in the future.

- Single quotes in label strings

  Strings in the image label are delimited by single quotes (’). In the past, this meant that single quotes could not be included in a label string. With the new RTL, however, this restriction no longer applies. A single quote in an input string (to **x/zladd**) 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. Care should be taken when creating labels with single quotes, as older versions of VICAR will not be able to read the file. This problem should disappear in time as the old versions get replaced.

- **xvpout**

  The maximum string length parameter for **xvpout** has been removed. It was formerly an optional argument, but was not used anywhere in the system. Since the string length can be obtained from the Fortran string itself, it is not needed for the Fortran **xvpout** call. The length parameter is, however, required on the C-language **zvpout** call.

- Label key size expanded

  The maximum size of a label key has been increased from 8 to 32 characters. This will allow more descriptive labels and will facilitate data exchange with other image formats, such as PDS and FITS. Note that 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. Property names may be up to 32 characters long as well.

  You must be careful when using longer label keys, at least at first. An image with a long label key will not be able to be read properly on an older VICAR system. Before using a long label key in an image, make sure that all potential users of the image are using a version of VICAR that handles long label keys.

- Property type added to labels

  The TYPE argument to **x/zladd**, **x/zldel**, **x/zlget**, and **x/zlinfo** now allows the type "PROPERTY" as well as "SYSTEM" and "HISTORY". "PROPERTY" must be specified for the TYPE argument of these routines to access the property labels. See Section 4.6, Property Labels, for more information.

- 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.

- 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.

4.4 Deprecated RTL Functionality

This section lists subroutines and capabilities of the RTL that are deprecated, meaning they should be avoided if possible. Deprecated functionality is still supported and may be used for compatibility with old code, but it should not be used for new code. Features on this list are candidates for becoming obsolete in the future if they are no longer needed.

- Parameter default flag

  Some of the parameter routines return a default flag, which indicates whether the parameter was input by the user or defaulted. This flag should not be used. It is maintained for compatibility reasons, but the parameter count should be used instead.

  The major problem with using the default flag is that there is no way to reset the flag under TAE. If the parameter was entered, and saved in a parameter file, there is no way for the user to reset the flag, even if the default values are re-entered.

  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.

- ULEN default to **zlhinfo**

  The ULEN optional argument to **zlhinfo** should not be defaulted. The default value is 8 for backwards compatibility. However, a value of 8 does not leave room for a C string null terminator, so task names get run together unless special care is taken. Use at least 9 for ULEN to guarantee a null terminator.

- Using non-CHARACTER variables for Fortran strings

  Although it is possible to treat strings in Fortran as BYTE or INTEGER arrays rather than as CHARACTER*n variables, this is highly discouraged. It is non-standard Fortran to use arrays for string manipulation and should be avoided. The RTL will not accept arrays for string arguments (except on the VAX for backwards compatibility). The practice should be avoided in all Fortran code.

- WORD, LONG, and COMPLEX data types

  The data types WORD, LONG, and COMPLEX are recognized by the RTL routines that accept data types, for backwards compatibility. However, they should never be used in new code. Use the standard set BYTE, HALF, FULL, REAL, DOUB, and COMP instead.

- **qprint/zqprint**

  These routines should not be used, as they require numeric string lengths and use Fortran carriage control, both of which are no longer necessary. Use **x/zvmessage** instead.

- **x/zlgetlabel**

  These routines should be replaced 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 keyword (the common usage of **x/zlgetlabel**) is prone to error as the given keyword 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 **x/zlgetlabel** for this.

- **x/zvend**

  These routines should not be called from user code. To abnormally terminate a program, use **abend/zabend**. To exit a program normally, simply return from the `main44` subroutine.

- **xvpblk**

  The routine **xvpblk** is not portable and should not be used. 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. See Section 7.6, Array I/O, for details.

- **x/zvsfile**

  This routine is not needed. Call **x/zvadd** with the U_FILE optional instead, which is completely equivalent.

- **x/zvsptr**

  The parameter processing routines (**x/zvparm** *et al*) will, in the absense of string length information, return multivalued 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.

- PARMS files

  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.

- Direct RTL tape support

  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 somewhat problematical under Unix, so it is discouraged. While this method of access is allowed, the underlying implementation is not portable, and currently only runs on Sun-4s, and only on one brand of tape drive. It is recommended that all processing 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).

- .Z*xx* temporary files

  The VMS VICAR style of specifying a temporary filename was to omit the extension, which then defaulted to .Z*xx*, where *xx* is based on the process ID. This form works only under VMS, so its use should be avoided (especially inside PDFs). The new form is to use a plus (+) before the filename; see Section 4.3, New Features in Old Routines, above, for details.

4.5 Obsolete Routines

The following routines are obsolete and are no longer supported, even under VMS. Some of them haven't been supported for a long time, but they were still documented.

- **make_upper_case** : This routine was never documented but was available externally. It should only be an internal routine, and is not now visible externally. Use appropriate SUBLIB routines to convert strings to upper case.

- **print**: This routine was supposed to print messages to the line printer or operator console, but that was disabled long ago. It only printed messages to the terminal. Replace by using **x/zvmessage**.

- **sfor2c**: While a new routine of the same name exists, there are enough changes in the calling sequence and rules of use to consider the old one to be an obsolete routine. Any calls to **sfor2c** will have to be completely rewritten.

- **vic1lab**: This routine has been moved to SUBLIB instead of the RTL. It has some changes in the calling sequence as well.

- **xvpinit**: This routine has been replaced by **zvpinit**. However, since it should only be called by the main include files and never by applications, this should have no impact.

- **xvprnt**: Used only for debugging, it printed out the contents of the internal file unit table. No longer supported.

- **xvrecpar**: This routine was used in conjunction with **xvcommand** o get outputs from a procedure. Instead of calling the sequence **xvcommand**, **xvrecpar**, now just call the single routine **x/zvcmdout**. Only one application ever used **xvrecpar**, and that has been updated, so there should be no impact for this change.

4.6 Property Labels

Property labels are a new feature in the RTL in delivery 8.0. While they have nothing to do with portability *per se*, they are documented here until such time as the *RTL Reference Manual* can be rewritten.

There are now three major sections of labels: system, history, and property. System and history existed previously, while property labels are new.

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.

Previously, history labels served a dual role: they contained processing history (what tasks were run in what order), but they also contained information about the current state of the image, which has nothing at all to do with history. 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. The map labels describe the map projection the image is in. 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 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 can also be used instead of binary labels for many applications. Property labels do not suffer the same portability problems as binary labels, and they are more readable. They can replace binary headers in most cases. Replacing binary prefixes may be more difficult. Although binary labels are still allowed, the use of property labels instead is encouraged.

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. Please note that the properties and names listed are *only* examples, and do not in any way establish the official names of anything.

```
---- 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)
```

4.6.1 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". You must specify which property the label belongs to via the "PROPERTY" optional argument to these routines. The "HIST" and "INSTANCE"

keywords are 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**.

From the user's point of view, the LABEL program has been extensively modified to recognize and use property labels. See the help for LABEL for details on how to use it.

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" keyword or at the start of the history labels (indicated by a "TASK" keyword).

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. The same is not true for deletion (via the RTL): if you want to delete an entire property you must delete all the elements from it and then delete the "PROPERTY" marker (using "PROPERTY" as the keyword to delete from the property set). The `label-delete` program can do this. It is important to note that this is exactly the way history labels work: tasks are created automatically when programs are run, but to delete a task you must delete all the keywords 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, of course. The copying of property labels can be controlled via **x/zvselpi**.

Property labels are a new feature, and some care must be taken if there is the possibility of using older versions of VICAR that don't support property labels. If you put property labels on an image, then older versions of VICAR will not be able to access the property labels. The image itself and the history label will be accessible, but the property labels will not. There are three problems with using an image with property labels on an old VICAR system. First, a "`label-list 'dump`" will go into an infinite loop if there is more than one property. Other forms of `label` work fine. Second, if you try to modify a system label item (either through the `label` program or with **x/zladd**), the system label item will go *after* the property labels (which are treated as unknown system label items). This can be a problem on the old system if the property labels are very long, and it is definitely a problem if you then try to use the image on a system that supports property labels. Third, property labels will not be copied from the primary input to the output on a system that does not support them. The information would be lost. So, be careful when using property labels if there is the possibility of mixing systems.

If you are implementing something new, like a look-up table or a new flight label, then you can use the property labels with no problems. However, if you are moving something that used to use history labels to the property label, like the map projection or an old flight label, then you have to make sure that you can read files with the information either in the property label or in the history label. This is because 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.

4.6.2 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 are only 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.

It is important to note that the name registry system is somewhat voluntary, in that 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.

4.6.3 Property Label Instances

In the current implementation, there is one and only one instance of any property set. However, there are plans for the near future to extend this and allow named property instances.

Under this scheme, there may be multiple sets of properties with the same PROPERTY label. These sets are distinguished via an *instance name*. The combination of property name and instance name must be unique. This is similar in concept to history labels, where there are multiple instances of a given task. However, while history instances are numbered, property instances are named.

The motivation for this change comes from the IBIS file format, where you want to be able to associate a set of properties not with the whole file, but with an individual column or set of columns. There are other uses, however. For example, a mosaic program might include the map projection label for each of its component images via separate instances of the map projection property.

The property instances will be compatible with the current, non-instanced property names. The semantics will be that a property with no instance name is the default, which is used if the desired instance name is not present for that property.

The details of this system, including the API, will be announced separately when it has been developed. This section is merely intended to give the reader a heads-up of coming features. In order to maintain compatibility with this new system, the label key "INSTNAME" should be considered reserved for future expansion, and should not be used in application-defined labels.

5 Data Types and Host Representations

This section deals with data types for image files. Handling different host data formats correctly is one of the biggest porting challenges.

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 those types of machines must be byte-swapped. In the floating point area, most machines use the IEEE standard, but VAXes have their own standard. Some of the IEEE-format machines are byte-swapped relative to each other, too. Data being transferred among these machines must be converted as well.

5.1 VICAR File Representations

This section addresses how pixel data is stored in a VICAR file so it may be accessed by different machines, and the reasoning behind that decision.

Conversion among hosts would be greatly simplified if everything were stored in ASCII instead of in a binary representation. In fact, that is how the image labels are stored. However, that is terribly 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, for example that everything must be stored in Sun format (which is big-endian, IEEE floating point). That would simplify the file format a little, but it would lead to very inefficient operation on other machines with different formats. If you were doing processing locally on a VAX, and every pixel had to 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! Besides, 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.

Should the user be responsible for converting the data when moving to a different machine, or should it be automatic? In the first VICAR conversion, from the IBM to the VAX, the user was responsible for running a conversion program. This was reasonable at the time, since there was no direct connection between the machines, and data from the old format had to be read off tape. Since that is a manual operation, an extra manual step was acceptable.

In this port, however, the situation is very different. MIPL will have a heterogeneous operating environment, with many different computers and different data representations running simultaneously. It is quite likely that they will share the same disk farm over the network, so the same data is transparently available on all the machines. The user may not be aware of what machine the file was created on. Therefore, it is unreasonable to expect the user to do his/her own translations. It must be handled automatically, and be completely transparent to the user. The user may notice a slowdown if the conversion is being performed, but the operation will still take place as expected.

This places the burden of data format translation squarely on the shoulders of the programmers. Applications *must* be able to do data format translations automatically. In order to ease

the burden somewhat, 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 the read side greatly simplifies the write side, while still ensuring 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 a large part 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 cases where applications will have to do their own conversion, however. The major ones are listed below.

- Binary labels: Binary labels (both headers and prefixes) are a major concern, enough that they have their own section below.
- 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. This makes Array I/O much less attractive in many cases, but it is necessary.
- 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.

In all cases the **x/zvtrans** family of RTL routines should be used to do the translations. 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 cannot 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. Plus, the translations are standardized, and thoroughly debugged. The **x/zvtrans** routines are coded to be efficient, especially for simple byte-swapping, so there is no need to write your own.

5.2 VICAR Data Type Labels

As mentioned above, 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. Note that the label items are identical to the keywords in Section 4.1, New Optional Keywords.

- **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

HOST label	Description of host machine
ALLIANT †	Alliant FX series computer
AXP-VMS	DEC 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 or any SPARCstation, or clone such as Solbourne running SunOS
TEK †	Tektronix workstation
VAX-VMS	DEC VAX running VMS
X86-SOLR †	Intel x86 machine running Solaris 2

†Host machine is not officially supported

Table 6: Valid VICAR HOST labels and machine types

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

Table 7: Valid VICAR integer formats

representation of the pixels in the file. Nevertheless, it should be kept consistent with INTFMT and REALFMT.

Table 6 lists 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 the ones listed below appearing in the label.

- **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.

  The valid values of INTFMT may change as the RTL is ported to new machines. However, the currently valid values are listed in Table 7.

- **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.

  The valid values of REALFMT may change as the RTL is ported to new machines. However, the currently valid values are listed in Table 8.

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

Table 8: 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 BREALFMT 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 BHOST are exactly the same as for the HOST label item above.

- **BINTFMT**, string: The format used to represent integers in the binary label. BINTFMT, BREALFMT, and BHOST 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, BREALFMT, and BHOST 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 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.

  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. Only names that are registered should be used in BLTYPE. See Section 5.6.3, Binary Label Type, for more details.

In addition, one label item is used to specify the data type. It is not new, but is listed here for completeness. 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.

- FORMAT, string: The data type of the pixels in the file. The valid values are unlikely to change, unlike the host format labels. They are:

  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).

5.3 Pixel Type Declarations

All pixel buffers in an application must be declared using the standard pixel type declarations. See Table 2 for the standard Fortran declarations, and Table 4 for the standard C declarations.

5.4 Pixel Sizes

In the VMS-only world, pixels were always fixed sizes. Many programs assumed that FULL (integer) was 4 bytes, DOUB was 8 bytes, HALF was 2 bytes, etc. This assumption is no longer valid. A given machine may have different sized data types, such as 64-bit (8 byte) integers. Although all current supported machines have the same sized data, it is still important to code programs such that they will work with different sizes.

There are really two aspects to pixel sizes. The first is when you are dealing with files. The file could be from a different machine, with different pixel sizes. In this case, you must use one of the RTL pixel size routines, **x/zvpixsize**, **x/zvpixsizeu**, or **x/zvpixsizeb**. These routines give you the size of a pixel in bytes, given the data type and machine formats (**x/zvpixsizeu** and **x/zvpixsizeb** pick up the machine formats from an image label). Do not simply assume that REALs are 4 bytes long, as that may not be the case on every supported machine. The pixel size routines are designed to be easy to call.

The second is when you are dealing with internal buffers. 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. Make sure that you use the proper data type for the pixel (see Table 4). Fortran has no `sizeof()` operator or equivalent, so you should use **xvpixsize** with machine formats of "NATIVE" or "LOCAL" instead.

5.5 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 set up 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 note that you can have as many translation buffers active at the same time as you wish. They can be set up ahead of time, then used whenever they need to be in the program. This is illustrated in the example below.

There is really 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_in**, and **x/zvtrans_out**. See the subroutine descriptions in Section 4.2.2, Translation Routines, for details on each routine.

The question arises of 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/zvtrans_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. On occasion, 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 the 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. It would have to be fleshed out to compile or run.

```
/* 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. */
```

5.6 Dealing with Binary Labels

Binary labels are application-defined extensions to a VICAR image that are used to store ancilliary information about the image. They are made up of two parts: binary headers, which are extra records that appear at the beginning of the image, and binary prefixes, which are extra bytes that appear at the beginning of each image record. Binary labels are not part of the image data. In fact, an application will never even 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 organizations) .

Binary prefixes are extra bytes that appear at the beginning of each and every 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 always 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 probably the most difficult porting challenge for application programmers. The basic problem is that the data stored in them is application-defined. The RTL has no idea what types of data are stored in the 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 the data to the native host format when it is read.

To make things worse, only a few application programs actually 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 can 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. How do you deal with the binary labels? They could certainly be ignored, but then the information would be lost. This would be a frequent occurence for a program like COPY or LABEL, and the binary labels would disappear often. Therefore, many general-purpose programs simply 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 does not need to know what kind of data it is.

But what happens when you start mixing machine types? Say you have a file with binary labels that 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.

This is clearly untenable, as the system labels say the file is in Sun format, but the binary labels are still in VAX format.

5.6.1 Separate Host Types

The solution to the binary label problem is to have *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. This of course means that 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 way around it. As long as applications make sure they use the binary label host formats while accessing the binary labels, there won't be a problem. However, it does place a heavy burden on application programmers to make sure they access binary labels correctly.

In the best of all worlds, all application programs that use a particular kind of binary label would be ported at once. That way, when one of the programs starts writing data in Sun format,

for example, all the other programs will be able to read it. However, this is not always practical. Some applications will be converted before others, and if a program writes a binary label in a non-VAX format, it will not be read correctly by an unported program. For this reason, the implementors responsible for some kinds of binary labels may choose to keep them in VAX format, at least temporarily, regardless of the machine they were run on. This way, unported programs can still access the data correctly. Image data doesn't have this problem because of the automatic RTL conversion.

Doing this violates the general rule of read anything, write native, since the applications are writing in VAX (possibly a non-native) format, but it is justifiable in some cases. Binary labels are typically small, so this does not impose much performance penalty. However, *all portable programs must be able to read any host binary label format*, even if the decision is made to always write the binary labels in VAX format. The reasons for this have more to do with system integrity than anything else. Eventually, after everything is ported, the decision may be made to start writing the binary labels in the native format, which will simplify the code that writes them considerably. It would be ridiculous to keep things in VAX format when everything is running on a Unix machine. If all the programs already read any format, then there will not be a problem when the switchover is made. If a program was not coded to read any format, then it would read incorrect data after the switch.

An alternative to forcing VAX format would be to port all the programs that read the binary labels first. Only after all the readers are ported would the porting of the programs that write the binary labels begin. This way, you are assured of all applications being able to read the binary labels at all times, since everything that could use them would be able to read a foreign format before a file in that format was ever generated.

Note that this problem of timing the porting of programs applies not only to programs that use binary labels, but also to programs that use a file for anything other than image pixel data. Some files currently in use in VICAR, like statistics files and old-style IBIS files, contain several different data types but are stored using the structure of a VICAR image file. Like binary labels, the RTL cannot convert the data in these files, so the application must do it. The issues discussed above, regarding always writing in VAX format and porting the read programs first, apply to these files as well.

One additional thing to watch out for with binary labels is when using UPDATE mode to change a file in place (i.e. modifying the file directly rather than copying it). In this case, the file is not converted to native format, since it is being modified in-place. 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 same is of course true for image data, but the RTL handles the conversion automatically.

It is highly recommended that a set of subroutines be written for each type of binary label to read/write/update that label. If all applications used this set of subroutines, then 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.

5.6.2 Using Binary Labels

This section discusses how application programs make use of the binary label support features provided by the RTL. All these features are documented individually in Section 4, RTL Routine

Changes, but an attempt is made to tie them all together here.

It should be noted that the use of binary labels is discouraged, due to the many portability problems outlined above. Where possible, property labels should be used instead, as they can usually serve the same function as binary headers (but 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 Section 4.6, Property Labels, 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, and the optional arguments mentioned may be used with either routine.

For an input file, the 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 (which could happen with 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, the binary label host formats are obtained from the file, just like for input files. The only difference is if you wish to change the binary label type of the file. You of course 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, 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, however.

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 a little more complex. There are two basic cases: either you are converting the binary labels to native format, or you are leaving them alone. Which option you choose is controlled by the BIN_CVT optional argument.

If you set BIN_CVT to ON, then you must write the binary labels in the native host format. The BHOST, BINTFMT, and BREALFMT system label items get set automatically to the native host formats. There is no override; the corresponding optional arguments are disabled when BIN_CVT is ON. You should also turn BIN_CVT ON if you are writing a new file with binary labels in native format (although the name implies "convert", it really 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, so please set the BLTYPE label using the BLTYPE optional argument. See the next section for a description of BLTYPE.

If BIN_CVT is OFF (which is the default), then the assumption is that you are copying the binary label without converting it. The binary label host formats used will come from the primary input file. If the primary input is not available, they default to VAX format. The primary input may of course 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 BIN_CVT

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, then make sure you specify that specific format in the optional arguments. You should 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 BIN_CVT is OFF, and you should set it if you know the kind of binary label being written. BLTYPE will be picked up from the primary input if you do not specify it. See the next section for a description of BLTYPE.

Once you have opened the file, then you must translate the 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 5, Data Types and Host Representations, except you are 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 it can be used. There can be no exceptions to this rule.

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

5.6.3 Binary Label Type

Historically, there has been no way to determine what kind of binary label was on a file, just the size of it. You could guess it was a certain kind (like a Voyager label), but if you were incorrect you would get invalid data, with no error checking. You simply had to know beforehand what kind of data you had.

In order to solve this problem, a new system label has been added, called BLTYPE (Binary Label TYPE). While BLTYPE does not affect portability *per se*, it does greatly improve the documentation of files and it makes the VICAR system more robust. Everyone using binary labels, whether they are working on a portable program or not, is encouraged to make use of BLTYPE.

BLTYPE is simply a string that describes the kind of binary label present in the file. It may be set via **x/zvadd** or **x/zvopen**, or by calling **x/zladd** on an already open file. No error checking or valid values are enforced by the RTL, so it is up to application programmers to make sure it is used properly.

The BLTYPE string should be a short string (maximum 32 characters) that gives a name to 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”.

In order to maintain a consistent set of names, a name registry (similar to the one for properties) has been established for BLTYPE. Every value for BLTYPE must be entered into this registry, with a pointer to documentation describing the layout of the binary label. If you want to create a new kind of binary label, simply tell the keeper of the registry what name you want to use and what the layout is (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.

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

It is important to note that 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 should 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 enable them 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 they could be completely different labels from different projects. Widespread use of BLTYPE will eventually allow a general-purpose program to be written that understands most if not all of the binary label types.

6 Porting C

This section talks about porting existing C code for VICAR applications. It does not go into all the rules for writing portable C; see a reference book on C for that. Rather, it talks about portability in the VICAR context. It also describes how to deal with machine dependencies in VICAR code, which is useful when writing new code as well as when porting old code. Finally, it discusses how to make use of ANSI C in the VICAR environment.

6.1 RTL Differences

When porting a C application, all of the RTL routine calls will need to be changed, due to the addition of the C-language interface. A portable application *must* use the C-language interface. Although using the old Fortran interface will work under VMS, it works *only* under VMS. A C application using the Fortran interface simply will not work on any other machine.

The new RTL calling sequence is described in detail in Section 3, RTL Calling Conventions. As a reminder, the highlights are listed again below.

- Name change. The routine names for the C-language interface start with a "**z**" instead of an "**x**". So, **xvopen** becomes **zvopen**.
- Terminator. All RTL routines with keyword-value pairs of optional arguments must have a terminator in the argument list, even if none of the optional arguments are used.
- Pass by value. All input values to the RTL are passed by value in the standard C way, rather than being passed by reference. Any output variables or arrays are, of course, still passed by reference. Practically, this means taking out a bunch of "`&`"'s before the arguments.
- Status return. The status code is now the function return, rather than being an argument. If automatic error checking is on (via **zveaction** or one of the "*_ACT" keywords), then the status return can be ignored.
- Optional arguments. Pure optional arguments are not allowed. Most routines require all arguments to be specified, but a few have had some removed. See Table 1 for a list of all the former optional argument routines.

6.2 Include Files

The standards for accessing include files have changed. The changes are necessary to ensure portability across platforms. On the VAX, there was not that much difference between putting an include filename in double quotes, angle brackets, or without any delimiter at all. To be portable, attention must be paid to which of these methods you use. Note that the include file standards have changed since the first draft edition of this *Porting Guide*.

There are five major classes of include files you will use. They are system, VICAR main, VICAR system, VICAR subroutine, and local.

Please keep track of which include files you actually need, and include only those. Some includes require that others be included first, but try to achieve the minimum set. Extra includes in your program don't gain you anything, and they slow down your compiles. They also make system maintenance much tougher, 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 of them 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. As examples, some of the known portable includes are listed below.

```
#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 there), 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 has changed to "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 those 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. Please include *only* the ones you need. Extraneous include files you don't use 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 pathnames, and should include the ".h" extension, just like for 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. See the descriptions in Section 11.1, `vimake`, for details.

```
#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 pathnames, and should include the ".h" extension, just like for VICAR system and subroutine includes. However, local includes must be listed in the INCLUDE_LIST macro in the imakefile so they can be cleaned up properly. See Section 11.1, `vimake`, for details.

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

6.3 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 floating point value as a single. This works because on VMS the first half of a double value looks like a float. This is not the case on any other machine.

All the VMS-specific code must, of course, 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 the next section, Section 6.4, Machine Dependencies, 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 C code; if not then see a standard C manual. If in doubt, try it, or ask.

- 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. Use a type cast or the **zvtrans** routines instead. This practice is more prevalent in Fortran than in C, but it is something you need to watch out for.

- Integers and double precision floating point values must not be used as short ints or single precision floats without conversion. On VMS, a pointer to a double could be treated as a pointer to a float. This works because the bit pattern for the first half of a double is the same as for a float. The same is true for short vs. normal integers — a pointer to an int could be treated as a pointer to a short or even to a byte. This is not valid on any other machine. The IEEE floating point standard (which most other machines use) has different bit patterns for doubles and for floats. Therefore, if you wish to use a double as a float (or

vice-versa) you must convert the data with a type cast. For integers, most other machines use the "big-endian" byte order. The first bytes of an int are actually the higher-order bytes, unlike VMS, so the value cannot be treated as a short.

This practice is most common (and hardest to find!) in subroutine calls, where a program is sloppy about passing a pointer to the correct data type. If the subroutine expects a pointer to a float, the pointer better point to a float! Fortunately, C normally passes items by value, so floats get promoted to doubles in the subroutine call automatically. However, when things get passed by pointer instead of by value, the data type needs to be checked carefully. The function prototyping in ANSI C should help find these kinds of problems.

- 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 ($) in it is suspect, as most VMS system routines have a $ in the name.

  There are basically two choices for dealing with these. The first (and better) choice 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). The second choice is to isolate the VMS-specific code, and write corresponding code that works on a Unix system. Both versions could either be included in the program itself (in-line or in a subroutine), or they could be put in a SUBLIB subroutine if it's a capability that could be generally useful. See the next section for ways of handling machine-dependent code.

- Assembly language (called Macro on the VAX) included in the program will of course have to be rewritten. You could write assembler versions for every machine supported, but this is *highly* discouraged due to the wide variety of available assembly languages. The best solution is to write the subroutine in a high-level language, preferably C. The macro code can be kept for use on the VAX if required, as long as there is a high-level language version for other machines. With the rapid rise in machine performance over the past years, which is expected to continue in the future, the use of assembly languages for performance reasons is becoming very hard to justify, once the extra development, maintenance, and porting costs are taken into account. If a particular machine *really* needs the performance boost, an assembler routine can be written for that machine, assuming a C version exists for other machines.

- Descriptors will have to be eliminated. Only a few machines use descriptors for Fortran strings, and the other machines that do are not compatible with the VAX format. Descriptors normally appear only in system calls (which have to be changed anyway) and in interfacing between Fortran and C. The Fortran-C interface is described in Section 9, Mixing Fortran and C, and there are routines available in the RTL to do string conversions (the **sfor2c** family).

- Fortran-C interfaces will have to 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 C by its Fortran interface, as that is not portable. String handling is a particular problem. See Section 9, Mixing Fortran and C, 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 Section 5, Data Types and Host Representations, for details.

- Do not assume that you know the size of a pixel in an input file. An integer may not be four bytes on every machine that VICAR runs on. Use the RTL routines **zvpixsize**, **zvpixsizeu**, or **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.

- Structure packing is not the same on all machines. Some architectures, like the VAX, pack all the items in a structure together with no extra space. Other architectures put every element of a structure at a longword boundary, or something similar. That is because they have access restrictions on data, such as not being able to access an integer at an odd address. This can cause problems when writing structures directly out to a file (where another machine may read them in), or trying to overlay one structure on another to access certain fields differently. If you need the size of a structure, don't just add up the size of its elements, use the `sizeof()` operator on the entire structure. For an example of reading a structure from a file, see Section 5.5, Converting Data Types & Hosts.

- Use of unions to access the same bit pattern in different ways (like EQUIVALENCE in Fortran) is very unportable. If unions are used, make sure you use only one of the definitions for any given piece of data. If you use the other definition, you must put a new value of that type into the union. Unions may not be the same size on different machines, either, due to structure packing considerations.

- Not all the "standard" C run-time library routines are the same. For example, the `memcpy()` function on VMS will handle overlapping moves correctly (where the source and destination ranges overlap), but the same routine on the Sun does not handle them like you might expect (use the **zmove** function instead). The `open()` call on VMS, while standard in most respects, will accept extra arguments to define the RMS file type. These arguments are not portable. Some of the less common or newer routines are not available everywhere. The Alliant has a rather limited set of string handling routines, for example.

- Optional arguments are not allowed in subroutines. You must 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 or flag bits. VICAR constants could change in the future, and "CANNOT_FIND_KEY" is much more understandable than "-38". System constants, for example the argument to the `lseek()` function that says to search from the end of the file, are even more likely to be different on different machines. They should always be symbolic constants, from the appropriate system include files.

- Global variables should not be initialized in include files. Under VMS, a global variable can be initialized in an include file that is used by several program modules. As long as the

initialization is the same in each module (which is true since it comes from an include), then it works. On other machines, however, this is not the case. You get a "multiply defined" error on the global variable. One way to get around this problem is to use a preprocessor macro to control the initialization. If the macro is defined, initialize it. If not, just declare it. The main program module then defines this macro before including the file in question. That way, the variables only get initialized once. For example:

```
/* file x.h */
#ifdef INITIALIZE
int glob = 5;
#else
int glob;
#endif

/* one of the program modules (preferably the main one) */
#define INITIALIZE                /* delete this line in other modules */
#include "x.h"
```

- Make sure that variable and subroutine names do not conflict with Unix system routines or the RTL internal routines. Of particular interest is the use of the variable name "stat", which is common in VICAR code for a status return. This conflicts with and overrides (if it's a global) the Unix system routine of the same name, which is critical for the RTL to work. Use of the name "stat" will cause your program to break. Use "status" or some other name instead.

  The other part of the problem is that 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, you must 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 pathname of "`disk:[dir.subdir]file.ext;version`" is quite different from the Unix pathname 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, but **zvopen** and **zvfilename** do). Filenames and pathnames 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 pathnames 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 **zvunit** take care of it), so the solution is simply to not specify the maximum string length in the PDF. Instead of saying "(STRING,40)", just say "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 may need changing. C language modules must end in a ".c" to be compatible with `vimake`. Other files have naming rules as well. See Section 11.1, `vimake`, for details.

6.4 Machine Dependencies

There will be times where machine dependencies cannot be avoided, and will be required in the code. These dependencies can be based on the operating system (VMS vs. Unix), or they can be based on the actual machine type or flavor of Unix that is running. Not all variants of Unix are created equal, and sometimes there can be significant differences among machines. This section describes how to handle such machine dependencies in your code.

There are two major ways of dealing with machine dependencies. The first is to isolate them into separate source files, and compile only the one that applies to the machine you're on. The second is to have the machine-dependent code in-line, and use the C preprocessor to select the appropriate version.

The first solution, separate source files, is appropriate when there are whole routines that are implemented differently under the different 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. The RTL uses this method internally quite often. The filenames should then by convention end in an underscore and the OS name, for example, "open_input_file_vms.c" and "open_input_file_unix.c".

Once you have the files separate, you must somehow get the appropriate one to compile during a build. This is handled in the imakefile (see Section 11.1, `vimake`, for details on the imakefile). The MODULE_LIST (or other appropriate macro) would be defined differently 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". The module list for the imakefile would look like this:

```
#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 is needed to make sure that the source code for the module not compiled is deleted during a clean-source operation. See Section 11.1, `vimake`, for details.

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

For differences between VMS and Unix, use the symbols "VMS_OS" and "UNIX_OS". They are defined as 0 or 1, so use "#if" instead of "#ifdef". You may use a "#else" to select the other operating system, but if you do, the else clause should apply to Unix. In other words, put "VMS_OS" on the if statement.

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

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

Differences between various flavors of Unix are a little more involved. The file xvmaininc.h defines symbols for all the types of machines VICAR runs on, like "VAX_ARCH", "SUN4_ARCH", "MAC_AUX_ARCH", etc. *These must not be used in program code!* Think what would happen when an attempt was made to port VICAR to a new machine if they were. This new machine is likely to have dependencies that are different than any other machine. So, someone would have to go through *every* program and subroutine in the system, examine all the #if statements, and add the new machine to each and every one of them in the appropriate place. That would be a horrendous job, and would make VICAR virtually impossible to port.

There is a solution. Differences in machines are not really the issue here. Differences in *features* are. 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 the Silicon Graphics and HP do not. When you want to get the optimal blocksize, you don't care what machines have it, just whether or not the feature you want is present. Another feature, the `mmap()` command, is available on Sun 4 and Silicon Graphics, but not on DECstation or HP. Note that the groupings are not the same. If you code machine dependencies based on *features*, rather than *machines*, then porting to a new machine means simply determining whether or not it has the feature in question, and setting up the include files properly. (Actually, a newer version of the HP operating system now supports mmap(). To make it available, only one define had to be changed.)

The file xvmaininc.h defines many of these machine dependencies that are relevant for the RTL. 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, must be the only ones used to handle machine dependencies. Please make sure all such defines do end in "_OS", so searches through the code to find the machine dependencies will be easier. This leads to the following general statement:

> Conditional compilation statements to handle machine dependencies shall never use machine names or types. They shall instead use names of specific features that are

required in the code. These names shall be defined in standard include files based on the machine type.

Where do these feature defines come from? As mentioned previously, all the ones needed by the RTL are in xvmaininc.h. You should examine that file to see what machine dependencies are currently known, and check it from time to time, as it will change. You will come across other areas of machine dependencies that aren't listed in xvmaininc.h. There are two ways to deal with these. The first is to contact the VICAR system programmer to add the dependency to xvmaininc.h. This is required if you need to use the dependency in an imakefile, as xvmaininc.h 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 convenient for an application programmer to change the file. For that reason, an include file is provided in the portable includes (p2$inc on VMS and $P2INC on Unix), called "vmachdep.h", that contains dependencies that are needed only by application programs. Use the style of xvmaininc.h when adding new dependencies, and make sure you find out 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. Due to the unique nature of "vmachdep.h", please try not to keep it reserved for very long. Reserve it, add your definition, and redeliver it, even if your application isn't ready for delivery. That way, other programmers can modify the file as well.

Do not define any feature dependencies anywhere other than these two files (xvmaininc.h and vmachdep.h). That will make porting VICAR much easier in the future.

Some examples are in order. These are paraphrased from the RTL source, but the same rules apply. If your definition is in vmachdep.h, the only difference is that vmachdep.h needs to be included in your source.

```
#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
```

6.5 ANSI C

ANSI C is now recommended over the older K&R C for all new VICAR programs. This section does not attempt to describe ANSI C; see one of many reference books on the topic for that. Rather, it describes what needs to be done within the VICAR environment in order to successfully use ANSI C. This entire section should be read before using ANSI C with VICAR.

The most important advantage of ANSI C over K&R C is function prototypes. 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, if used properly, should catch many common programming errors.

Function prototypes are not a requirement; ANSI C allows the old style, non-prototype declarations as well. However, if the prototypes are available, you should make use of them. You will undoubtedly find many more compiler warnings if you use prototypes, but that is a good thing. Most of those warnings are in fact errors in the code, or questionable coding practices. Take heed of them, 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 you must be careful to use only argument types that won't be promoted in the absense of a prototype in order to remain compatible. Any ANSI C reference should be able to provide more details on this.

The most noticeable difference for programs when you start using ANSI C is the type of the `main44()` function. Previously, the return type of `main44()` was undefined, so it defaulted to

int. When using ANSI C, `main44()` is defined correctly as returning nothing. So, you must now declare it as "`void main44()`" in your main program.

Other than that, 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.

For subroutine writers, using ANSI C is a little more involved. You should provide an include file that defines the prototypes for the functions in your subroutine package. The include file may be very short if you have only one function, or quite long for a big package. Note that even if you are writing in K&R C, you should still provide a prototype include file so that ANSI C (or C++) users can make better use of your function. This include file should generally be the same one that you put your other public definitions in, 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_PROTO, 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, it is likely that 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. The third thing is to allow support of C++. C++ uses the same prototype mechanism as ANSI C, with the exception that prototypes are required (rather than recommended). 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. It can be treated 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 to be aware of regarding ANSI C prototypes. First, and most important, do *not* put prototypes on any Fortran bridge routines. They are not needed because Fortran doesn't understand them, and the prototypes mess up the Fortran string passing mechanisms. Use the old-style declarations instead. Of course, 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. A strict reading of 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 don't use 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.

Finally, there is the issue of VAX compatibility. VICAR is currently based on VMS 5.4 (for the VAX; the Alpha uses a more recent version). This version of VMS does not have a true ANSI C compiler available. The compiler does support most ANSI C features (such as prototypes), but it is not fully compliant. Pay special attention to the VAX/VMS compiler when using ANSI C within VICAR.

7 Porting Fortran

This section talks about porting existing Fortran code for VICAR applications. It does not go into all the rules for writing portable Fortran; see a reference book on Fortran for that (there are a few allowed VMS extensions to Fortran, which are described below). Rather, it talks about portability in the VICAR context. It also describes how to deal with machine dependencies in VICAR code, which is useful when writing new code as well as when porting old code.

7.1 RTL Differences

When porting a Fortran application, most of the RTL routine calls will need to be modified slightly, due to the need for an argument list terminator and the elimination of optional arguments. A portable application must use the new rules below when calling the RTL. Although following the old calling conventions instead will work under VMS, they will work *only* under VMS. To be portable, the new conventions must be used.

The new RTL calling sequence is described in detail in Section 3, RTL Calling Conventions. As a reminder, the highlights are listed again below.

- Terminator. All RTL routines with keyword-value pairs of optional arguments must have a terminator in the argument list, even if none of the optional arguments are used.
- Character strings. All strings input to the RTL must be CHARACTER*n variables or constants. Formerly, BYTE or LOGICAL*1 arrays were allowed in RTL arguments, but no longer. BYTE and LOGICAL*1 do not have a string length indication, and there is no way for the routine to tell whether the argument is a CHARACTER*n (which does have a length), or a byte array. Since the length is required, BYTE and LOGICAL*1 no longer work. This applies both to string arguments and to keywords in the keyword-value pairs, since they are strings also. The change to CHARACTER*n is going to be most troublesome with the output routines **xvmessage** and **qprint**, since they are often called with BYTE arrays. See Section 7.5, READ & WRITE to Strings, below, for details on new methods of doing string output.
- Optional arguments. Pure optional arguments are not allowed. Most routines require all arguments to be specified, but a few have had some removed. See Table 1 for a list of all the optional argument routines.

7.2 Include Files

Fortran include files are a tough problem under Unix. That is because on some Fortran compilers (notably on the Sun), the INCLUDE statement does not accept environment variables, nor is there a way on the command line to specify a directory for includes. That means that includes must either be in the current directory during the compile, or they must have an absolute pathname in the INCLUDE statement. The latter solution is clearly unacceptable, since the directory names change not only from system to system, but for different versions (development, operational, etc.) of VICAR!

This means that all includes must be in the current directory in order for the Fortran compiler to find them. This is achieved via the build files created by `vimake`. On Unix, they create a symbolic link for each include you use (which must be specified in the imakefile) in the current

directory. The compiler can then find the includes via these links. The links are deleted when the compile is finished. On VMS, temporary logical names are created for the includes you use that point at the correct location. The logical names go away after the compile.

There are four major classes of include files you will use. They are VICAR main, VICAR system, VICAR subroutine, and local. There are no system includes available in Fortran.

Please keep track of which include files you actually need, and include only those. Some includes require that others be included first, but try to achieve the minimum set. Extra includes in your program don't gain you anything, and they slow down your compiles. They also make system maintenance much tougher, 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).

The VICAR main include file, which all VICAR programs should start with, must be in upper case, with single quotes. The name of the main include has changed to "VICMAIN_FOR", with no directory specifiers. Do not add the "`/NOLIST`" qualifier that is common in current VICAR applications. Since all applications need VICMAIN_FOR, it is made available automatically in the build files, and you do not need to list it in the imakefile.

```
INCLUDE 'VICMAIN_FOR'
```

VICAR system include are those provided as part of the RTL. Their names should be in lower case, without a file type extension or pathname. They should be in single quotes. The valid ones are listed below (as of this writing, there is only one). They must be listed in the imakefile, in FTNINC_LIST, to be accessible. See Section 11.1, `vimake`, for more details.

```
INCLUDE 'errdefs'
```

VICAR subroutine include files are quite similar to the VICAR system includes. They should be in lower case, without a file type extension or pathname, and they should be in single quotes. They must also be listed in the imakefile, in FTNINC_LIST, to be accessible. The include file itself comes from one of the application include directories, such as p2$inc ($P2INC) or vrdi$inc ($VRDIINC), and must be in lower case with an extension of ".fin" (for Fortran INclude).

```
INCLUDE 'sublib_inc'
INCLUDE 'fortport'
INCLUDE 'verrdefs'
```

Local includes, which are delivered as part of the COM file and are used only by that application, should be referenced in lower case, *with* the ".fin" extension. They should *not* be listed in FTNINC_LIST, although all local includes must be in INCLUDE_LIST so they can be cleaned up properly.

```
INCLUDE 'my_include.fin'
```

7.3 No EQUIVALENCE for Type Conversion

The use of EQUIVALENCE to convert among data types, especially byte to integer, is widespread in current Fortran application code. A byte will typically be equivalenced to an integer, so storing the data in the byte variable makes it accessible via the integer variable. This practice

is absolutely not portable. EQUIVALENCE should *never* be used to convert any data types. If you use EQUIVALENCE, it must be done to conserve storage only. You must access the data using the same data type you stored it with.

To do data type conversion, use the **xvtrans** family of routines. See Section 5.5, Converting Data Types & Hosts, for details. In addition, a method is available for converting between byte and integer (since it is by far the most common conversion) that is more efficient than calling the **xvtrans** routines. This method consists of the functions `INT2BYTE` and `BYTE2INT`. To use them, you must include the file "fortport" in the subroutine. "fortport" is a SUBLIB include, so it must be listed in FTNINC_LIST in the imakefile to be available. These conversion functions assume that the data is strictly in the 0 to 255 range. No bounds checking is performed. The functions are actually implemented on some machines as array lookups rather than functions, and on others they may be statement functions, so the include file must be present to get the appropriate definition.

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

7.4 CHARACTER*n for Strings

Possibly the biggest challenge specific to porting VICAR Fortran code is in the area of character strings. Formerly, BYTE or LOGICAL*1 arrays (or even INTEGER arrays) were commonly used to store character strings (the term "BYTE arrays" here refers to all three). These strings were created with SUBLIB routines like `outcon` and `mvl`, and were used in **qprint** and `incon`. They were used interchangeably with CHARACTER*n strings as arguments to many subroutines, especially the RTL. This was made possible by the fact that CHARACTER*n variables were passed by descriptor under VMS, which meant that the called subroutine could tell whether it was a CHARACTER*n variable or a BYTE array (actually, the methods used were nondeterministic and failed on occasion, but they worked most of the time and were used anyway).

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. An attempt to pass in another type of array will cause your program to crash because the RTL expects string lengths on all strings (it will work under VMS due to backward compatibility, but on no other machine). This applies to all RTL routines, but is especially important for the output routines **qprint** and **xvmessage** (**xvmessage** is preferred over **qprint** for new code), since those routines are often passed BYTE arrays for output. All strings destined for output must be CHARACTER*n variables. The next section, Section 7.5, READ & WRITE to Strings, discusses how to do output conversion to create the strings (instead of `outcon` *et al*).

While changing output strings to CHARACTER*n is possible (it's a chore, but it's not hard), 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* just copy data directly to a CHARACTER*n variable from a BYTE array, or vice-versa, since that will fail if there are descriptors present (some other machines have descriptors, but they are formatted differently than the VAX). You must use `mvcl` or `mvlc` to convert strings.

Subroutines that expect strings should in general expect CHARACTER*n strings. In the cases mentioned above where it is impractical to change, then only BYTE arrays may be passed in. Passing a CHARACTER*n 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.

7.5 READ & WRITE to Strings

The conversion of strings from BYTE to CHARACTER*n implies that the methods used to do input and output to strings must change as well. Previous code often set up a buffer, then used `outcon` to put numeric values in the string. The result would then be printed. For example (taken from the program `amos`):

```
 REAL*8 VRES, VRAD, VTAN
 LOGICAL*1 MS1(52)/' ','V','E','L','=','*','*','*','*','*','.','*',
&'*',' ',' ','(','V','R','A','D',',','V','T','A','N',')','=','(',
&'+','*','*','*','.','*','*',',',',','+','*','*','*','*','*','.','*',
&'*',')',' ','M','/','S','E','C'/
 CALL OUTCON(VRES,MS1(14),9,2)
 CALL OUTCON(VRAD,MS1(35),7,2)
 CALL OUTCON(VTAN,MS1(45),9,2)
 CALL QPRINT(MS1,52)
```

`outcon` (and its counterpart, `incon`) have many serious portability problems. They would have to be converted to output to CHARACTER*n variables instead of BYTE arrays, since the output would be printed via **qprint** or **xvmessage** anyway. `outcon` and `incon` were originally created to avoid using the Fortran I/O library in the days on the IBM mainframe when memory was at a premium. That is no longer a constraint. The Fortran I/O library is more portable, and easier to use. For these reasons, `outcon` and `incon` will not be supported in the portable system.

To perform input and output conversion, simply use the Fortran I/O package on what is called an "internal file". An internal file to Fortran is simply a CHARACTER*n variable. Just give the name of the variable (it may be a substring if you wish) instead of the unit number in a Fortran READ or WRITE statement. Note that you must still write to a string and send the string to **qprint** or **xvmessage**. Do not attempt to write directly to the terminal. Any messages written directly to the terminal will not appear in the session log, and could very well be printed in unexpected ways due to interaction with VICAR I/O. Also, messages written directly will not be handled properly by the forthcoming VICAR GUI.

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, ' ')
```

Since an internal file may be a substring as well, you can do a more direct conversion (although the previous example is generally the preferred method):

```
      REAL*8 VRES, VRAD, VTAN
      CHARACTER*80 MS1
      DATA MS1 /'VEL=******.** (VRAD,VTAN)=(+***.**,+******.**) M/SEC'/
      WRITE (MS1(5:13), '(F9.2)') VRES
      WRITE (MS1(28:34), '(F7.2)') VRAD
      WRITE (MS1(36:44), '(F9.2)') VTAN
      CALL XVMESSAGE(MS1, ' ')
```

There are three things to note about this example. First, the DATA statement is separate from the declaration. Putting initialization data on the declaration line is not portable. Second, the format specifier may be put in-line in the WRITE statement if you wish. Last, 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 `able86` 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.

7.6 Array I/O

The RTL has a method of accessing a file, called array I/O, where the file is mapped to virtual memory, and the address of the file is returned. To access the file, the program need not use I/O routines such as **x/zvread** and **x/zvwrit**. The file appears as a single large array at the returned address. Actual file I/O is accomplished via the paging mechanism of the operating system. On operating systems where this is not available, it is simulated by reading the file into a large chunk of memory when it is opened, and writing it out again when it is closed.

The key to array I/O is returning the address of the file in memory. This address is actually a pointer to the data. Under C, this is no problem. Fortran, however, does not understand pointers. Therefore, array I/O is not allowed directly in Fortran. There is a trick that can allow it to be used in some cases, however.

Previously, the few Fortran programs that used array I/O did it by receiving the address in an integer variable. This variable could then be passed in to a subroutine using %VAL, which passes the argument by value instead of by reference. The result of this was that the pointer got passed by value. The called routine expected an array passed by address, in other words a pointer to the array. Since the variable containing the address was passed by value, the subroutine was happy.

Unfortunately, %VAL is not portable and may not be used. Therefore, pure Fortran is unable to use array I/O. However, C does not have any problems passing things by reference or by value. So, you can use array I/O in a manner very similar to the SUBLIB `stacka` subroutine. The **zvopen** call must be made in a C-language function. It can then pass the returned address by value to a Fortran subroutine. The Fortran subroutine can access the file by treating the parameter as an array.

This method requires that all the Fortran code that uses the array I/O file must be in the subroutine that is called from C. This is quite similar to the way STACKA works. This may require significant revisions of the programs that use array I/O from Fortran, but fortunately they are fairly rare.

Another alternative would be to simply not use array I/O. Any program that uses array I/O should have an alternate method of accessing the file using line I/O (**x/zvread** and **x/zvwrit**), in case the file is too big to be opened via array I/O. The line I/O backup is not required, but it is a good feature to include if possible. Use of array I/O only, with no backup line I/O, can severely limit the size of files that may be processed by the program. With a line I/O backup, a VMS version of the program could use array I/O if possible, but the Unix version of the program would use exclusively the line I/O alternative.

7.7 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. In general, only standard Fortran-77 code should be used in a portable program. To find out what is standard and what isn't, you could look at the ANSI Fortran standard, or you can look in 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 Fortran-77 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 you must provide a means for performing the same function on machines that don't have that extension. In other words, it's not worth it to use non-standard Fortran features.

These are the *only* allowed 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 in Table 2.
- 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.

7.8 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 the VMS-specific code must, of course, 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 the next section on Machine Dependencies 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). If in doubt, try it, or ask.

- 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 **xvtrans** 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 any other machine. The IEEE floating point standard (which most other machines use) 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 ($) in it is suspect, as most VMS system routines have a $ in the name.

  There are basically 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 the next section 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 of course 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. Section 6, Porting C, has more details about handling VAX MACRO.

- 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 will have to 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 Section 9, Mixing Fortran and C, 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 Section 5, Data Types and Host Representations, for details.

- Do not assume that you know the size of a pixel in an input file. An integer may not be four bytes on every machine that VICAR runs on. Use the RTL routines **xvpixsize**, **xvpixsizeu**, or **xvpixsizeb** 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, you must put a new value of that type into 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. You must 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.

- Make sure that variable and subroutine names do 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, you must 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 pathname of "`disk:[dir.subdir]file.ext;version`" is quite different from the Unix pathname 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 **xvopen** and **xvfilename** do). Filenames and pathnames 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 pathnames 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 **xvunit** take care of it), so the solution is simply to not specify the maximum string length in the PDF. Instead of saying "(STRING,40)", just say "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`. That is because Unix likes ".f", and it is more picky about filename extensions than VMS is. Other files have naming rules as well. See Section 11.1, `vimake`, for details.

- Many VAX Fortran language extensions are present 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.

7.9 Machine Dependencies

There will be times where machine dependencies cannot be avoided, and will be required in the code. These dependencies can be based on the operating system (VMS vs. Unix), or they can be based on the actual machine type or flavor of Unix that is running. Not all variants of Unix are created equal, and sometimes there can be significant differences among machines. This section describes how to handle such machine dependencies in your Fortran code.

Fortran is at a severe disadvantage when it comes to writing machine-dependent code. Simply put, Fortran does not have a portable means for doing conditional compilation. The C preprocessor does a wonderful job at this, but it cannot be used to conditionally compile Fortran code in a portable manner. For this reason, the best method to write machine-dependent code in Fortran is to write it in C instead!

If machine-dependent code has to be written in Fortran, it must be isolated into separate source files. Only the source files containing the code that applies to the machine you're on will be compiled. In-line code variants for different machines is not possible due to the lack of a conditional compilation method.

The filenames for the machine-dependent code should by convention end in an underscore and the OS name or feature type that it depends on, for example, "open_input_file_vms.f" and "open_input_file_unix.f".

Once you have the files separate, you must get the appropriate one to compile during a build. This is handled in the imakefile (see Section 11.1, cmd vimake, for details on the imakefile). The MODULE_LIST (or other appropriate macro) would be defined differently based on the machine type. Since cmd vimake uses the C preprocessor, the rules for machine-dependent preprocessor symbols should be used. For example, a program named "prog.f" calls a routine that is OS-dependent, named "sub_vms.f" and "sub_unix.f". The module list for the imakefile would look like this:

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

```
#define CLEAN_OTHER_LIST sub_vms.f
#endif
```

The CLEAN_OTHER_LIST macro is needed to make sure that the source code for the module not compiled is deleted during a clean-source operation. See Section 11.1, `vimake`, for details.

For more details on the preprocessor symbols allowed in the imakefile, see Section 6.4, Machine Dependencies, under the section on Porting C. Note that specific machine types must not be used, as described in that section. Use feature defines instead. Note that any such feature defines must be in xvmaininc.h, since that is the only include file that `vimake` can access.

8 Porting TCL

TCL procedure PDFs must be portable as well. TCL is the TAE Command Language, used to write most VICAR procedures. Although it is designed to be fairly portable, in that the commands are the same for any version of TAE, there are some things you need to watch out for.

The most significant thing that will have to be changed is use of the `dcl` command inside a procedure. This command allows you to put VMS DCL commands in your procedure, which are, of course, 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. Fortunately, there is a global variable called "$syschar" that 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 stick to common shell commands.

Another major trouble area for porting TCL is filenames. Filenames and pathnames 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 test files, based on which system you are using.

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

For example, the following lines are 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
```

You should refer to the TAE documentation if you have more questions on writing portable TCL.

9 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. Section 2, Portability Constraints, talks about the reasons behind some of the rules listed below.

In general, 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. This is because the necessary calling conventions are so different. If you are writing an internal subroutine which is used only internally to your program, you may need an interface for only one language. Also, some SUBLIB routines only make sense when called from one language. However, most subroutines will have two interfaces.

9.1 Bridge Routines

The calling interface for the language a subroutine is written in will be straightforward. You simply have to adhere to the rules for that language. The interface for the other language will typically be implemented via a "bridge" routine, which must be written in C.

This bridge routine accepts arguments in the format of the opposite language, and converts them to the form of the language the subroutine is written in. It then calls the main subroutine. For example, a subroutine written in C would have a bridge also written in C that would accept 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 calls 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). See Section 6.5, ANSI C, for details.

9.2 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. This is described more fully in Section 2.5, Subroutine Names.

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. The macro is used anywhere the subroutine name would be, including in the declaration of the subroutine. The subroutine name is the argument for the macro. Note that "ftnbridge.h" must be included. For example:

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

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

This would be called by the Fortran routine like this:

```
      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 be annoying in the caller's code, plus would cause problems if the routine were ever converted to C. The bridge routine takes care of this. For example:

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

The C bridge would look 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 like this:

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

9.3 Passing Numeric Arguments

The passing of numeric arguments and arrays between Fortran and C is fairly straightforward. The data type equivalences are listed in Table 9. 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 of course change the data type 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 pay attention to this table.

For numeric arguments, keep in mind that Fortran passes arguments by reference in all cases, while C normally passes input arguments by value. Such arguments must be converted in the bridge routine. A Fortran to C bridge would declare all arguments as pointers, then put asterisks (*) in front of the appropriate arguments to dereference them when calling the main routine. A C to Fortran bridge would declare the appropriate arguments as values (not pointers), and use an ampersand (&) in the call to the Fortran routine to convert them to pointers. This is shown in the examples in the previous section.

Be careful with the arguments, since 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 Section 9.4, Passing Strings.

Table 9: Fortran - C equivalences for arguments

Two-dimensional (or higher) arrays can be a major problem, since Fortran treats things in column-major order, while C uses row-major order. This means the order of the subscripts is reversed. This can be handled either by copying the array and reshuffling it in the bridge (which can be quite inefficient), or by documenting the fact that the subscripts need to be reversed in the other language to call the routine properly.

9.4 Passing Strings

Passing Fortran strings is quite a problem, since there is no standardization among vendors. The RTL currently supports six different methods of passing Fortran strings in arguments! Fortunately, 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.

All code that passes a string to or from Fortran *must* use the RTL string conversion routines (the **sfor2c** and **sc2for** family). And, any code that calls the conversion routines must set the FTN_STRING flag in the imakefile. See Section 11.1, `vimake`, for details.

Although it is possible to treat the string in Fortran as a BYTE array rather than as a CHARACTER*n variable, thus bypassing the need for the string conversion routines, this is highly discouraged. It is non-standard Fortran to use BYTE arrays for string manipulation and should be avoided.

9.4.1 Accepting Fortran Strings in C

Writing a C routine that accepts Fortran strings as arguments is not too difficult. Simply use the **sfor2c** (for input from a Fortran program) and **sc2for** (for output to a Fortran program) family of routines to convert the Fortran string into a C string and vice-versa.

The **sfor2c** and **sc2for** family of routines are described in Section 4.2.4, Fortran String Conversion Routines. There are examples in that section as well.

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.

9.4.2 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. First is the C-callable first bridge.

```
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);
}
```

This first bridge simply takes the length of the string parameter, and passes it along with the string pointer to the second-stage bridge. It also takes care of the scalar and array argument conversion, and the FTN_NAME macro. The second bridge is not intended to be called by anything except the first bridge.

```
      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
```

This second bridge converts the C string (passed as a BYTE array) and the length into a Fortran CHARACTER*n variable, which is then passed into the real `bodvar` routine (which is the Fortran-callable version).

10 SUBLIB Subroutine Library

This section discusses the creation and use of new subroutines for the SUBLIB library in VICAR. Although the discussion mentions only the class 2 subroutine library (p2$sub), the rules apply to the class 1 and 3 libraries (p1$sub and p3$sub) and other libraries (such as the HW library, hw$sub) as well.

The new VICAR directory structure allows the creation of more than one subroutine library in each application class. For example, the general SUBLIB library could be augmented with a Galileo-specific subroutine library with the subroutines that apply only to the Galileo project. Or, a Real-Time subroutine library could be created with subroutines that apply only to the Real-Time subsystem. While this capability is not currently implemented (there is only one subroutine library per class), the flexibility exists in the system to add these extra libraries in the future if they are deemed necessary.

It is important to remember that MIPL should not be doing a wholesale conversion of SUBLIB. Rather, subroutines should be converted as they are needed in the process of porting programs. Thus, if no ported programs use a SUBLIB routine, then it will not get ported. This helps keep the size of SUBLIB down, and should eliminate some of the unnecessary routines. The practical upshot of this is that the contents of the new SUBLIB will be changing constantly during the VICAR porting process. Keep a close eye on the available routines (the source code is in p2$sub or $P2SUB). If you need to port a SUBLIB routine, check to make sure nobody else is doing it, then if not, let configuration management know you are doing it. That way, duplication of subroutines can be avoided.

10.1 Relationship to Old SUBLIB

The portable SUBLIB is completely independent from the old VMS-specific SUBLIB. As a matter of fact, an application *cannot* use both at the same time. If it is a portable application, it must use the new library. If it has not yet been ported, it must use the old one.

There are many reasons for the complete split between the old and the new subroutines. The major reason is that it allows programmers to change the calling sequences easily for the SUBLIB routines, while still retaining the old names. Since the names are duplicated, an attempt to link to both libraries would cause an error. Another reason is that it ensures portable applications will not attempt to use non-portable subroutines, an error that would not show up on the VMS system. Finally, it makes configuration management of the system much easier, since portable subroutines are clearly isolated and identified as such.

10.2 When to Create a SUBLIB Subroutine

The first thing to ask when thinking about a new SUBLIB subroutine is: Is the routine generally useful? If the program you're writing is the only one that would possibly be interested in the subroutine, then don't put it in SUBLIB. There is a lot of subroutines in the old SUBLIB that are not particularly useful and are not generally used. Try to keep SUBLIB clean by not including subroutines that aren't generally useful.

On the other hand, make sure that anything that *is* generally useful gets put in SUBLIB. There are a lot of cases of duplicated code in the VICAR system that would benefit from being made generally available in SUBLIB (much of the Magellan software is a good example).

It's really a judgement call as to whether or not the subroutine should go into SUBLIB. The rules haven't really changed with the port to Unix. The port merely allows an opportunity to clean up SUBLIB with minimal impact. Just keep these guidelines in mind when thinking about creating a new SUBLIB routine.

10.3 Calling Sequences

Keeping the old and new SUBLIBs separate allows the programmer freedom to change the calling sequences for SUBLIB subroutines. There are several reasons for changing the calling sequences, which are discussed below.

The flip side of changing the calling sequences is that programmers doing a port must pay attention to the new calling sequences, which may not be the same as what they're used to. Programmers will have to be careful that the calling sequence they used is appropriate for the new SUBLIB. However, the benefits far outweigh those costs.

The first and most important reason for changing the calling sequence is to have separate Fortran and C calling sequences. The reasons for doing so have been described fully throughout this document. When the separate calling sequences are created, they should be appropriate for the language involved. The Fortran interface should look like a Fortran call, and the C interface should look like a typical C call. You should feel free to change the calling sequences to fit. For C, this often means accepting input arguments by value instead of reference. For Fortran, this often means accepting CHARACTER*n variables instead of BYTE arrays. Sometimes is it impractical to change to CHARACTER*n, if the BYTE array is too entrenched. In that case, consider providing *two* calling sequences, one with CHARACTER*n and one with BYTE. See the routine `vic1lab` for an example of this.

Another important reason to change the calling sequences is that optional arguments are not allowed. Many current SUBLIB routines accept optional arguments. These arguments will either have to be eliminated, or forced to be present. This may mean multiple subroutine entry points in some cases, one with the arguments present, and one without. In general, this should be avoided. Decide whether or not the arguments are important, and if they are, include them. If you provide a reasonable default (like 0), then it's not too hard for the application programmer to include the extra arguments when calling the routine.

The last major reason for changing the calling sequence is more a matter of style. Some of the SUBLIB routines have very obscure calling sequences, that are difficult to figure out. If there is a better way to call the routine, then feel free to change it. However, don't make gratuitous changes just for the sake of changing something; there should be a clear advantage in making the change.

10.4 Fortran vs. C

When porting a SUBLIB subroutine, an important consideration is what language to write it in. Many of the old SUBLIB routines are written in Fortran. You should consider rewriting some of them in C.

Many of these Fortran subroutines do system calls to perform some function (for example, `ostime`, `datfmt`, and `daydat` all use VMS system services to obtain the date in various ways). The VMS system services do not exist in Unix, so the code will have to be changed to make Unix system calls. However, Unix system calls from Fortran are not well standardized, whereas they are much more so when called from C. Also, VMS and Unix code will have to be separate, and

machine-specific code may have to be used to get around differences in different variants of Unix. It is very difficult to do machine-specific code in Fortran due to the lack of a preprocessor, while it is easy in C. Therefore, any code that calls the operating system directly should be written in C.

For subroutines that do not call the operating system, the case is less clear, and Fortran is acceptable. However, writing a C bridge for a Fortran routine is much more difficult than writing a Fortran bridge for a C routine, especially if there are strings in the argument list. Any routine that accepts strings in the argument list could definitely benefit from being written in C.

10.5 Other-Language Bridges

As mentioned previously, most SUBLIB routines written in Fortran need to have a C bridge, and most routines written in C need a Fortran bridge. The rules for creating such bridges, and examples of them, are in Section 9, Mixing Fortran and C.

There are cases, although rare, where the alternate-language bridge will not be needed. For example, the routines `mvcl` and `mvlc` convert strings between CHARACTER*n and BYTE arrays. This function is not useful in C, since C does not use CHARACTER*n variables and there are already routines to accept them in arguments. Therefore, these routines only need a Fortran interface, since they will only ever be called from Fortran.

Another example might be some of the Real-Time subroutines. The entire Real-Time system is written in C, including some SUBLIB routines. These routines are designed to be called only from Real-Time, which is entirely in C. Therefore, no Fortran bridge needs to be written for them. These routines are in SUBLIB even though they are used only by Real-Time because the Real-Time system consists of more than one application program.

10.6 Help Files

The issue of SUBLIB help files has not yet been fully resolved. At the time of this writing, the tentative plan is to deliver both program and subroutine help in HTML (HyperText Markup Language) form, which is used by the WWW (World Wide Web) and the Mosaic browser. The Mosaic document browser will then be the primary way for a user or programmer to access help. See the memo *MIPS Use of Mosaic for Documentation* by Peter Shames, IOM SE/PMBS-93.039, 21 Oct 1993, for details on this plan and HTML, WWW, and Mosaic .

However, there are many details to be worked out before this plan is implemented. The primary hurdle is to write converters from HTML to the other help formats, namely VMS `help`, TAE PDF, and Unix `man` formats, which are still required to be supported. For this reason, and others, the use of HTML for help is still TBD.

In the meantime, simply put VMS-specific subroutine help in the “other” category in the application packer. Help for programs or procedures goes in the PDF, as before.

11 Creating Applications

Once you have ported an application, you must be able to build it, and to deliver it to Configuration Management to go into the VICAR system. This section describes how to build (compile and link) programs, how to package up all the parts of the application, and some pointers on doing the test scripts necessary for class 2 (R2LIB) delivery to the VICAR system.

11.1 `vimake`

Most Unix programs come with a "makefile" that is run through the standard Unix command `make` to compile and link the program. The makefile escribes everything needed to build the program, including what options to use for the compiler, what libraries are needed for the linker, and where to install the manual pages. Using `make` has many advantages, including compiling only what has changed since the last compile.

Unfortunately, VMS does not come with a `make` program. There is a layered product available, called `MMS`, that performs the same function with very similar makefiles, but it spawns a subprocess to do the compilation, which is very slow under VMS. It is also not available on all systems. Therefore, it is not practical for VICAR applications or subroutines.

Previously, VICAR programs were built in one of two basic ways. The first, and older, method was to have compile and link statements directly in the ".COM" file that contained all the application pieces. The pieces would be extracted, and the compiles performed, all in one step. The second method was to have a separate compile and link file, typically named "CLapp.COM" (for Compile and Link the application). This file was packaged in the application COM file along with the source code. It could then be executed independently of the application COM file to build the program.

The use of an independent command procedure to build the program under VMS has many advantages. It allows the application COM file to be separate. It works on all VMS machines, since DCL is standard. And, it is fast, since no subprocesses need to be created. The old CL files suffer from a terrible lack of standardization, and are not the easiest things to use, but the basic concept is a good one.

So, it boils down to this. To build applications under Unix, it is highly desirable to use `make` with a makefile. To build them under VMS, it is highly desirable to use DCL with a command procedure. Obviously, these two files (the makefile and the command procedure) are going to be incompatible. Worse yet, there is quite a bit of variation required for makefiles on different Unix machines. The commands used often vary, and sometimes there are differences in the format and capabilities of `make` itself.

It would be very cumbersome to require the VICAR application programmer to maintain two sets of build files for every application (for VMS and Unix). Furthermore, the differences in Unix makefiles would make it almost impossible to come up with a single makefile for all Unix systems. Fortunately, a program called `imake` has been written to solve these problems.

`imake` is used extensively in the X-windows system to allow building the system on many different platforms. TAE uses it as well. It is a program that makes makefiles. `imake` makes use of the C preprocessor for macros and conditional statements in order to customize the makefile for a particular platform. The input "imakefile" is a reduced makefile that contains only the program-specific parts, not the system-specific parts. A template file provides the system-specific parts of the makefile for each machine type. The `imake` program does some minor cleanup on

the imakefile, runs it and the template through the C preprocessor, and does some cleanup on the output. The result is a makefile that is customized for the system you are on.

This helps to create Unix makefiles, but what about VMS? If you are familiar with `imake`, you know that the imakefiles typically look quite a bit like a makefile. There is a set of rules and definitions that are set up at the top of the makefile, but basically the structure of the imakefile is retained in the makefile. It would be impossible to generate a VMS command procedure from a typical imakefile.

The VICAR program `vimake` takes the concept of `imake` and goes one step further. Since the compile and link statements for all VICAR programs are quite similar, they do not need to be specified at all in the imakefile. In fact, `vimake` extracts *all* of the control out of the imakefile, leaving only C preprocessor commands.

> The VICAR imakefile contains only a description of *what* is to be built, not *how* to build it.

A VICAR imakefile consists only of C preprocessor statements, mostly #define's and a few #if's. These #define statements set up the filenames used in the program, and select various options on how to build the program. The `vimake` program uses this information and a system-specific template to create a build file for any system. Under VMS, it creates a DCL command file that will build the program. Under Unix, it creates a makefile instead. The makefile and DCL command file are never delivered with the program; only the imakefile is.

There are many advantages to this scheme. First of all, the programmer need only create one imakefile, instead of separate build files for each system. Second, it is much easier to create the imakefile than it would be to create a full-blown build file. Third, all build files are standardized, since they are created entirely by the `vimake` templates. They all operate the same, and can be fairly complex since the application programmer never modifies them directly. Fourth, locations of libraries and other files are standardized, since the imakefile doesn't specify where the library is, just what library it wants. Finally, changing the way applications are built is easy. Only the templates need to change, and all applications will be built the new way.

11.1.1 Creating and Using a VICAR Imakefile

Since `vimake` is based on the C preprocessor, every line in an imakefile will be a C preprocessor command. Most lines will be "#define", with some comments and "#if" statements on occasion. Comments are allowed, in the standard C style (enclosed in `/* ... */`).

It may be easier to start with an example. Here is the imakefile for the program `gen`:

```
#define PROGRAM gen

#define MODULE_LIST gen.c

#define MAIN_LANG_C
#define USES_C

#define R2LIB

#define LIB_RTL
#define LIB_TAE
```

The first two lines define macros with values. PROGRAM specifies that this is an application program, rather than a subroutine. It also specifies the name of the application. MODULE_LIST tells `vimake` what source code modules make up the application.

The rest of the lines simply define switches; there are no values associated with them. MAIN_LANG_C tells `vimake` that the main program is written in C (or ANSI C). USES_C says that some (non-ANSI) C is used in the application (that may sound redundant but is needed when there is more than one module with mixed languages). R2LIB says that this application goes in R2LIB. The LIB_RTL and LIB_TAE switches indicate which libraries to link with, in this case the RTL and TAE libraries.

Due to the internals of how `vimake` operates, there is unfortunately no real error checking. You have to be sure to spell the macros correctly, or they will simply be ignored. This may cause surprising results. If `vimake` is not operating the way you expect, first check to make sure that all the preprocessor macros are spelled correctly.

`vimake` can also handle some machine dependencies. All the macros defined in xvmaininc.h are available for use in #if tatements. This is typically used to compile a VMS-specific module only under VMS, and its Unix-specific counterpart only under Unix. See Section 6.4, Machine Dependencies, in the Porting C section, for details on what preprocessor macros are available from xvmaininc.h.

To use an imakefile, simply type the command `vimake` followed by the name of the imakefile. Note that imakefiles must have a ".imake" extension, but you should not give the extension on the `vimake` command. The command is the same under both VMS and Unix.

```
vimake gen
```

If you are running VMS, a file called "gen.bld" will be created (although it is a DCL COM file, a ".com" extension would confuse it with the packed application file). This file can be executed to compile and link the program:

```
$ @gen.bld
```

There are many options you can give on the command line to control the build, which are described in Section 11.1.3, Using the Generated VMS Build File.

If you are running Unix, a file called "gen.make" will be created when you run `vimake`. This file can be submitted to `make` to compile and link the program:

```
% make -f gen.make
```

There are many different build targets you can give to control the build process, which are described in Section 11.1.4, Using the Generated Unix Makefile.

The imakefile is actually composed of several parts. These parts are described below, with some examples. Although they can be in any order, the parts should generally follow the order listed below for consistency. For a complete list of valid #define's, see the next section.

- Type and name of program unit. The valid types are PROGRAM, SUBROUTINE, and PROCEDURE. One of these must be defined. The value for the definition is the name of the program or subroutine. The type of the program unit determines which of the other `vimake` features are available. There is currently not much to be done for a PROCEDURE, but an imakefile must still be present, for consistency. Documentation may still need to be built for procedures.

```
#define PROGRAM logmos

#define SUBROUTINE knuth

#define PROCEDURE midrarch
```

- List of modules and includes. The names of all source code modules must be defined (this does not apply to PROCEDUREs). MODULE_LIST should contain the names of all the compilable source code, with the appropriate extension. Note that Fortran modules must end in ".f", *not* ".for". Unix cares about such things. The modules are listed in the order they should be linked; i.e. the main module should be first (this can be overridden by LINK_LIST). INCLUDE_LIST should contain the names of all include files that are local to the program or subroutine, i.e. ones that are in the application COM file and not in p2$inc ($P2INC in Unix). They should also have the appropriate filename extension. Both lists should be space-separated lists of names. Don't use tabs, use spaces. All the names *must* be in lower case. If you run out of room on a line, the standard C preprocessor continuation character can be used, which is a backslash ("\") at the end of the line, or use continuation lists.

  Fortran system includes are a special case. They should be listed in the FTNINC_LIST macro, even though they are not a part of the program unit. See Section 7.2, Include Files, under Porting Fortran, for a description of which includes go in FTNINC_LIST. Note that local includes, which are part of the application, go in INCLUDE_LIST instead.

```
#define MODULE_LIST copy.f

#define MODULE_LIST logmos.c logmos_subs.c logmos_mosaic.c
#define INCLUDE_LIST logmos_defines.h logmos_structures.h logmos_globals.h

#if VMS_OS
#define MODULE_LIST amosids.f camosids.c amosufo_vms.mar
#define CLEAN_OTHER_LIST amosufo_unix.c
#else
#define MODULE_LIST amosids.f camosids.c amosufo_unix.c
#define CLEAN_OTHER_LIST amosufo_vms.mar
#endif

#define MODULE_LIST prog.f
#define INCLUDE_LIST myinc.fin
#define FTNINC_LIST errdefs sublib_inc

#define MODULE_LIST view.c utils.c image.c vdt.c plot.c host.c \
  overlap.c vprofile.c
```

  Note that VMS macro (.mar) and array processor files (.vfc) are allowed by `vimake`, but they are of course VMS specific. Also, when MODULE_LIST is defined differently for different machines, make sure the unused source code is listed in CLEAN_OTHER_LIST. This is so a clean-source operation can find all the source code to delete.

- Main language. If the application is of type PROGRAM, then the language the main program is written in must be defined. This is used by the Unix version of `vimake` to determine which command to use to link the program. The main program is defined as the one that the `main44` subroutine is written in. The languages that any other modules are written in do not matter.

```
#define MAIN_LANG_C

#define MAIN_LANG_FORTRAN
```

- Languages used. For every different language used, you must define a USES macro. If any modules are written in Kernighan and Ritchie (K&R) C (which is the case for most VICAR code), then define USES_C. If the modules are written in ANSI C, then define USES_ANSI_C. Note that USES_C and USES_ANSI_C are mutually exclusive. If any modules are written in Fortran, then define USES_FORTRAN. You must define both if you use both languages. USES_MACRO and USES_VFC are also available, but of course they are VMS-specific.

```
#define USES_C
#define USES_FORTRAN
#if VMS_OS
#define USES_MACRO
#endif
```

- Class of program unit. If you are working with a type PROGRAM, then you must define either R1LIB, R2LIB, R3LIB, or HWLIB to indicate the class of the program. If it is a portable SUBROUTINE, then define either P1_SUBLIB, P2_SUBLIB, P3_SUBLIB, or HW_SUBLIB to indicate the class. For unported, VMS-specific subrouines, you can define OLD_SUBLIB or OLD_SUBLIB3. (these are *only* to allow `vimake` to be used with the current system. Just because a subroutine has VMS-specific parts does not mean it goes in OLD_SUBLIB, as long as there are Unix-specifiic parts that do the same thing). The program class is used to select which library a subroutine goes in, to pick up the proper include directories, and to allow error checking on programs (e.g. a R2LIB program cannot use R3LIB subroutines), although the error checking is not yet implemented.

```
#define R2LIB

#define P3_SUBLIB
```

- Documentation files for the module. This applies to all types of modules, and is in fact the only real use most PROCEDURES have for the imakefile. Any documentation that requires building should go in this section. Currently, the only type implemented is a TAE error message file (.msg file), which is built using the `msgbld` program in TAE. Error message files are specified using TAE_ERRMSG. Other types of documentation will be implemented in the future. If no supported documentation types are supplied with the module, then do not define any of the documentation macros.

```
#define TAE_ERRMSG sffac
```

- Libraries needed for link. This mostly applies to PROGRAMs, although SUBROUTINE might need it on occasion. Every library you need to link to should be specified by defining the appropriate LIB macro. This normally includes LIB_RTL and LIB_TAE for most VICAR programs. The C run-time library is automatic, so you don't need to include it. The order is arbitrary, since they are only #define's, but you should try to keep them in order of highest-level to lowest-level, just to be consistent. For a complete list of all LIB macros currently defined, see the next section.

  It is quite likely, especially at first, that you will need a library that is not available via `vimake`. If so, contact the VICAR system programmer, and the library will be added. It is not hard or time-consuming to do. Although some mechanisms have been included to allow you to set up your own library names for testing, this is *highly* discouraged, and is system-dependent in any case. The VICAR system programmer needs to keep track of what libraries are in use, so they are not missed when VICAR is ported to a new system or delivered to an external site.

  Some of the libraries have _DEBUG forms, which link the program with the debuggable version of that library. These should be used only during testing. When the program is ready for delivery, it must not use any debuggable libraries.

  Some of the LIB macros also set up include directories for the C compiler. In unusual circumstances, you might need to include a LIB macro in a SUBROUTINE in order to pick up an include file.

```
#define LIB_P2SUB
#define LIB_MATH77
#define LIB_RTL
#define LIB_TAE
```

Here are some more examples of VICAR imakefiles. A simple C program, `gen`, was listed previously. Here is a simple Fortran program:

```
#define PROGRAM copy

#define MODULE_LIST copy.f

#define MAIN_LANG_FORTRAN
#define USES_FORTRAN

#define R2LIB

#define LIB_RTL
#define LIB_TAE
```

The following is an example subroutine that uses several different languages and has VMS-specific code. It illustrates how existing VMS macro code can be retained for efficiency while still having portable C code for other machines. Although the subroutine has not yet been ported, the example shows what the imakefile could look like when it does get ported.

```
#define SUBROUTINE amosids

#if VMS_OS
#define MODULE_LIST amosids.f camosids.c amosufo_vms.mar
#define CLEAN_OTHER_LIST amosufo_unix.c
#else
#define MODULE_LIST amosids.f camosids.c amosufo_unix.c
#define CLEAN_OTHER_LIST amosufo_vms.mar
#endif

#define P2_SUBLIB

#define USES_C
#define USES_FORTRAN
#if VMS_OS
#define USES_MACRO
#endif
```

The last example is for a more complex program, that has lots of modules, mixed languages, and uses several subroutine libraries. Again, the program has not yet been ported, but this is what the imakefile could look like.

```
/* C-style comments are okay if you really feel the need */

#define PROGRAM mgncorr

#define MODULE_LIST mgncorr.f mgncorr_fort.f mgncorr_support.c \
 mgncorr_correl.f mgncorr_interact.c mgncorr_graphics.c mgncorr_vdt.c \
 mgncorr_logs.c

#define MAIN_LANG_FORTRAN
#define USES_C
#define USES_FORTRAN

#define R2LIB

#define LIB_P2SUB
#define LIB_VRDI
#define LIB_MATH77
#define LIB_RTL
#define LIB_TAE
```

11.1.2 Valid vimake Commands

This section lists all the valid `vimake` macros. Although the order doesn't matter to `vimake` (since they are all preprocessor defines), you should keep things generally in the order presented here for consistency.

There are eight broad classes of macros, which are listed in separate sections below.

11.1.2.1 Module Type Macros

These macros define the type of the module. One and only one of these macros must appear. Note: Do not use a tab between the macro and the name on any of these; use only spaces. `make` can be quite sensitive about tab characters in the module type names.

- PROGRAM x — Specifies that this imakefile is for an executable program. The value x specifies the name of the program, which should be the same as the name of the COM file without the ".com" extension. One and only one of PROGRAM, SUBROUTINE, and PROCEDURE must be defined.

- SUBROUTINE x — Specifies that this imakefile is for a subroutine or subroutine package. The value x specifies the name of the subroutine, or of the collection of subroutines, which should be the same as the name of the COM file without the ".com" extension. One and only one of PROGRAM, SUBROUTINE, and PROCEDURE must be defined.

- PROCEDURE x — Specifies that this imakefile is for a TCL procedure. The value x specifies the name of the procedure, which should be the same as the name of the COM file without the ".com" extension. One and only one of PROGRAM, SUBROUTINE, and PROCEDURE must be defined.

11.1.2.2 Name List Macros

These macros list the files involved with this imakefile. MODULE_LIST must always be given for anything other than PROCEDURE. The others may or may not be needed.

- MODULE_LIST l — Specifies the list of source code modules. The argument l is a space-separated list of filenames, including the extensions. The extensions must match the language types in the supplied USES macros. All files must be in the current directory, i.e. no pathnames. The MODULE_LIST is used to specify the modules to compile, link, put in the library, and clean (delete after the build). Although some of those functions can be overridded with other lists, normally only MODULE_LIST will be used. MODULE_LIST must be defined, except for PROCEDUREs. MODULE_LIST can only handle about 150 to 200 characters (2.5 lines) worth of filenames. The limit is somewhat lower on VMS than on Unix. If you have more files than MODULE_LIST can handle by itself, use the list extension macros MODULE_LIST2, MODULE_LIST3, and MODULE_LIST4. You must define MODULE_LIST in any case; the extensions are also used if present. They should be used in order. Each of the extensions can handle about 200 characters. If you need more than three extensions, contact the VICAR system programmer to add more extensions.

- MODULE_LIST2 l — The first extension for MODULE_LIST if it is too big. See MODULE_LIST.

- MODULE_LIST3 l — The second extension for MODULE_LIST if MODULE_LIST2 fills up. See MODULE_LIST.

- MODULE_LIST4 l — The third extension for MODULE_LIST if MODULE_LIST3 fills up. See MODULE_LIST.

- INCLUDE_LIST *l* — The argument *l* is a space-separated list of local include files, if any. All includes delivered with the COM file must be listed here, with the appropriate filename extensions. If there are no local includes, then do not define INCLUDE_LIST. System includes, or includes from SUBLIB, should not be in the list. This list is used for the source cleaning operation and for makefile dependencies.

- FTNINC_LIST *l* — The argument *l* is a space-separated list of Fortran system and SUBLIB includes used in the Fortran modules, if any. All Fortran system and SUBLIB includes must be listed here, *without* any filename extensions. If there are no Fortran system or SUBLIB includes, then do not define FTNINC_LIST. This list is used to create logical names or symbolic links to the includes before the compile step.

- CLEAN_OBJ_LIST *l* — The argument *l* is a space-separated list of object files to delete (clean) after a compile. If not given, this list defaults to the value for MODULE_LIST, so CLEAN_OBJ_LIST should be rarely if ever be used. The names in CLEAN_OBJ_LIST must have *source code* filename extensions. `vimake` will convert them to standard object module names automatically. CLEAN_OBJ_LIST may be extended with CLEAN_OBJ_LIST2, CLEAN_OBJ_LIST3, and CLEAN_OBJ_LIST4 in the same way as MODULE_LIST. See MODULE_LIST for details.

- CLEAN_OBJ_LIST2 *l* — The first extension for CLEAN_OBJ_LIST if it is too big. See CLEAN_OBJ_LIST.

- CLEAN_OBJ_LIST3 *l* — The second extension for CLEAN_OBJ_LIST if CLEAN_OBJ_LIST2 fills up. See CLEAN_OBJ_LIST.

- CLEAN_OBJ_LIST4 *l* — The third extension for CLEAN_OBJ_LIST if CLEAN_OBJ_LIST3 fills up. See CLEAN_OBJ_LIST.

- CLEAN_SRC_LIST *l* — The argument *l* is a space-separated list of source code to delete during a clean-source operation. This happens during system builds. ince the source code just came from the COM file, it does not need to be kept after the build. If not given, this list defaults to the value for MODULE_LIST, so CLEAN_SRC_LIST should rarely if ever be used. CLEAN_SRC_LIST may be extended with CLEAN_SRC_LIST2, CLEAN_SRC_LIST3, and CLEAN_SRC_LIST4 in the same way as MODULE_LIST. See MODULE_LIST for details.

- CLEAN_SRC_LIST2 *l* — The first extension for CLEAN_SRC_LIST if it is too big. See CLEAN_SRC_LIST.

- CLEAN_SRC_LIST3 *l* — The second extension for CLEAN_SRC_LIST if CLEAN_SRC_LIST2 fills up. See CLEAN_SRC_LIST.

- CLEAN_SRC_LIST4 *l* — The third extension for CLEAN_SRC_LIST if CLEAN_SRC_LIST3 fills up. See CLEAN_SRC_LIST.

- CLEAN_OTHER_LIST *l* — The argument *l* is a space-separated list of other files to clean during a clean-source operation. After a clean-source is performed, only the files needed to execute the program (usually the program itself and the PDF) and the original COM file should still exist in the directory. If any files are unpacked or created

during the build process that are used only during the build process and that not covered by MODULE_LIST or INCLUDE_LIST (or automatically deleted as the .bld, .make, and .imake files are), then they should be listed in CLEAN_OTHER_LIST. These files are typically source code files that are used only on other machines, if MODULE_LIST or INCLUDE_LIST are defined conditionally. If there are no extra files (the usual case), then do not define CLEAN_OTHER_LIST. There are currently no extension macros defined for CLEAN_OTHER_LIST.

- LINK_LIST *l* — The argument *l* is a space-separated list of object code modules to link, in the order they will appear in the link statement. It is only valid for type PROGRAM. The object code modules must have a ".o" extension, even for VMS (it is converted to ".obj" automatically). If not given, this list defaults to the value for MODULE_LIST (which is automatically converted to object-name format), so LINK_LIST should rarely if ever be used. LINK_LIST may be extended with LINK_LIST2, LINK_LIST3, and LINK_LIST4 in the same way as MODULE_LIST. See MODULE_LIST for details.

- LINK_LIST2 *l* — The first extension for LINK_LIST if it is too big. See LINK_LIST.

- LINK_LIST3 *l* — The second extension for LINK_LIST if LINK_LIST2 fills up. See LINK_LIST.

- LINK_LIST4 *l* — The third extension for LINK_LIST if LINK_LIST3 fills up. See LINK_LIST.

- LIB_LIST *l* — The argument *l* is a space-separated list of object code modules to put in the object code library. It is only valid for type SUBROUTINE. The object code modules must have a ".o" extension, even for VMS (it is converted to ".obj" automatically). If not given, this list defaults to the value for MODULE_LIST (which is automatically converted to object-name format), so LIB_LIST should rarely if ever be used. LIB_LIST may be extended with LIB_LIST2, LIB_LIST3, and LIB_LIST4 in the same way as MODULE_LIST. See MODULE_LIST for details.

- LIB_LIST2 *l* — The first extension for LIB_LIST if it is too big. See LIB_LIST.

- LIB_LIST3 *l* — The second extension for LIB_LIST if LIB_LIST2 fills up. See LIB_LIST.

- LIB_LIST4 *l* — The third extension for LIB_LIST if LIB_LIST3 fills up. See LIB_LIST.

11.1.2.3 Main Language Macros

These macros define which language the main program is written in. One and only one of these must be present for type PROGRAM. They are not needed for any other type.

- MAIN_LANG_C — This flag, if defined, says that the main program is written in C. It is valid only for type PROGRAM. The main program is defined as the `main44()` subroutine for most VICAR programs, or `main()` if the program doesn't use the standard VICAR startup routines. For programs, one and only one of MAIN_LANG_C and MAIN_LANG_FORTRAN must be defined. Use MAIN_LANG_C for both K&R C and ANSI C.

- MAIN_LANG_FORTRAN — This flag, if defined, says that the main program is written in Fortran. It is valid only for type PROGRAM. The main program is defined as the `main44` subroutine for most VICAR programs, or the `main` subroutine if the program doesn't use the standard VICAR startup routines. For programs, one and only one of MAIN_LANG_C and MAIN_LANG_FORTRAN must be defined.

11.1.2.4 Languages Used Macros

These macros define what languages are used by source files in this imakefile. For anything other than PROCEDURE, at least one of these must be defined, and more than one is okay if needed.

- USES_C — This flag, if defined, says that at least one source module is written in C, using Kernighan and Ritchie (K&R) C, as opposed to ANSI C. Most VICAR code is written using K&R C, so this will be the flag used most often. USES_C may or may not accept some ANSI code, depending on the machine, but if there is any conflict then the K&R interpretation will be used. This flag is valid for types PROGRAM and SUBROUTINE. If any C code is to be compiled, either USES_C or USES_ANSI_C must be defined, but not both. Only one version of C may be used at a time, so all of the C modules in the imakefile must be either ANSI or K&R C. All C source modules must have a filename extension of ".c". Any number of other USES flags may be defined, if needed. All C modules have the RTL and TAE include files available. Other include directories can be set up via the LIB macros or the class macros (R2LIB, P2_SUBLIB, etc.).

- USES_ANSI_C — This flag, if defined, says that at least one source module is written in ANSI-standard C, as opposed to the older Kernighan and Ritchie (K&R) C. Most VICAR code is currently written in K&R C, although it is recommended that new code be written in ANSI C. See Section 6.5, ANSI C, for information on using ANSI C with VICAR. USES_ANSI_C may or may not accept non-ANSI constructs, depending on the machine implementation, but it should generate warnings if non-ANSI constructs are used. USES_ANSI_C is valid for types PROGRAM and SUBROUTINE. If any C code is to be compiled, either USES_C or USES_ANSI_C must be defined, but not both. Only one version of C may be used at a time, so all of the C modules in the imakefile must be either ANSI or K&R C. All C source modules must have a filename extension of ".c". Any number of other USES flags may be defined, if needed. All C modules have the RTL and TAE include files available. Other include directories can be set up via the LIB macros or the class macros (R2LIB, P2_SUBLIB, etc.). Note that main programs written in ANSI C still use the MAIN_LANG_C macro.

- USES_FORTRAN — This flag, if defined, says that at least one source module is written in Fortran. It is valid for types PROGRAM and SUBROUTINE. If any Fortran code is to be compiled, USES_FORTRAN must be defined. All Fortran source modules must have a filename extension of ".f". Any number of USES flags may be defined, if needed.

- USES_MACRO — This flag, if defined, says that at least one source module is written in VAX Macro. It is valid for types PROGRAM and SUBROUTINE. If any VAX Macro code is to be assembled, USES_MACRO must be defined. Of course, VAX Macro code is not portable, so in a portable application a portable version of the same function must

be available. Typically, MODULE_LIST will be defined differently for the VMS and Unix versions (see the examples in the previous section). All VAX Macro source modules must have a filename extension of ".mar". Any number of USES flags may be defined, if needed.

- USES_VFC — This flag, if defined, says that at least one source module is written in the VFC array processor language. It is valid for types PROGRAM and SUBROUTINE. If any VFC code is to be compiled, USES_VFC must be defined. At the present time, VFC code is not portable, so in a portable application a portable version of the same function must be available. Typically, MODULE_LIST will be defined differently for the VMS and Unix version. All VFC source modules must have a filename extension of ".vfc". Any number of USES flags may be defined, if needed.

11.1.2.5 Build Flag Macros

These macros set up various flags for the compilation or link steps. Define them as needed.

- FTN_STRING — This flag, if defined, indicates that one or more of the Fortran string conversion routines are called by any routine in the program unit. It is valid for types PROGRAM and SUBROUTINE. The Fortran string routines are **sc2for**, **sc2for_array**, **sfor2c**, **sfor2c_array**, **sfor2len**, and **sfor2ptr**. They are available only from C. Some machines (notably a Sun-4) require that modules using these routines be compiled with a lower level of optimization. If none of the string conversion routines are called, then do not define this flag. Technically, the flag must be defined if a C routine merely accepts a Fortran string, even if a subroutine ultimately calls the conversion routine. However, the same routine normally accepts the string and calls the conversion routine.

- C_OPTIONS x — The argument x defines extra options that are to be given to the C compiler. If no options are needed, do not define C_OPTIONS. It is valid only in conjunction with the USES_C or USES_ANSI_C flags. The argument x may be in any format the C compiler allows, and may include spaces. There is no translation performed on the options, so they are machine dependent. C_OPTIONS is intended for program development use only. A program or subroutine should rarely if ever be delivered with C_OPTIONS defined. If it is, the C_OPTIONS must be defined in a machine-dependent conditional.

- FORTRAN_OPTIONS x — The argument x defines extra options that are to be given to the Fortran compiler. If no options are needed, do not define FORTRAN_OPTIONS. It is valid only in conjunction with the USES_FORTRAN flag. The argument x may be in any format the Fortran compiler allows, and may include spaces. There is no translation performed on the options, so they are machine dependent. FORTRAN_OPTIONS is intended for program development use only. A program or subroutine should rarely if ever be delivered with FORTRAN_OPTIONS defined. If it is, the FORTRAN_OPTIONS must be defined in a machine-dependent conditional.

- LINK_OPTIONS x — The argument x defines extra options that are to be given to the linker. If no options are needed, do not define LINK_OPTIONS. It is valid only for type PROGRAM. The argument x may be in any format the linker allows, and may include spaces. There is no translation performed on the options, so they are machine dependent. Under VMS, the options are placed after any `vimake`-generated options; under Unix the options are placed at the beginning of the command line. LINK_OPTIONS is intended for

program development use only. A program or subroutine should rarely if ever be delivered with LINK_OPTIONS defined. If it is, the LINK_OPTIONS must be defined in a machine-dependent conditional.

- DEBUG — This flag, if defined, causes the makefile to be built for debugging. It is valid for PROGRAMs and SUBROUTINEs only. It is not needed under VMS, or on some versions of Unix (like the Sun). It is only needed for certain varieties of Unix that do not have conditional acros in their versions of `make`. For these machines, you must set DEBUG in the imakefile, and rerun `vimake`, to build a program for the debugger. For Unix machines that do have conditional macros, you can simply use the "debug" target in the standard generated makefile, so the DEBUG flag is not needed. For VMS, you can use the "DEBUG" secondary option in the standard build file, so again the DEBUG flag is not needed. A program or subroutine should never be delivered with DEBUG defined.

- PROFILE — This flag, if defined, causes the makefile to be built for profiling. It is valid for PROGRAMs and SUBROUTINEs only. It is not needed under VMS, or on some versions of Unix (like the Sun). It is only needed for certain varieties of Unix that do not have conditional macros in their versions of `make`. For these machines, you must set PROFILE in the imakefile, and rerun `vimake`, to build a program for the profiler. For Unix machines that do have conditional macros, you can simply use the "profile" target in the standard generated makefile, so the PROFILE flag is not needed. For VMS, you can use the "PROFILE" secondary option in the standard build file, so again the PROFILE flag is not needed. A program or subroutine should never be delivered with PROFILE defined.

11.1.2.6 Module Class Macros

These macros define what class the program is in. One and only one of these must be defined.

- R1LIB — This flag, if defined, says that the program is a class 1 VICAR application, and makes the class 1 SUBLIB includes available to the C compiler. It is valid for type PROGRAM only. One and only one of R1LIB, R2LIB, R3LIB, HWLIB, and TEST must be defined for a PROGRAM.

- R2LIB — This flag, if defined, says that the program is a class 2 VICAR application, and makes the class 2 SUBLIB includes available to the C compiler. It is valid for type PROGRAM only. One and only one of R1LIB, R2LIB, R3LIB, HWLIB, and TEST must be defined for a PROGRAM.

- R3LIB — This flag, if defined, says that the program is a class 3 VICAR application, and makes the class 2 and 3 SUBLIB includes available to the C compiler. It is valid for type PROGRAM only. One and only one of R1LIB, R2LIB, R3LIB, HWLIB, and TEST must be defined for a PROGRAM.

- HWLIB — This flag, if defined, says that the program is an HW application, specific to the Mars '94 project, and makes the HW includes available to the C compiler. It is valid for type PROGRAM only. One and only one of R1LIB, R2LIB, R3LIB, HWLIB, and TEST must be defined for a PROGRAM.

- TEST — This flag, if defined, says that the program is a test application. It is valid for type PROGRAM only, but will normally appear only in the test code for a SUBROUTINE. One and only one of R1LIB, R2LIB, R3LIB, HWLIB, and TEST must be defined for a PROGRAM.

- P1_SUBLIB — This flag, if defined, says that the subroutine is a class 1 portable VICAR subroutine, and makes the class 1 SUBLIB includes available to the compiler. It is valid for type SUBROUTINE only. One and only one of the *_SUBLIB macros must be defined for a SUBROUTINE.

- P2_SUBLIB — This flag, if defined, says that the subroutine is a class 2 portable VICAR subroutine, and makes the class 2 SUBLIB includes available to the compiler. It is valid for type SUBROUTINE only. One and only one of the *_SUBLIB macros must be defined for a SUBROUTINE.

- P3_SUBLIB — This flag, if defined, says that the subroutine is a class 3 portable VICAR subroutine, and makes the class 2 and 3 SUBLIB includes available to the compiler. It is valid for type SUBROUTINE only. One and only one of the *_SUBLIB macros must be defined for a SUBROUTINE.

- HW_SUBLIB — This flag, if defined, says that the subroutine is an HW subroutine, specific to the Mars '94 project, and makes the HW includes available to the compiler. It is valid for type SUBROUTINE only. One and only one of the *_SUBLIB macros must be defined for a SUBROUTINE.

- OLD_SUBLIB — This flag, if defined, says that the subroutine is a class 2 *unportable* VMS-specific VICAR subroutine. The subroutine may not be used with any portable applications. OLD_SUBLIB is valid for type SUBROUTINE only. One and only one of the *_SUBLIB macros must be defined for a SUBROUTINE.

- OLD_SUBLIB3 — This flag, if defined, says that the subroutine is a class 3 *unportable* VMS-specific VICAR subroutine. The subroutine may not be used with any portable applications. OLD_SUBLIB3 is valid for type SUBROUTINE only. One and only one of the *_SUBLIB macros must be defined for a SUBROUTINE.

11.1.2.7 Documentation Macros

These macros describe documentation that must be build. Use them if needed.

- TAE_ERRMSG x — The value x is the name of the TAE error message file, if present, without extension. This file provides help on error messages generated by the program, and is considered a "doc" file by the application packer program. The file must be named according to standard TAE conventions: "*name*fac.msg", where *name* is the facility part of the message key (the part before the "-"), usually the name of the program. The value x must not include the ".msg" extension. So, TAE_ERRMSG might be set to "sffac" or "bidrsfac". The message files are built with the TAE program `msgbld`.

11.1.2.8 Library Macros

These macros define the libraries needed for the compile and link steps. Almost all PROGRAMs and some SUBROUTINEs will define several of these. Define as many as are needed. They are not useful for type PROCEDURE. LOCAL_LIBRARY is somewhat special so it is listed first; the others are alphabetical.

- LOCAL_LIBRARY x — The argument x defines the name of the local library to use. It is defined in a system-specific manner, so it will have to be #if'd based on the operating system. The default if LOCAL_LIBRARY is not defined (the normal case) is "sublib.olb" for VMS and "sublib.a" for Unix. LOCAL_LIBRARY is used differently for PROGRAMs and SUBROUTINEs. For SUBROUTINEs, it is the name of the object library that the modules are inserted into during a non-system build (system builds go directly to the appropriate system library). This is typically used during development and debugging. For PROGRAMs, it is the name of the object library that is included in the link statement when the LIB_LOCAL flag is set. Again, it is used only during development and debugging. A program or subroutine should never be delivered with LOCAL_LIBRARY defined, since the program or subroutine must build in directories other than yours. LOCAL_LIBRARY is provided merely as a convenience during program development.

- LIB_CPLT — This flag, if defined, links the program to the Common Plotting Package library. It is valid only for type PROGRAM. The Common Plotting Package is currently available under VMS only.

- LIB_C_NOSHR — Normally the program is automatically linked to the C run-time library as a shared library or a VMS shareable image, and no flag need be given. The C_NOSHR flag, if defined, links the C run-time library as a standard link library instead. It is valid only for type PROGRAM. C_NOSHR should only rarely be used.

- LIB_C3JPEG — This flag, if defined, links the program to the library for the C Cubed JPEG decompression board. It is valid only for type PROGRAM. This flag also makes the C3JPEG library includes available to the C compiler. LIB_C3JPEG may be used from type SUBROUTINE for this purpose. Please note that this library is *not* available on all platforms. Currently, it is available only on Sun-4 platforms, as the library is hardware-dependent. If the library is not available (specified by the C3JPEG_AVAIL_OS flag defined in xvmaininc.h), then the LIB_C3JPEG flag is ignored.

- LIB_DTR — This flag, if defined, links the program to the VMS Datatrieve database shareable image. It is valid only for type PROGRAM. The DTR library is available under VMS only.

- LIB_FORTRAN — This flag, if defined, links the program to the Fortran run-time library. This flag should *only* be used if MAIN_LANG_C is set; the Fortran library is included automatically for MAIN_LANG_FORTRAN. The LIB_FORTRAN flag is needed only if you are linking a C main program to a subroutine written in Fortran that uses some I/O statements (such as WRITE to a string). This subroutine may be hidden in a library such as P2SUB, so if you get unexplained link errors, you might need this flag. It is important to note that the Fortran library is automatically included under VMS (where this flag is ignored), so you will not know if you need it unless you try linking the program on a Unix system. LIB_FORTRAN is valid only for type PROGRAM.

- LIB_FPS — This flag, if defined, links the program to the FPS routines for the VMS array processor. It is valid only for type PROGRAM. The FPS library is currently available under VMS only.
- LIB_HWSUB — This flag, if defined, links the program to the portable HW (Mars '94 specific) subroutine library. It is valid only for type PROGRAM. This flag also makes HW includes available to the C compiler. LIB_HWSUB may be used from type SUBROUTINE for this purpose.
- LIB_HWSUB_DEBUG — This flag, if defined, links the program to the debuggable version of the portable HW (Mars '94 specific) subroutine library. It is valid only for type PROGRAM. A program should never be delivered with HWSUB_DEBUG, as it is intended for program development and maintenance only. The HWSUB_DEBUG library has not yet been implemented. This flag also makes the HW includes available to the C compiler. LIB_HWSUB_DEBUG may be used from type SUBROUTINE for this purpose.
- LIB_LOCAL — This flag, if defined, links the program to the local library defined in LOCAL_LIBRARY (or the default local library). It is valid only for type PROGRAM. A program should never be delivered with LIB_LOCAL defined, since the program must link in directories other than yours. LIB_LOCAL is provided merely as a convenience during program development.
- LIB_MATH77 — This flag, if defined, links the program to the MATH77 mathematics subroutine library. It is valid only for type PROGRAM.
- LIB_MATRACOMP — This flag, if defined, links the program to the Matra compression library. This library is proprietary code supplied by Matra for use with the Mars '94 project. This flag is valid only for type PROGRAM (it is not needed for SUBROUTINE since there are no includes). Please note that this library is *not* available on all platforms. Currently, it is available only on Sun-4 platforms. If the library is not available, then the LIB_MATRACOMP flag is ignored. As a temporary measure, the C3JPEG_AVAIL_OS flag defined in xvmaininc.h may be used to determine availability of the Matra software. This will change in the future (when link groups for subroutines are implemented), so if you need to use C3JPEG_AVAIL_OS for the Matra library, please put in a big comment marking this usage as temporary.
- LIB_MDMS — This flag, if defined, links the program to the MDMS (Multimission Data Management Subsystem) client library. It is valid only for type PROGRAM. This flag also makes the MDMS includes available to the C compiler. LIB_MDMS may be used from type SUBROUTINE for this purpose.
- LIB_MDMS_FEI — This flag, if defined, links the program to the MDMS FEI (File Exchange Interface) library. It is valid only for type PROGRAM. This flag also makes the FEI includes available to the C compiler. LIB_MDMS_FEI may be used from type SUBROUTINE for this purpose.
- LIB_MOTIF — This flag, if defined, links the program to the X-windows and Motif libraries, specifically X11, Xt, and Xm. It is valid only for type PROGRAM. This flag also makes the X and Motif includes available to the C compiler. LIB_MOTIF may be used from type SUBROUTINE for this purpose.

- LIB_NETWORK — This flag, if defined, links the program to the network support libraries, including RPC's (Remote Procedure Calls) and sockets. It is valid only for type PROGRAM. This flag also makes the network includes available to the C compiler. LIB_NETWORK may be used from type SUBROUTINE for this purpose. Currently, the Multinet network support libraries for VMS require a different include syntax. For example, instead of "rpc/rpc.h" the syntax would simply be "rpc.h". A conditional compile should take care of this. It is hoped that this difference will be resolved in the future.

- LIB_NETWORK_NOSHR — This flag, if defined, links the program to the non-shareable version of the network support libraries. It is otherwise identical to LIB_NETWORK. LIB_NETWORK_NOSHR is needed if the PVM libraries are used.

- LIB_NIMSCAL — This flag, if defined, links the program to the NIMS calibration library. It is valid only for type PROGRAM. The NIMS calibration library is currently available under VMS only.

- LIB_PDS_LABEL — This flag, if defined, links the program to the PDS (Planetary Data System) label library. It is valid only for type PROGRAM. This flag also makes the PDS label library includes available to the C compiler. LIB_PDS_LABEL may be used from type SUBROUTINE for this purpose.

- LIB_PVM — This flag, if defined, links the program to the PVM (Parallel Virtual Machine) library. It is valid only for type PROGRAM. This flag also makes the PVM includes available to the C compiler. LIB_PVM may be used from type SUBROUTINE for this purpose.

- LIB_P1SUB — This flag, if defined, links the program to the portable class 1 SUBLIB library. It is valid only for type PROGRAM. This flag also makes the class 1 SUBLIB includes available to the C compiler. LIB_P1SUB may be used from type SUBROUTINE for this purpose.

- LIB_P1SUB_DEBUG — This flag, if defined, links the program to the debuggable version of the portable class 1 SUBLIB library. It is valid only for type PROGRAM. A program should never be delivered with P1SUB_DEBUG, as it is intended for program development and maintenance only. The P1SUB_DEBUG library has not yet been implemented. This flag also makes the class 1 SUBLIB includes available to the C compiler. LIB_P1SUB_DEBUG may be used from type SUBROUTINE for this purpose.

- LIB_P2SUB — This flag, if defined, links the program to the portable SUBLIB library. It is valid only for type PROGRAM. If you link to P2SUB, you may *not* link to S2 or S3. This flag also makes the class 2 SUBLIB includes available to the C compiler. LIB_P2SUB may be used from type SUBROUTINE for this purpose.

- LIB_P2SUB_DEBUG — This flag, if defined, links the program to the debuggable version of the portable SUBLIB library. It is valid only for type PROGRAM. If you link to P2SUB_DEBUG, you may *not* link to S2 or S3. A program should never be delivered with P2SUB_DEBUG, as it is intended for program development and maintenance only. The P2SUB_DEBUG library has not yet been implemented. This flag also makes the class 2 SUBLIB includes available to the C compiler. LIB_P2SUB_DEBUG may be used from type SUBROUTINE for this purpose.

- LIB_P3SUB — This flag, if defined, links the program to the portable class 3 SUBLIB library. It is valid only for type PROGRAM. If you link to P3SUB, you may *not* link to S2 or S3. This flag also makes the class 3 SUBLIB includes available to the C compiler. LIB_P3SUB may be used from type SUBROUTINE for this purpose.

- LIB_P3SUB_DEBUG — This flag, if defined, links the program to the debuggable version of the class 3 portable SUBLIB library. It is valid only for type PROGRAM. If you link to P3SUB_DEBUG, you may *not* link to S2 or S3. A program should never be delivered with P3SUB_DEBUG, as it is intended for program development and maintenance only. The P3SUB_DEBUG library has not yet been implemented. This flag also makes the class 3 SUBLIB includes available to the C compiler. LIB_P2SUB_DEBUG may be used from type SUBROUTINE for this purpose.

- LIB_RDM — This flag, if defined, links the program to the RDM library. It is valid only for type PROGRAM. RDM is currently available under VMS only, where it links as a shareable image.

- LIB_RTL — This flag, if defined, links the program to the VICAR Run-Time Library. It is valid only for type PROGRAM. Under VMS it links the RTL as a shareable image. Some versions of Unix support shared libraries, so RTL will link to the RTL shared library when it is implemented.

- LIB_RTL_DEBUG indexLIB_RTL_DEBUG — This flag, if defined, links the program to the debuggable version of the VICAR Run-Time Library. It is valid only for type PROGRAM. A program should never be delivered with RTL_DEBUG, as it is intended for program development and maintenance only.

- LIB_RTL_NOSHR indexLIB_RTL_NOSHR — This flag, if defined, links the program to the VICAR Run-Time Library as a standard link library, as opposed to a shared library or a VMS shareable image. It is valid only for type PROGRAM. RTL is preferred over RTL_NOSHR.

- LIB_SPICE — This flag, if defined, links the program to the SPICE subroutine library. It is valid only for type PROGRAM.

- LIB_SYBASE — This flag, if defined, links the program to the Sybase database client library. It is valid only for type PROGRAM. This flag also makes the Sybase includes available to the C compiler. LIB_SYBASE may be used from type SUBROUTINE for this purpose.

- LIB_S2 — This flag, if defined, links the program to the old, unportable, VMS-specific SUBLIB library. It is valid only for type PROGRAM. S2 is available under VMS only. If you link to S2, you may *not* link to P2SUB or P3SUB.

- LIB_S2_DEBUG — This flag, if defined, links the program to the debuggable version of the old, unportable, VMS-specific SUBLIB library. It is valid only for type PROGRAM. S2_DEBUG is available under VMS only. If you link to S2_DEBUG, you may *not* link to P2SUB or P3SUB. A program should never be delivered with S2_DEBUG, as it is intended for program development and maintenance only. The S2_DEBUG library has not yet been implemented.

- LIB_S3 — This flag, if defined, links the program to the old, unportable, VMS-specific class 3 SUBLIB library. It is valid only for type PROGRAM. S3 is available under VMS only. If you link to S3, you may *not* link to P2SUB or P3SUB.

- LIB_S3_DEBUG — This flag, if defined, links the program to the debuggable version of the old, unportable, VMS-specific class 3 SUBLIB library. It is valid only for type PROGRAM. S3_DEBUG is available under VMS only. If you link to S3_DEBUG, you may *not* link to P2SUB or P3SUB. A program should never be delivered with S3_DEBUG, as it is intended for program development and maintenance only. The S3_DEBUG library has not yet been implemented.

- LIB_TAE — This flag, if defined, links the program to the TAE object library. It is valid only for type PROGRAM. Under VMS it links TAE as a shareable image. Some versions of Unix support shared libraries, so TAE will link to the TAE shared library when it is implemented.

- LIB_TAE_NOSHR — This flag, if defined, links the program to the TAE object library as a standard link library, as opposed to a shared library or a VMS shareable image. It is valid only for type PROGRAM. TAE is preferred over TAE_NOSHR.

- LIB_VRDI — This flag, if defined, links the program to the Virtual Raster Display Interface (VRDI) library. It is valid only for type PROGRAM. Under VMS it links the VRDI as a shareable image. Some versions of Unix support shared libraries, so VRDI will link to the VRDI shared library when it is implemented. This flag also makes the VRDI includes available to the C compiler. LIB_VRDI may be used from type SUBROUTINE for this purpose.

- LIB_VRDI_DEBUG — This flag, if defined, links the program to the debuggable version of the VRDI library. It is valid only for type PROGRAM. A program should never be delivered with VRDI_DEBUG, as it is intended for program development and maintenance only. This flag also makes the VRDI includes available to the C compiler. LIB_VRDI_DEBUG may be used from type SUBROUTINE for this purpose.

- LIB_VRDI_NOSHR — This flag, if defined, links the program to the VRDI as a standard link library, as opposed to a shared library or a VMS shareable image. It is valid only for type PROGRAM. VRDI is preferred over VRDI_NOSHR. This flag also makes the VRDI includes available to the C compiler. LIB_VRDI_NOSHR may be used from type SUBROUTINE for this purpose.

11.1.3 Using the Generated VMS Build File

The generated VMS build file is quite powerful. This section describes the options you can use to control the build process.

The build files will look different depending on whether you are building a PROGRAM, SUBROUTINE, or PROCEDURE, and what languages you use. All options are described below. Some may not apply, depending on what you are building.

The build file is executed via the standard "`@`" command. The build file can have three arguments. The first is the primary option, the second is the secondary option list, and the third is a module list. All three parameters are optional. In the lists below, the capitalized letters are

required (although they may be in lower case on the command line), and the lower case letters are optional. So, most options can be abbreviated.

The primary options are:

- COMPile: Compiles all or a few source code modules. Valid for PROGRAM and SUBROUTINE only.
- DOC: Builds the documentation files for the unit. Currently, the only supported type of documentation file is a TAE error message file, although other types will be added. Valid for PROGRAM, SUBROUTINE, and PROCEDURE.
- LINK: Links the program. Valid for PROGRAM only.
- INSTall: Installs modules in the object library. Valid for SUBROUTINE only.
- STD: Builds a private version ("standard") of the unit in the current directory. It is the default of no primary option is given. It is valid for PROGRAM, SUBROUTINE, and PROCEDURE. For PROGRAM, it is equivalent to the COMPile and LINK steps together. For SUBROUTINE, it is equivalent to the COMPile and INSTall-LOCAL steps together. For PROCEDURE, it is essentially a no-op.
- ALL: Builds a private version of the unit in the current directory, including documentation. It is valid for PROGRAM, SUBROUTINE, and PROCEDURE. For PROGRAM, it is equivalent to the COMPile, DOC, and LINK steps together. For SUBROUTINE, it is equivalent to the COMPile, DOC, and INSTall-LOCAL steps together. For PROCEDURE, it performs the DOC step.
- SYStem: Performs a system build of the unit. This option should only be used by Configuration Management to build the VICAR system. It is valid for PROGRAM, SUBROUTINE, and PROCEDURE. For PROGRAM, it is equivalent to COMPile, DOC, LINK, CLEAN-OBJ, and CLEAN-SRC. For SUBROUTINE, it is equivalent to COMPile, DOC, INSTall-SYSTEM, CLEAN-OBJ, and CLEAN-SRC. For PROCEDURE, it is equivalent to DOC and CLEAN-SRC.
- CLEAN: Deletes and/or purges files that are used during the build but are not needed during program execution. It is valid for PROGRAM, SUBROUTINE, and PROCEDURE.

The default primary option is STD, which may be omitted. If so, then the secondary option list becomes the first argument to the BLD file, and the module list becomes the second argument.

The secondary option list is a list of options, separated by commas and with no blanks, that modify how the primary options are performed. They are associated with a primary option, so the corresponding primary must be given for the secondary to take effect (unless the primary is the default STD).

Note that some primaries, such as STD, ALL, and SYStem, are actually a combination of several other primaries. The secondary options apply in these cases as well. So, for example, the secondaries for COMPile and LINK may be given for STD.

There is one secondary that does not need a primary:

- NORMal: Used only as a placeholder if no secondary options are needed but the module list is. If you need to use the module list, but have no secondary options, you must give the secondary option NORMal. It should rarely be used.

The secondary options for the COMPile primary option are:

- DEBug: Compiles the source code for use with the debugger. The options "`/debug/noopt`" are given to the compiler (only "`/debug`" is given to Macro).
- PROfile: Compiles the source code for use with PCA, the Performance and Coverage Analyzer. The option "`/debug`" is given to the compiler.
- LISt: Generates a list file. The option "`/list`" is given to the compiler.
- LISTALL: Generates a full list. The option "`/show=all`" is given to the compiler. LISTALL implies LIST, so you need not give both.
- LISTXREF: Generates a cross reference listing. The option "`/cross_ref`" is given to the compiler. LISTXREF implies LIST, so you need not give both.
- LINT: Runs the `lint` syntax checker for C. This option is not currently implemented.

The secondary options for the LINK primary option are:

- DEBug: Links the code for use with the debugger. The option "`/debug`" is given to the linker.
- PROfile: Links the code for use with PCA, the Performance and Coverage Analyzer. The option "`/debug=sys$library:pca$obj.obj`" is given to the linker.
- MAP: Creates a link map file. The option "`/map`" is given to the linker.
- MAPALL: Creates a full link map file. The option "`/full`" is given to the linker. MAPALL implies MAP, so you need not give both.
- MAPXREF: Includes a cross reference listing in the map file. The option "`/cross_ref`" is given to the linker. MAPXREF implies MAP, so you need not give both.

The secondary options for the INSTall primary option are:

- LOCAL: Installs the object code in the local (private) library. This is the default. The name of the library may be specified with the LOCAL_LIBRARY macro in the imakefile.
- SYSTEM: Installs the object code in the VICAR system library. This option may only be used by Configuration Management.

The secondary options for the CLEAN primary option are:

- OBJ: Deletes object and list files. For PROGRAMs, it purges the executable. This is the default. It is valid only for PROGRAM and SUBROUTINE.
- SRC: Deletes the source code, imakefile, and BLD files. Be very careful with this option! It is intended mainly for system builds, where the source code can be deleted after the build because it is maintained in the COM file. If you are modifying code and do not have an up-to-date COM file, then do *not* use the CLEAN-SRC option.

The secondary options for the DOC primary option are:

- MSG: Builds the TAE error message file. If no secondary options for DOC are given, then all documentation is built. If a secondary option is present, then only the types given in the secondaries are built.

The last parameter to the BLD file is the module list. It is a list of modules to compile or clean. Normally, the entire application is built at once, so this is not often used. However, the capability is there to build only some of the code. This is useful if you are modifying one module of a large program. Once everything is compiled, you need only compile the module you are changing. The names given in the module list must match exactly with the names in the MODULE_LIST macro in the imakefile. If you want to give more than one module name, then separate them with spaces and enclose the whole list in double quotes.

Some examples may prove helpful. The first example merely compiles a version of the program into the local directory:

```
@prog.bld
```

The next example is the same, except the documentation (if present) is built as well:

```
@prog.bld all
```

This example shows building the program for use in the debugger. Note how the secondary option is first because STD was defaulted:

```
@prog.bld debug
```

The next example shows recompiling a single module out of a large application, and then relinking it with the debugger. A link map is created. The other modules must have already been compiled:

```
@prog.bld comp deb module.c
@prog.bld link deb,map
```

The last example shows shows how to obtain a full compile listing with cross-reference from a pair of modules:

```
@prog.bld comp listall,listxref "module1.c module2.f"
```

11.1.4 Using the Generated Unix Makefile

The generated Unix makefile is a fairly standard makefile. Since most VICAR programmers are not familiar with `make` , this section briefly describes how to use it. For more details on using `make`, see the documentation for it.

In a nutshell, a makefile describes the dependencies between parts of an application, and how to build those parts to create the output program. Dependencies are simply the files that are needed to build a piece of the program. For example, the executable depends on all the object files and the link libraries it needs. Each object file depends on its corresponding source file, and the include files that it uses. A link library depends on the object files that make it up.

Source code may depend on a preprocessor, such as `lex` or `yacc`, or may come from a source code management system. All these dependencies can be tracked by `make`. `make` is smart enough to figure out what needs to be built based on what has changed, and to build only those parts. So, if you modify only one source module, `make` checks the modification dates, realizes only one module has changed, and recompiles only that module.

The `vimake`-generated makefile takes advantage of many of these features. It does not allow specifying which include files are used by which source files, but since VICAR applications are typically fairly small, this should not cause a problem. It may simply mean a few extra compiles. Full automatic dependency checking may be added to the makefile in the future.

By default, `make` looks for several filenames for the makefile, including "Makefile" and "makefile". However, the generated makefile is named "*file*.make" where *file* is the name of the application. Therefore, you must give the "`-f`" option to `make` to specify the name of the makefile. If you are doing lots of development on one program, you may rename the file to be "Makefile" (or better yet, create a symbolic link) so `make` will find it automatically, but this precludes having more than one application in the same directory, so it should not be done all the time. To run `make`, use the following syntax:

```
make -f file.make targets
```

where "file" is the name of the application and "targets" is an optional list of targets to build (described below).

`make` operates through the use of targets. A target is the final result of the `make`. A target could be an object file name, in which case that file would get compiled. It could be the executable name, in which case all the compiles and links necessary to create the executable are performed. There are also special targets, such as "all" or "clean.src", that cause certain actions to be performed. More than one target may be given on the same command, separated by spaces.

The targets available depend on whether you are building a PROGRAM, SUBROUTINE, or PROCEDURE, and on how complex the build is. The allowed targets are listed below. Not all targets will be available in all generated makefiles.

It is important to realize that there are fewer options on the makefile than there are on the VMS BLD file. This is largely due to the fact that `make` takes care of a lot of the details for you. You generally don't need to specify compiling a single module, or only doing the link, because `make` will determine what needs to be done and do it for you.

- std: Builds a private version ("standard") of the unit in the current directory. It is the default target if none is given. It is valid for PROGRAM, SUBROUTINE, and PROCEDURE. For type SUBROUTINE, the modules are installed in the local link library (see library.local).

- all: Builds a private version of the unit in the current directory, including the documentation. It is valid for PROGRAM, SUBROUTINE, and PROCEDURE. For type SUBROUTINE, the modules are installed in the local link library (see library.local).

- debug: Builds a debuggable version of the unit ("`-g`" option to the compiler). This target is only available on machines with conditional macros in `make`, such as a Sun. On machines without conditional macros, debug is not a valid target. To build the program for debug on these machines, you must set the DEBUG flag in the imakefile, and re-run `vimake`. When available, the debug target is valid for PROGRAM and SUBROUTINE.

- profile: Builds a version of the unit for use with the profiler ("`-pg`" option to the compiler). This target is only available on machines with conditional macros in `make`, such as a Sun. On machines without conditional macros, profile is not a valid target. To build the program for use with the profiler on these machines, you must set the PROFILE flag in the imakefile, and re-run `vimake`. When available, the profile target is valid for PROGRAM and SUBROUTINE.

- system: Performs a system build of the unit. This target should only be used by Configuration Management to build the VICAR system. It is valid for PROGRAM, SUBROUTINE, and PROCEDURE. Note that the system target performs a clean.src operation, which will delete the source code (since it just came from the COM file, it's not needed any more).

- compile: Compiles all the source code, but does not link or install the objects in a library. It is valid for PROGRAM and SUBROUTINE.

- *Object module name*: The name of an object module (with ".o" extension) may be given to compile only one module. This is valid for PROGRAM and SUBROUTINE. Specifying an object module name is not particularly useful, since `make` already determines which modules need to be compiled, so this option will rarely be used.

- *Executable name*: The name of the program itself may be used as a target, for type PROGRAM only. It is equivalent to the target "std".

- library.local: Installs the object code in the local (private) library. The name of the library may be specified with the LOCAL_LIBRARY macro in the imakefile. The modules are compiled if needed first. This target is valid for SUBROUTINE only.

- library.system: Installs the object code in the VICAR system library. This option may only be used by Configuration Management. The modules are compiled if needed first. This target is valid for SUBROUTINE only.

- clean.obj: Deletes all object code for this unit. It is valid only for PROGRAM and SUBROUTINE.

- clean.src: Deletes the source code, imakefile, and makefiles. Be very careful with this target! It is intended mainly for system builds, where the source code can be deleted after the build because it is maintainted in the COM file. If you are modifying code and do not have an up-to-date COM file, then do *not* use the clean.src target.

- doc: Builds all the documentation files for the unit. Currently, the only supported type of documentation file is a TAE error message file, although other types will be added. Valid for PROGRAM, SUBROUTINE, and PROCEDURE.

- doc.errmsg: Builds the TAE error message file.

Some examples may prove helpful. The first example merely compiles a version of the program into the local directory:

```
make -f prog.make
```

The next example is the same, except the documentation (if present) is built as well:

```
make -f prog.make all
```

This example shows building the program for use in the debugger, for systems that support the "debug" target.

```
make -f prog.make debug
```

The last example shows how to delete the object modules after a build. This is not recommended during most program development, because it forces all the modules to be recompiled every time. However, it can be useful to force a recompile of everything on occasion.

```
make -f prog.make clean.obj
```

11.2 Application Packer

A typical VICAR application program, procedure, or SUBLIB subroutine is made up of several different files. These files include source code, include files, PDFs, imakefiles, test files, documentation files, and others. In order to better manage all these files, they are packed into one file so they can be treated as a unit. These files are called COM files due to the ".com" filename extension.

Previously, there were two types of COM files. The first was hand-built, and simply included each module after a DCL `create` command. The build procedure was included as an integral part of this COM file. The second was created by the program `mipl_arch` (which was never delivered). `mipl_arch` took the individual files and put them in the COM file in a form suitable for a stream editor, but they were hard for people to read. It also included a checksum on each file so the COM file could not be directly edited. It was slow, and included a large amount of DCL code so the file would be self-unpacking.

There is now a third scheme for generating COM files that combines the best of both systems, and creates files that can be packed and unpacked under both VMS and Unix. The build procedure is a separate file, the COM file is automatically created, and the COM files are standardized as in `mipl_arch`. But, the COM file is easy for people to read, can be edited without unpacking, has minimal header information, and is fast, like the hand COM files. These new COM files are created and used via a pair of programs called `vpack` and `vunpack` (the "V" is for "VICAR").

`vpack` and `vunpack` are complementary, cross-platform programs which are designed to pack and unpack application COM files. A COM file typically includes source code, an imakefile, and a PDF file. The COM file may also include test files and other, application-specific files. Under VMS, the COM file is self-extracting and self-executing (i.e., automatically compiling and linking the code), as the previous COM files were. Under Unix, the `vunpack` program is needed to extract the files from the COM file.

11.2.1 `vpack`

The `vpack` program accepts a list of parameters as follows:

```
vpack <file.com>  [-u]
                  [-s Source file(s)]
                  [-i IMAKE template file(s)]
                  [-b VMS build file(s)]
```

```
[-m UNIX make file(s)]
[-p PDF file(s)]
[-t Test file(s)]
[-d Documentation file(s)]
[-o "Other" file(s)]
```

The "`-b`" (VMS build file) and "`-m`" (Unix makefile) should not be used. The "`-i`" (imakefile) option should be used instead. Machine-specific build files are allowed only under very unusual circumstances.

The "`-u`" option tells `vpack` that the module is unportable (VMS-specific), so the correct header information can be generated (the default for executing the COM file directly with no arguments is changed from "STD" to "SYS"). If `vpack` is used with unportable modules, "`-u`" must be present; for portable modules, it must not be.

If you call the `vpack` program with no parameters, the program will tell you that you have made a syntax error and will tell you the proper syntax to use.

Based on the parameters provided, the `vpack` program will create the new COM file (named in the first argument), read the lists of files in the order given, and append them to the new COM file. None of the source files is altered or deleted. The output of the program is the new COM file, which is an ASCII file. The `vpack` program requires that the COM filename be included as a parameter and also requires that at least one list of files be included.

As an example, the command necessary to assemble the Magellan program `view` into a COM file is:

```
vpack view.com -s view.h host.c image.c overlap.c view.c
               -i view.imake
               -p view.pdf
               -t tstview.pdf tstview.scr
```

You will note that an include file (view.h) is considered a "source" file. Multiple file names can be separated by commas or spaces or both. Of course, if you spread the command over several lines like that, you would need to use a continuation character appropriate for the operating system you are using.

The `vpack` program can be executed in three different ways. The command can be executed directly from the command line as shown above, assuming the command line is long enough to hold the entire command. It can also be executed from a repack file created by the `vpack` program. For example, the repack file created for the `view` program is as follows:

```
$ vpack view.com -
        -s view.h host.c image.c overlap.c view.c -
        -i view.imake -
        -p view.pdf -
        -t tstview.pdf tstview.scr
$ Exit
```

Under VMS only, this file can be executed directly by typing:

```
@view.repack
```

The operating system will read the file and execute it exactly as if you had typed the command from the command line. The same limit on the length of a command line applies in this case.

The third method for executing the `vpack` program also uses the repack file, but uses it as a parameter to the program. Type:

```
vpack view.repack
```

to execute the program. This is the preferred method, and is required under Unix since DCL COM files cannot be executed directly. There is no command line length limit with this method.

The `vpack` program creates a COM file in the following format:

- A standard header which lists all of the available parameters and options for the COM file in the VMS environment as comments. This header also lists the version number of the `vpack` program which created the file and the name of the file. See below for an example of a typical header.
- DCL code which parses command-line options in the VMS environment and sets up the necessary conditions to carry out the user's instructions. For example, if the user typed "`@file.com source`", this section would set the DCL variables to create the source files from the COM file.
- A repack file which can be used by the `vpack` program or can be used stand-alone (under VMS only) to repack the COM file after making changes to the files which comprise it.
- The various sections of files which make up the COM file, e.g., source file(s), PDF file(s), etc. Each section of files consists of a label header (e.g., "$Source_File:"), followed by the files within the section.

If the COM file includes an imakefile, for use with `vimake`, the COM file header options will include many of the options which can be utilized by the generated build file (e.g., COMPILE, ALL, SYSTEM, CLEAN, etc.). The `vpack` program is intelligent enough to adjust the header options based on the type of files included. If only source files are assembled into the COM file, then the only options listed in the header will be those which unpack the various files — such options as compiling and linking will not be included. If the COM file does not include a PDF file, then the option to unpack the PDF file is omitted from the COM file, and so on.

A typical header, once again for the `view` program, is shown below. Since the `view` program includes source files, a PDF file, test files, and an imakefile, all of these options are listed in the header file. The various options for the COM file are only available under VMS. The options are discussed in greater detail below. For further information on the use of `vimake` and the build options, see Section 11.1, `vimake`.

```
$!****************************************************************************
$!
$! Compile+link proc for MIPL module view
$! VPACK Version 1.4, Thursday, June 25, 1992, 09:41:13
$!
$! Execute by entering: $ @view
$!
$! The primary option controls how much is to be built.  It must be in
```

```
$! the first parameter.  Only the capitalized letters below are necessary.
$!
$! Primary options are:
$!   COMPile     Compile the program modules
$!   ALL         Build a private version, and unpack the PDF and DOC files.
$!   STD         Build a private version, and unpack the PDF file(s).
$!   SYStem     Build the system version with the CLEAN option, and
$!               unpack the PDF and DOC files.
$!   CLEAN       Clean (delete/purge) parts of the code, see secondary options
$!   UNPACK      All files are created.
$!   REPACK      Only the repack file is created.
$!   SOURCE      Only the source files are created.
$!   SORC        Only the source files are created.
$!               (This parameter is left in for backward compatibility).
$!   PDF         Only the PDF file is created.
$!   TEST        Only the test files are created.\index{TEST}
$!   IMAKE       Only the IMAKE file (used with the VIMAKE program) is created.
$!   DOC         Only the documentation files are created.
$!
$!   The default is to use the STD parameter if none is provided.
$!
$!******************************************************************************
$!
$! The secondary options modify how the primary option is performed.
$! Note that secondary options apply to particular primary options,
$! listed below.  If more than one secondary is desired, separate them by
$! commas so the entire list is in a single parameter.
$!
$! Secondary options are:
$! COMPile,ALL:
$!   DEBug      Compile for debug               (/debug/noopt)
$!   PROfile    Compile for PCA                 (/debug)
$!   LISt       Generate a list file            (/list)
$!   LISTALL    Generate a full list            (/show=all)   (implies LIST)
$! CLEAN:
$!   OBJ        Delete object and list files, and purge executable (default)
$!   SRC        Delete source and make files
$!
$!******************************************************************************
$!
$ write sys$output "*** module view ***"
```

As was stated above, each of the following options is only available when the COM file is executed under VMS. To unpack files from the COM file in the UNIX environment, the companion program `vunpack` must be used.

It should be noted that many of these options pertaining to the build process will not typically be used. The preferred way of modifying an application is to unpack it, change it, and repack

the results when it's ready for delivery. The same options are available on the `vimake`-generated BLD file. However, the COM file may be edited directly without unpacking, in which case these options could be useful.

- COMPile: The compile option is only available when the program's files include an imakefile created for use with the `vimake` utility. This option causes the COM file to create the source file(s) and the imakefile, if necessary. `vimake` is called to create the appropriate VMS BLD file. The COM file then executes the BLD file to carry out the compile command. This option also has several secondary options:
  - DEBug: Compile for debug (`/debug/noopt`)
  - PROfile: Compile for PCA (`/debug`)
  - LISt: Generate a list file (`/list`)
  - LISTALL: Generate a full list (`/show=all`) (implies LIST)

  If, for example, you wish to compile the modules for debugging, the appropriate syntax for executing the COM file would be:

  ```
  @view comp deb
  ```

  If you wanted to generate a list file as well, you would type:

  ```
  @view comp deb,list
  ```

  Notice that the secondary options are separated by commas, with no intervening spaces.

  This command would generate debuggable versions of the object file(s) for the program. To create the executable, you would still need to call the BLD file with the "LINK DEBUG" parameters.

- ALL: This option builds a private version of the executable in the default directory. By default, no secondary options are used. Since the ALL option carries out the COMPILE command, it accepts the same secondary options described above. The ALL option also unpacks the PDF and documentation files and calls the BLD file to carry out any additional processing of the documentation files when necessary.

- STD: This option carries out the same actions as the ALL command, with the exception that documentation files are not unpacked or generated. STD is the default option for portable modules (ones without the "`-u`" flag on the `vpack` command).

- SYStem: This option builds the system version of the executable and executes the CLEAN option as well. None of the secondary options are activated for the COMPILE and LINK commands, hile the CLEAN command has both OBJ and SRC secondary options activated. Additionally, the PDF and documentation files are unpacked and any additional processing of the documentation files needed is carried out. This command should only be executed by Configuration Management. This option is the default (for compatibility reasons) if the "`-u`" (unportable) flag is given to `vpack`.

- CLEAN: This option deletes and/or purges files from the disk, depending on which secondary options are selected. If the OBJ option is selected, the command deletes any object and list files and purges the executable. This is the default secondary option for the CLEAN command. If the SRC option is selected, the source, imakefile, and build file are deleted. Be careful with the SRC option; make sure the COM file is up to date before deleting the files that make it up.

The remaining COM file options all have to do with which files are unpacked from the COM file. This gives you complete control over which files are removed from the COM file. When the files are unpacked from the COM file, the COM file itself remains unaltered.

- UNPACK: All files are created.
- REPACK: Only the repack file is created.
- SOURCE: Only the source files are created.
- PDF: only the PDF file is created.
- TEST: only the test files are created.
- DOC: only the documentation files are created.
- IMAKE: only the imakefile (used with the `vimake` program) is created.

Since the COM file created by `vpack` is an ASCII file, it is editable by the user. It is recommended, however, that you follow the normal procedure of unpacking the file(s), making and testing the necessary modifications, then using `vpack` to rebuild the COM file.

11.2.2 `vunpack`

The `vunpack` program does precisely what its name suggests — it unpacks files from a COM file created by `vpack`. Its primary use is on systems other than VMS, where the COM file is not self-extracting. It may of course be used under VMS also.

The `vunpack` program accepts a list of parameters as follows:

```
vunpack <file.com>  [{source|pdf|imake|build|make|
                      test|repack|doc|std|system|unpack|all}]
```

or

```
vunpack <file.com> -f file.1 file.2 file.3
```

As in the `vpack` program, if you call the `vunpack` program with no parameters, the program will tell you that you made an error and will tell you the proper syntax to use.

The first parameter tells `vunpack` which COM file to extract files from, and the remaining (optional) parameters tell the program which file(s) to extract. You may extract more than one type of file, if desired, by putting more than one type on the command line, separated by spaces. If no parameters other than the COM file name are provided, the program unpacks all the files in the COM file.

The `vunpack` program will not automatically compile and link the program, but it can extract the imakefile so that you can create the BLD file (VMS) or makefile (Unix) which can create the executable for you. See Section 11.1, `vimake`, for details.

The `vunpack` program also allows you to extract a list of specified files from the COM file. Instead of unpacking groups of files, by specifying "`-f`" and supplying a list of one or more file names, you can have `vunpack` unpack only the file(s) you need.

For example, if only host.c was needed from the view.com file, you would type the following:

```
vunpack view.com -f host.c
```

However, if you need all of the source files and the imakefile from VIEW.COM, you would type the following:

```
vunpack view.com source imake
```

The `vunpack` program will extract files from the COM file and put them in the current directory, but it will not alter the COM file in any way.

11.3 Test Routines

Test routines are required for every VICAR class 2 (R2LIB) application and subroutine. Test routines are largely the same as before, but there are a few differences.

As before, programs and procedures typically have a procedure PDF as the test routine. This PDF must be portable as well. See Section 8, Porting TCL, for details on making a PDF portable.

Subroutines must have a test program that calls the subroutine, as well as a procedure PDF that calls the test program. The test program must be portable as well, and should have its own imakefile and its own PDF. So subroutines often have four test files: the test program, the imakefile, the test program PDF, and the test PDF that calls it. All of these files go under the TEST file category in `vpack`. To run the test, you would first unpack just the TEST files, run `vimake` on the test imakefile, build the program, and finally execute the test PDF.

Keep in mind that the test programs and scripts should test all of the major functions of the program, rather than just testing that the program doesn't crash. This is not a change, but it is a point that needs emphasizing.

The log that's generated when the test is run must in most cases be delivered with the program. This ensures that you run the test before delivery, in order to make sure it still works. You should really run `diff` on the new log versus the old log, to make sure nothing changed. In any case, Test & Integration will have to compare the logs.

One thing that really makes checking the logs difficult is the `usage` statistics that TAE automatically puts in the log after each program execution. The date, amount of time, page faults, etc. used will vary every time the test is run, and yet make no difference in the results of the regression test. The usage statistics generate lots of meaningless differences, which tend to hide any real differences.

For this reason, the TAE global variable $AUTOUSAGE has been added to control the automatic printing of usage statistics. It can take three values, "BATCH", "ALL", and "NONE". To use it, you must use the TAE command "`refgbl $AUTOUSAGE`" before the "body" statement, then you can set it with a standard "let" command. The values are described below.

- BATCH: This is the default, and the way previous versions of VICAR worked all the time. In this mode, automatic usage statistics are printed after every program execution in batch mode. In interactive mode, usage statistics are not printed unless the `usage` command is explicitly given.
- ALL: This causes automatic usage statistics to be printed after *every* program execution, in both batch and interactive modes.
- NONE: This supresses automatic usage statistics in both batch and interactive modes. With "NONE" set, the only way to get the statistics is via the `usage` command.

It is recommended that $AUTOUSAGE be set to "NONE" in every test script, to avoid meaningless usage statistics.

Another problem in the checking of test scripts is the issue of numerical precision. It is possible that the results of a test involving floating-point calculations may be different on different machines in the least significant few digits. This is due to differences in numerical representation and precision, as well as possible differences in library routines. The acceptable difference between the results is defined in the *MSTP Software Requirements Document*, by Steve Pohorsky, JPL D-10637.

12 VICAR Directory Structure

The directory structure of VICAR is now quite different from the old directory structure. Directories have been generally rearranged to provide a more hierarchical and consistent directory tree. The directory tree is shown diagramatically in Figure 1, on the next page.

It is important to note that while directories have moved all over the place, the VMS logical names that point to the directories are in general the same (there have been a few deletions and a lot of additions, but most of the useful old ones are still there). So, as long as you use logical names (such as "v2$source" for the RTL source), like you are supposed to, you won't notice the directory changes. Under Unix, environment variables will point at the various directories instead of logical names.

So, although the directory tree changes will not affect most users, they are mentioned below for completeness.

Every subsystem within VICAR is now isolated in its own directory tree. For example, all the RTL source and includes are under the "rtl" directory. All source, includes, fonts, and programs for the VRDI are under the "vrdi" directory. All source, includes, subroutines, and executables for class 2 applications are under the "p2" directory. This makes it clearer where a module stands in the VICAR hierarchy.

The new directory structure is designed to allow several different computer systems to use the same copy of VICAR. All machine-specific object code and executables are isolated in directories bearing the name of the machine they belong to. For example, the class 2 application executables (r2lib) for a Sun-4 system reside in p2/lib/sun-4, while for an HP 700 they would be in p2/lib/hp-700. The different machines access the same directory tree using NFS (Network File System), but still have machine-specific compiled code where needed. These machine-specific directories are represented by the "targets..." boxes on the directory chart.

The directory tree has been renamed, as shown in the diagram. The top level used to be called "mipl" or some derivative (such as "mipl060"). Now, it is called "vicar" or some derivative (such as "vicar120").

Everything that is not directly a part of VICAR has been removed from the VICAR source tree. The old Development system directory tree contained many files that had nothing to do with the running VICAR system, such as CMS libraries, old build logs, delivery audit trails, and test images. These have all been eliminated from the VICAR tree, and have their own directories (actually most of them are still in the old "mipl" directory tree).

The fetch and build procedures have been cleaned up and are now entirely separate operations. The fetch routine gets everything from the code management system (CMS) at once, and then the build routine compiles and links everything as a separate operation. This makes building VICAR at foreign sites much easier. The same philosophy holds under both VMS and Unix — fill all the directories at once, then build everything. The fetch files are all in the utilities directory, while the build files are in the top-level directories of the subsystems they build.

There are two directories that are created entirely during the build process. They fall outside the subsystem hierarchy, since they contain files from many different places in the directory tree. These files are grouped into these directories for convenience. Since they are created during the build, and contain no source code, being outside the hierarchy does not present a significant problem. The first is the "olb" directory, which contains all the object (link) libraries in the VICAR system. The second is the "lib" directory, which contains miscellaneous executables and TAE PDFs from various parts of the VICAR system.

Figure 1: VICAR Directory Tree

13 Application Examples

There has been a lot of information presented in this porting guide. Perhaps the best way to digest it all is to look at examples. Rather than printing an example here, it is easier to point you at good on-line examples and let you look at them there, or print them, at your leisure.

All ported applications are kept on the MIPL VAXcluster in the CMS libraries p2prog$cms (applications) and p2sub$cms (subroutines). P3prog$cms and p3sub$cms are the class 3 equivalents. You can use these libraries to determine which applications have been ported to study them.

On a Unix system with VICAR, obviously all applications in the source directories have been ported, or they wouldn't be there.

The question arises, what applications should you look at? There are now plenty of programs and subroutines that have been ported that can be used for reference. On a Unix system, look in $P2SUB for subroutines, and $P2SOURCE for programs. On a VMS system, look in p2$sub for subroutines. Programs are in p2$source, but so are all the unportable programs. You can either consult the p2prog$cms CMS library, or refer to a Unix system, to figure out which programs are portable.

It is recommended that everyone doing a port examine these applications and subroutines as examples of how to write portable code, and how to package everything together into a COM file.

In addition, the RTL source code itself can be used as examples of how to write Fortran bridges and machine-specific code. The routines that accept keyword-value pairs as arguments are a special case, but the other routines follow the same Fortran bridge rules as a SUBLIB routine must. And the RTL has a lot of machine-dependent code in it to use as examples.

14 Summary of Major Portability Rules

This section contains a brief summary of the major portability rules spelled out in this document, and a reference to the sections where the rules are discussed in more detail. *This is not a complete list and must not be used as a substitute for reading this document!* For example, rules for calling specific routines are not described, nor are the more minor suggestions and recommendations. This section is intended only as a reminder of the major topics. Rationale for the rules is not described here; see the referenced sections for more details.

- Portability

  All VICAR software must run on all platforms that MIPL supports. [Section 1.2]

- Separate Fortran and C Interfaces

  All subroutines must have separate Fortran and C calling sequences, with different names. [Sections 2.4, 9, 9.2]

  Subroutines must call only the appropriate language interface. [Sections 3.3, 3.3.2, 7.1, 9]

- No Optional Arguments

  Routines accepting pure optional arguments are not allowed. [Section 2.1]

  Keyword-value style arguments are permissable in C only. [Sections 2.1, 3.2]

- Data Formats and I/O

  Applications shall be able to read files written in any host representation. [Section 5.1]

  Applications shall normally write files in the native host representations of the machine on which they are currently running. [Section 5.1]

  Programs reading binary labels must be able to read any host format, and convert it to native format before using. [Sections 5.6.1, 5.6.2]

- Data Format Translation

  Do not write your own pixel data type or host representation conversion routines. The routine **x/zvtrans** is the only standard way to translate data among different host representations and pixel data types in VICAR. [Sections 4.2, 5.1]

  Do not use EQUIVALENCE for data type conversion in Fortran. The `INT2BYTE` or `BYTE2INT` routines may be used instead. [Section 7.3]

- Pixel Sizes

  Do not assume the size in bytes of a pixel or other data; it may be different on different machines. Use one of the **x/zvpixsize** routines, or `sizeof` from C, to determine the size of a data element. [Section 5.4]

- Name Registries

  Every property name or BLTYPE name used must be entered into the appropriate name registry. [Sections 4.6.2, 5.6.3]

- Operating System Calls

  All OS-specific code must be eliminated or isolated, and should be written in C if possible. [Sections 6.3, 7.8, 10.4]

- Conditional Compilation

  Conditional compilation statements to handle machine dependencies must never use machine names or types. They shall instead use names of specific features that are defined in standard include files based on the machine type. [Section 6.4]

  Feature dependencies must be defined only in xvmaininc.h or vmachdep.h. [Section 6.4]

  The symbols VMS_OS and UNIX_OS may be used for differences between VMS and Unix operating systems only. [Section 6.4]

- No Terminal I/O

  VICAR applications may not write directly to the terminal (e.g. stdout). Use **x/zvmessage** instead. [Section 7.5]

- ANSI C

  Function prototype include files must use #ifndef wrappers, _NO_PROTO, and extern "C" declarations. [Section 6.5]

  Fortran bridge routines must not use prototypes or the prototype form of declaration. [Sections 6.5, 9.1]

- Fortran 77 Standard

  All Fortran code must conform to the ANSI Fortran-77 standard, with the exception of the allowed extensions listed in the reference. [Section 7.7]

- Fortran Strings

  All Fortran-callable subroutines written in C must use the RTL string conversion routines (**sfor2c** *et al*) to handle CHARACTER*n arguments. [Sections 4.2.4, 9.4]

- Separate SUBLIBs

  Applications cannot use both the unportable and the portable SUBLIB libraries at the same time. [Section 10.1]

- `vimake`

  Standard VICAR application software must use imakefiles compatible with `vimake` to create their build files. [Section 11.1]

- `vpack`

  Standard VICAR application software must be packed into a COM file using the `vpack` command. [Section 11.2]

15 Summary of Calling Sequences

This section contains a brief synopsis of the calling sequences for all the RTL routines, in alphabetical order. No description is given for the routines or their arguments; see the *VICAR Run-Time Library Reference Manual* or Section 4.2, New Routines, for those.

This summary is mainly intended as a guide to help you in the porting process. Use it to find out what the required parameters are for routines that formerly had optional arguments, and how to convert calls from the old Fortran interface to the new C interface.

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. See the *VICAR Run-Time Library Reference Manual* as updated in this document for details on which optionals are active.

The actual data type declarations for the data types below are listed in Table 3 for Fortran, and Table 5 for C. 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

---

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_N4	input	integer
U_ORG	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

xvfilename/zvfilename

```
call xvfilename(in_name, out_name, status)
status = zvfilename(in_name, out_name, out_len);
```

in_name	input	string
out_name	output	string
out_len	input	string
status	output	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
N4	output	integer
ORG	output	string
PIX_SIZE	output	integer
REALFMT	output	string
RECSIZE	output	integer
TYPE	output	string
VARSIZE	output	integer

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

```
keyword = xviptst(name)
keyword = zviptst(name);
```

name	input	string
keyword	output	integer

xvmessage/zvmessage

```
call xvmessage\index{xvmessage}(message, key)
zvmessage(message, key);
```

message	input	string
key	input	string

xvopen/zvopen

```
call xvopen\index{xvopen}(unit, status, <optionals>, ' ')
status = zvopen\index{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_N4	input	integer
U_ORG	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

```
keyword = xvptst(name)
keyword = zvptst(name);
```

name	input	string
keyword	output	integer

xvread/zvread

```
call xvread(unit, buffer, status, <optionals>, ' ')
status = zvread{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\index{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

16 Index

Index