Multimission Image Processing Laboratory

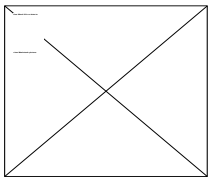
Building and Delivering VICAR Applications

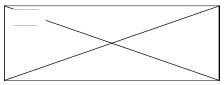
R. Deen
L. Bolef

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

JPL D-16065

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





1. Introduction

This manual describes how to build (compile and link) VICAR programs, to package the parts of the application, and to write the test scripts necessary for class 2 (R2LIB) delivery to the VICAR system.

Before changing or editing this manual, please see 4 Appendix A: About This Document (PAGE 31).

2. vimake

Most UNIX programs come with a "makefile" that is run with the UNIX command `make` to compile and link the program. The makefile describes everything needed to build the program, including compiler options, linker libraries, and location of manual (man) pages. `make` has many advantages, including compiling only what has changed since the last compile.

The preferred method for the VMS operating system is to have compile and link statements directly in a `".COM"` file that contains all the application parts. These are extracted and compiled in one step. 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.

To build applications under UNIX, it is highly desirable to use `make` with a makefile. To build under VMS, it is desirable to use DCL with a command procedure. The makefile and the command procedure are 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. A program called imake has been written to solve these problems.

`imake` is used extensively in the X-windows system to build it on many platforms. TAE uses it as well. `imake` generates makefiles. It uses the C preprocessor for macros and conditional statements to customize the generated 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 are similar to makefiles. There is a set of rules and definitions that are set up at the top of the makefile, but 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 has only 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:

- The programmer need only create one imakefile, instead of separate build files for each system.
- It is much easier to create the imakefile than it would be to create a full-blown build file.
- Build files are standardized, since they are created entirely by the vimake templates, and never modified by the application programmer.
- Locations of libraries and other files are standardized, since the imakefile doesn't specify where the library is, just what library it wants.
- Changing the way applications are built is easy. only the templates need to change, and all applications will be built the new way.

2.1 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 by `/* and */`.

Following is an example 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 t ells 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, because 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 statements. This is typically used to compile a VMS-specific module only under VMS, and its UNIX-specific counterpart only under UNIX.

To use an imakefile, simply type the command vimake followed by the name of the imakefile. imakefiles must have a ".imake" extension, but you should not 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 2.3 Using the Generated VMS Build File (PAGE 21).

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:

There are many different build targets you can give to control the build process, which are described in Section 2.4 Using the Generated UNIX makefile (PAGE 23).

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. In UNIX style, FORTRAN modules must end in ".f", *not* ".for". 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 are space-separated (not tab-separated) lists of names. 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 2.2.2 Name List Macros (PAGE 9), for a description of which includes go in FTNINC_LIST. Local includes, which are part of the application, go in INCLUDE_LIST instead. Example:

```
#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
```

VMS macro (.mar) and array processor files (.vfc) are allowed by vimake, but are VMS specific. When MODULE_LIST is defined differently for different machines, make sure the unused source code is listed in CLEAN_OTHER_LIST 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.** Define a USES macro for each language used:
    - USES_C for modules are written in Kernighan and Ritchie (K&R) C (which is the case for most VICAR code).
    - USES_ANSI_C for modules written in ANSI C (USES_C and USES_ANSI_C are mutually exclusive).
    - USES_ FORTRAN for modules written in FORTAN.
    - USES_MACRO (VMS macro:.mar) and USES_VFC (array processor files: .vfc) are VMS-specific.

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

The vimake system also has the capability to support various kinds of scripts. This is an extension of the PROCEDURE type, which previously has been used only for PDF procedures and has not been very useful.

In order to deliver a script, define the imake file as a PROCEDURE type, then list the scripts in MODULE_LIST. Define the appropriate USES_* macro(s) (see below) and deliver. All other standard vimake rules apply (for instance, you should define the appropriate library, e.g. R2LIB). The scripts can be packed either as pdf's (-p) or as source (-s), but source is probably better.

For backwards compatibility, a PROCEDURE with no module list is assumed to have xxx.pdf as its only module (where xxx is the procedure name). This covers most of the existing procedures, but modules with more than on pdf will eventually need to be modified to list them all. They will work as-is for now, but when we move to an install-based system for builds in the future, where we build in one directory and install the results into another, they won't, so update them as you can.

USES_ name	**Extension**	**Disposition**
USES_PDF	.pdf	left alone for now
USES_SH	.sh	renamed without extension, chmod +x
USES_CSH	.csh	renamed without extension, chmod +x
USES_KSH	.ksh	renamed without extension, chmod +x
USES_BASH	.bash	renamed without extension, chmod +x
USES_PERL	.perl or .pl	renamed without extension, chmod +x, left alone on VMS

USES_DCL	.dcl	ignored on Unix, renamed to .com on VMS
USES_TCL_TK	.tcl	auto_mkindex run on Unix, left alone on VMS

Table 1: Supported script types.

The four Unix shells (sh, csh, ksh, bash) are ignored on VMS, i.e. you could have both .sh and .dcl files in the same MODULE_LIST without having to do #ifdef's based on the operating system.

Note also that while every file requires an extension, many of them are stripped off on Unix, leaving just the base name. So, xxx.csh becomes just xxx and yyy.perl becomes just yyy. And DCL .COM files are named .dcl instead of .com to help avoid confusion with packed .com files (the build will copy them to a .com extension for runtime use).

Below is an example. You would not normally have this many types of things in the same .com file.

```
#define PROCEDURE test
#define MODULE_LIST u.dcl v.pdf w.csh x.sh y.sh z.pl \
Dpos.tcl form1.tcl form12.tcl

#define R2LIB
#define USES_DCL
#define USES_PDF
#define USES_CSH
#define USES_SH
#define USES_PERL
#define USES_TCL_TK
```

- **Class of program unit.** If you are working with a type PROGRAM, then 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 VMS-specific subroutines, 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-specific 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 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), 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 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 always included, you need not add 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, for consistency.

For a complete list of all LIB macros currently defined, see the next section.

You may need a library that is not available via vimake. If so, contact the VICAR system programmer, and the library will be added. 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. The VICAR system programmer must track libraries are in use, so they are not omitted when VICAR is ported to a new system or delivered to an external site.

Some libraries have _DEBUG forms, which link the program with the debuggable version of that library. These are 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
```

Following are more examples of VICAR imakefiles. A simple C program, `gen`, was given above. Below 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
```

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 using portable C code for other machines.

```
#define SUBROUTINE amosids
#if VMS_OS
#define MODULE_LIST amosids. fcamosids. 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 a more complex program, that has many modules, mixed languages, and uses several subroutine libraries.

```
/* 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
```

2.2 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 entries in the order presented here for consistency.

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

2.2.1 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*- This imakefile produces an executable program. "*x"* specifies the name of the program. By convention, the name of the COM file minus ".com".
- **SUBROUTINE** *x*- This imakefile produces a subroutine or subroutine package. The value *x* specifies the name of the subroutine, or of the collection of subroutines. By convention, the name of the COM file minus ".com".
- **PROCEDURE** *x* - Specifies that this imakefile is for a TCL procedure. "*x"* specifies the name of the procedure. By convention, the name of the COM file minus ".com".

Only one of PROGRAM, SUBROUTINE, and PROCEDURE may be defined.

2.2.2 Name List Macros

These macros list the files used in the imakefile. MODULE_LIST must always be given for anything other than PROCEDURE. Others are optional.

- **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 path names. 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 overridden with other lists, normally only MODULE_LIST will be used. MODULE_LIST must be defined. MODULE_LIST can only handle about 150 to 200 characters (2. 5 lines) worth of filenames. The limit is 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. 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. 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 makefile dependencies.

- **INCLUDE_LIST2** *l* - The first extension for INCLUDE_LIST. See INCLUDE_LIST.
- **INCLUDE_LIST3** *l* - The second extension for INCLUDE_LIST if INCLUDE_LIST2 fills up. See INCLUDE_LIST.
- **INCLUDE_LIST4** *l* - The third extension for INCLUDE_LIST if INCLUDE_LIST3 fills up. See INCLUDE_LIST.
- **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* 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_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 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. 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. Since 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 .imakefiles 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 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. 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 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. 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.
- **SCRIPT_LIST** - List of script files (and extensions). Similar to LINK_LIST, in that it is derived from MODULE_LIST normally and SCRIPT_LIST is an override.

SCRIPT_LIST is not implemented on VMS though, so it should not be set by the user (i.e. use MODULE_LIST). Scripts are "built" depending on the type of script, often nothing but a chmod +x is done to them, but it depends on the script type. They're all listed in various USES_x below. Scripts should only be defined for type PROCEDURE (as opposed to PROGRAM or SUBROUTINE). Note that if a PROCEDURE has no MODULE_LIST, it automatically gets "PROCEDURE.pdf" where PROCEDURE is the name defined in the PROCEDURE #define. This is for backwards compatibility. So for example, pdf procedure imake's should now #define MODULE_LIST x.pdf and also #define USES_PDF.

- **SCRIPT_LIST2** - *l* - The first extension for SCRIPT_LIST. See SCRIPT_LIST.
- **SCRIPT_LIST3** - *l* - The second extension for SCRIPT_LIST if SCRIPT_LIST2 fills up. See SCRIPT_LIST.
- **SCRIPT_LIST4** - *l* - The third extension for SCRIPT_LIST if SCRIPT_LIST3 fills up. See

SCRIPT_LIST.

- **SYMBOL_LIST** - Optional list of symbols which should be externally visible in the shared library. The SYMBOL_LIST elements should be defined using the SYMBOL and FSYMBOL macros:

```
#define SYMBOL_LIST SYMBOL(zvzinit) SYMBOL(zzinit) \

SYMBOL(zvpinit) SYMBOL(zv_rtl_init) SYMBOL(sc2for)
SYMBOL(sc2for_array)\

SYMBOL(sfor2c) SYMBOL(sfor2c_array) SYMBOL(sfor2len) SYMBOL(sfor2ptr) \

FSYMBOL(abend) SYMBOL(zabend) FSYMBOL(qprint) SYMBOL(zqprint) \

FSYMBOL(xladd) SYMBOL(zladd) FSYMBOL(xldel) SYMBOL(zldel) \

...
```

Where SYMBOL defines a C symbol, and FSYMBOL defines a Fortran-callable symbol (underscores are added where necessary). See for example librtl.com or librtlf.com in rtl, or for a smaller one, script_glue.com in gui/prog.

2.2.3 Main Language Macros

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

- **MAIN_LANG_C** - If defined, the main program is written in C. 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, MAIN_LANG_C_PLUS_PLUS and MAIN_LANG_ FORTRAN must be defined. Use MAIN_LANG_C for C++, K&R C and ANSI C.
- **MAIN_LANG_C_PLUS_PLUS** - Main language is C++, similar to MAIN_LANG_C.
- **MAIN_ LANG_FORTRAN** - If defined, the main program is written in FORTRAN. 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.

2.2.4 Languages Used Macros

These macros define which languages are used by source files in this imakefile. All the "USES_*" macros indicate that the named language is used in MODULE_LIST; the extension indicates which language it is (e.g. .cc is C++, .c is C, .csh is C-shell, etc.).

- **USES_ANSI_C** - If defined, 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.

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 included via the LIB macros or the class macros (R2LIB, P2_SUBLIB, etc.). main programs written in ANSI C still use the MAIN_LANG_C macro.

- **USES_BASH** - Bourne-again shell scripts. Renames from x.bash to x and chmod +x's. Unix only.

- **USES_C** - 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 be the flag used most often. USES_C may or may not accept some ANS I code, depending on the machine, but if there is any conflict then the K&R interpretation will be used.

USES_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 included via the LIB macros or the class macros(R2LIB, P2_SUBLIB, etc.).

- **USES_CSH** - C-shell scripts. Renames from x.csh to x and chmod +x's. Unix only.
- **USES_C_PLUS_PLUS** - At least one source module is written in C++. All C++ source modules must have a filename extension of ".cc".
- **USES_DCL** - Renames the file from x.dcl to x.com on VMS only. Ignored on Unix. Note that DCL scripts must be delivered with ".dcl" extension to distinguish them from vpack's .com files, but the build renames them so they are runnable.
- **USES_EXTRACT** - Used in librtlf only. It indicates that MODULE_LIST contains EXTRACT or EXTRACT_TAE macros. These pull individual *.o's out of a library (as opposed to the whole library), and are useful for generating shared libraries, where you want to include a specific *.o but not the whole library. The syntax is EXTRACT(module, libname) where module is the name of the module to extract (with the .o), and libname is the name of the library to extract from (without the "lib" part). For example:

```
#define MODULE_LIST EXTRACT(xvzinit.o,rtl) \
EXTRACT_TAE(xqgenbr.o,tae) EXTRACT_TAE(xqtaskbr.o,tae)
```

The only difference between EXTRACT and EXTRACT_TAE is that EXTRACT looks for its library in $V2OLB (v2$olb), while EXTRACT_TAE looks in $TAELIB ($taelib).

- **USES_ FORTRAN** - If defined, 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_KSH** - Korn-shell scripts. Renames from x.ksh to x and chmod +x's. Unix only.
- **USES_MACRO** - If defined, 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.

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_PDF** - PDF scripts, filename extensin of ".pdf". Does nothing with them (at least at the moment, future enhancement might automatically run the pdf->html help converter, or something like that).
- **USES_PERL** - Perl scripts. On Unix, renames x.perl or x.pl to x and chmod +x's. Does nothing on VMS, i.e. VMS works with Perl scripts but the build doesn't do anything to them (no renames).
- **USES_SH** - Bourne-shell scripts. Renames from x.sh to x and chmod +x's. Unix only.
- **USES_TCL_TK** - Tcl/Tk scripts, with extension ".tcl". Runs "auto_mkindex" in the directory, which generates the index file that tcl needs to find the scripts. This should work on VMS too, but this compile step is not yet implemented.

- **USES_VFC** - If defined, 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 versions. All VFC source modules must have a filename extension of ".vfc". Any number of USES flags may be defined, if needed.

2.2.5 Build Flag Macros

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

- **FTN_STRING** - 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 operating systems (notably 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. FTN_STRING must be defined if a C routine merely accepts a FORTRAN string, even if a subroutine ultimately calls the conversion routine. 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 not be delivered with C_ OPTIONS defined. If it is, the C_OPTIONS must be defined in a machine-dependent conditional.

- **CCC_TEMPLATES** - Flag indicating that the module uses C++ templates. For VMS only, you need a #pragma in the source file to indicate template instantiation, which must be in one and only one module that uses a particular template instantiation (you should use a symbol such as TEMPLATE_PRAGMA_NEEDED instead of checking for VMS specifically):

```
#pragma define_template SL_List<SptParamBase *>
```

The template's .h file should include the .cc file for VMS and AXP_UNIX. Appropriate symbols have not yet been defined system wide; contact the VICAR System Programmer for assistance if you need to use this.

- **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 not 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. The argument *x* maybe 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 not be delivered with LINK_ OPTIONS

defined. If it is, the LINK_ OPTIONS must be defined in a machine-dependent conditional.

- **DEBUG** - 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 (though you can use it on any system) that do not have conditional macros in their versions of `make`. For these machines, set DEBUG in the imakefile, and rerun vimake, to build a program for the debugger.

For UNIX machines that do have conditional macros, simply use the "debug" target in the standard generated makefile; the DEBUG flag is not needed. For VMS, you can use the "DEBUG" secondary option in the standard build file, so the DEBUG flag is not needed. A program or subroutine should never be delivered with DEBUG defined.

- **PROFILE** - 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 (though you can use it on any system) that do not have conditional macros in their versions of `make`. For these machines, 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 the PROFILE flag is not needed. A program or subroutine should never be delivered with PROFILE defined.

- **SHARED_LIBRARY** - Instructs vimake to build a shared library, instead of a normal program. Should be used with type PROGRAM.

2.2.6 Program Class Macros

These macros define what class the program is in. Valid for type PROGRAM only. They specify how to link the library for PROGRAM. One (and only one) of these must be defined.

- **MPFLIB** - Mars Pathfinder application. Like R2LIB for MPF programs.
- **R1LIB** - Class 1 VICAR application, makes the class 1 SUBLIB includes available to the C compiler.
- **GUILIB** - GUI application.
- **R2LIB** - Class 2 VICAR application, makes the class 1 and 2 SUBLIB includes available to the C compiler.
- **R3LIB** - Class 3 VICAR application, makes the class 1, 2 and 3 SUBLIB includes available to the C compiler.
- **HWLIB** - HW application, specific to the Mars '96 project, makes the HW includes available to the C compiler.
- **TEST** - Test application.

2.2.7 Subroutine Class Macros

These macros define what class the subroutine is in. Valid for type SUBROUTINE only. They specify how to link the library for SUBROUTINE. One (and only one) of these must be defined.

- **MPF_SUBLIB** - Mars Pathfinder application. Like P2_SUBLIB for MPF subroutines.
- **P1_SUBLIB** - Class 1 portable VICAR subroutine, makes the class 1 SUBLIB includes available to the compiler.
- **P2_SUBLIB** - Class 2 portable VICAR subroutine, makes the class 1 and 2 SUBLIB includes available to the compiler.
- **P3_SUBLIB** - Class 3 portable VICAR subroutine, makes the class 1, 2 and 3 SUBLIB includes

available to the compiler.

- **GUI_SUBLIB** - Like P2_SUBLIB but for gui/gui sub-library. (the GUI has four sub-libraries: gui, base, client, and server).
- **HW_SUBLIB** - HW subroutines, specific to the Mars '96 project, make the HW includes available to the compiler.
- **OLD_SUBLIB** - Class 2 *unportable* VMS-specific VICAR subroutine. The subroutine may not be used with any portable applications.
- **OLD_SUBLIB3** - Obsolete. Class 3 *unportable* VMS-specific VICAR subroutine. The subroutine may not be used with any portable applications.
- **SAGE_BASE_SUBLIB** - Like P2_SUBLIB but for SAGE/SAGE sub-library. (SAGE has three sub-libraries: gui, base, client, and server).
- **SAGE_CLIENT_SUBLIB** - Like P2_SUBLIB but for SAGE/SAGE sub-library. (SAGE has three sub-libraries: gui, base, client, and server).
- **SAGE_SERVER_SUBLIB** - Like P2_SUBLIB but for SAGE/SAGE sub-library. (SAGE has three sub-libraries: gui, base, client, and server).
- **TLM_COMMON_SUBLIB** - Uses the telemetry /sub directory.
- **TLM_GLL_SUBLIB** - Uses the telemetry GLL specific directory.
- **TLM_RPC_SUBLIB** - RPC subroutine for the telemetry subsystem. Uses the idl compiler.

2.2.8 Documentation Macros

These macros describe documentation that must be built. 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 called a “doc” file by the application packer program.

The file must be named according t o 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`.

2.2.9 Library Macros

These macros define the libraries needed for the compile and link steps. LIB_*'s specify how to link the library for PROGRAM, and where to find include files to both PROGRAM and SUBROUTINE. Almost all PROGRAMs and some SUBROUTINEs will define several. Define as many as are needed. They are not useful for type PROCEDURE. LOCAL_LIBRARY is special so it is listed first; the others are in alphabetical order.

See http://rushmore.JPL.NASA.GOV/install/externalsoftware_old.html for a list of external software packages (below) with infomation about each.

- **LOCAL_INCLUDE** - Allow -I to be manually specified in imake file. Should be used only for testing, NEVER in a delivery. Similar to LOCAL_LIBRARY (below).
- **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, LOCAL_LIBRARY 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, LOCAL_LIBRARY is the name of the object library included in the link statement when the LIB_LOCAL flag is set. It is used only during development and debugging. A program or subroutine is never 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_ACE_WRAPPERS** - ACE external library,
- **LIB_CANDELA** - Candela external library. Proprietary, for Sun4 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.
- **LIB_C3JPEG** - Obsolete. Library for the C Cubed JPEG decompression board. LIB_C3JPEG also makes the C3JPEG library includes available to the C compiler.

LIB_C3JPEG 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_CPLT** - Obsolete. Common Plotting Package library. The Common Plotting Package is currently available under VMS only.
- **LIB_CURSES** - CURSES package, a set of subroutines for handling navigation on a terminal screen using the cursor.
- **LIB_DALI** - GEM external library.
- **LIB_DD_PLUS_PLUS** - DD++ external library.
- **LIB_DTR** - Obsolete. VMS Datatrieve database shareable image. The DTR library is available under VMS only.
- **LIB_ FORTRAN** - 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 when linking a C main program to a subroutine written in FORTRAN that uses 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 may need this flag.

The FORTRAN library is automatically included under VMS (flag is ignored), so you will not know if you need it unless you link the program on a UNIX system

- **LIB_FPS** - Obsolete. FPS routines for the VMS array processor. The FPS library is currently available under VMS only.
- **LIB_HWSUB** - portable HW (Mars '94/’96 specific) subroutine library. This flag also makes HW includes available to the C compiler.
- **LIB_HWSUB_DEBUG** - debuggable version of the portable HW (Mars '94/’96 specific) subroutine library. A program should not be delivered with HWSUB_DEBUG. 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_KERBEROS** - Kerberos Version 4 libraries. This flag should be defined for programs that define the LIB_MDMS, and use the MDMS DBMS Query Interface libary (QI).
- **LIB_KERBEROS5** - Kerberos Version 5 libraries. This flag should be defined for programs that use FEI version 3.
- **LIB_LOCAL** - 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** - MATH77 mathematics subroutine library.
- **LIB_MATRACOMP** - Obsolete. Matra compression library. This library is proprietary code supplied by Matra for use with the Mars '96 project.

LIB_MATRACOMP is currently 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 linkgroups for subroutines are implemented), so if you use C3JPEG_AVAIL_OS for the Matra library, put in a comment marking this usage as temporary.

- **LIB_MDMS** - MDMS (Multimission Data Management Subsystem) client library. This flag also makes the MDMS includes available to the C compiler.
- **LIB_MDMS_FEI** - MDMS FEI (File Exchange Interface) library. 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_MDMS_FEI_3** - File Exchange Interface (FEI) client library. Currently, you must also define LIB_KERBEROS5, and LIB_PTHREAD to use the FEI_3 library.
- **LIB_MOTIFAPP** - The MotifApp library.
- **LIB_MOTIF** - X-windows and Motif libraries, specifically X11, Xt, and Xm. This flag also makes the X and Motif includes available to the C compiler.
- **LIB_MPFSUB** - Mars Pathfinder library.
- **LIB_MPFSUB_DEBUG** - Mars Pathfinder debug library. Not implemented under UNIX.
- **LIB_NETWORK** - network support libraries, including RPC's (Remote Procedure Calls) and sockets. This flag also makes the network includes available to the C compiler.

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** - 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** - NIMS calibration library. The NIMS calibration library is currently available under VMS only.
- **LIB_PDS** - Current PDS (Planetary Data System) label library, including lablib3 and OAL.
- **LIB_PDS_LABEL** - Deprecated. Old PDS (Planetary Data System) label library.
- **LIB_PVM** - PVM (Parallel Virtual Machine) library. This flag also makes the PVM includes available to the C compiler.
- **LIB_P1SUB** - portable class 1 SUBLIB library. This flag also makes the class 1 SUBLIB includes available to the C compiler.
- **LIB_P1SUB_DEBUG** - debuggable version of the portable class 1 SUBLIB library. 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_P2SUB** - portable SUBLIB library. If you link toP2SUB, you may *not* link to S2 or S3. This flag also makes the class 2 SUBLIB includes available to the C compiler.
- **LIB_P2SUB_DEBUG** - debuggable version of the portable SUBLIB library. 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_P3SUB** - portable class 3 SUBLIB library. If you link to P3SUB, you may *not* link to S2 or S3. This flag also makes the class 3SUBLIB includes available to the C compiler.
- **LIB_P3SUB_DEBUG** - debuggable version of the class 3 portable SUBLIB library. 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_PTHREAD** - POSIX thread library. This library is available beginning with Solaris 2.5, IRIX 6.2 with a special patch, and HPUX 11.x.
- **LIB_RDM** - Obsolete. RDM library. RDM is currently available under VMS only, where it links as a shareable image.
- **LIB_ROGUEWAVE** - Tools.h++ version 6.1 class library from Roguewave Inc. (http://www.roguewave.com/products/tools/tools.html). Contains over 140 classes, including dates, times, strings, sets, bags, B-Trees, sorted collections, linked lists, queues, stacks, collection, internationalization, streaming and an interface to the Standard C++ Library.
- **LIB_RTL** - VICAR Run-Time Library. For both VMS and most Unix platforms. If a shared library isn't available on a Unix platform, the non-shared version will be used.
- **LIB_RTL_DEBUG** - debuggable version of the VICAR Run-Time Library. A program should never be delivered with RTL_DEBUG. It is used for program development and maintenance only.
- **LIB_RTL_NOSHR** - VICAR Run-Time Library as a standard link library, as opposed to a shared library or a VMS shareable image. Statically links (to .a/.olb) rather than dynamically linking to the shared library (.so/.exe). RTL is preferred over RTL_NOSHR.
- **LIB_SAGE_BASE** - gui/base sub-library
- **LIB_SAGE_BASE_DEBUG** - gui/base debug sub-library
- **LIB_SAGE_CLIENT** - gui/client sub-library
- **LIB_SAGE_CLIENT_DEBUG** - gui/client debug sub-library
- **LIB_SAGE_SERVER** - gui/server sub-library
- **LIB_SAGE_SERVER_DEBUG** - gui/server debug sub-library
- **LIB_SIMBAD -** SIMBAD external library.
- **LIB_SYBASE_NOSHR** - Non-shareable form of SYBASE lib.
- **LIB_SPICE** - SPICE subroutine library.
- **LIB_SYBASE** - Sybase database client library. 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** - Obsolete. Old, unportable, VMS-specific SUBLIB library. S2 is available under VMS only. If you link to S2, you may *not* link to P2SUB or P3SUB.

- **LIB_S2_DEBUG** - Obsolete. debuggable version of the old, unportable, VMS-specific SUBLIB library. Valid only for type PROGRAM. S2_DEBUG is available under VMS only. If you link toS2_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** - Obsolete. Old, unportable, VMS-specific class 3 SUBLIB library. Valid for type PROGRAM. S3 is available under VMS only. If you link to S3, you may *not* link toP2SUB or P3SUB.
- **LIB_S3_DEBUG** - Obsolete. Debuggable version of the old, unportable, VMS-specific class 3 SUBLIB library. S3_DEBUG is available under VMS only. If you link to S3_DEBUG, you may *not* link to P2SUB or P3SUB. A program should not 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** - TAE object library. Under VMS it links TAE as a shareable image. Some versions of UNIX support shared libraries, so TAE links to the TAE shared library.
- **LIB_TAE_NOSHR** - TAE object library as a standard link library, as opposed to a shared library or a VMS shareable image. TAE is preferred over TAE_NOSHR.
- **LIB_TCL_TK** - Tcl/Tk library. (not to be confused with TAE's TCL!)
- **LIB_THREAD** - Solaris thread library. Valid on Solaris 2.x only.
- **LIB_TIFF** - TIFF external library. Also links to geotiff and zlib.
- **LIB_TLM_COMMON** - Telemetry common library.
- **LIB_TLM_GLL** - Telemetry GLL specific library.
- **LIB_TLM_GLL_SUBLIB** - Telemetry GLL specific subroutine library.
- **LIB_TLM_RPC_CLIENT** - Telemetry RPC client library.
- **LIB_TLM_RPC_SERVER** - Telemetry RPC server library.
- **LIB_VRDI** - Virtual Raster Display Interface (VRDI) library. 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** - debuggable version of the VRDI library. A program should never be delivered with VRDI_DEBUG; 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** - VRDI as a standard link library, as opposed to a shared library or a VMS shareable image. 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.
- **LIB_XEXT** - Xext library, part of X-windows ("extensions")
- **LIB_XMU** - Xmu library, part of X-windows ("miscellaneous utilities")
- **LIB_XPM** - X Pixmap library
- **LIB_XRT_3D** - 3D Graph. Proprietary widget set from the KL Group.
- **LIB_XRT_GRAPH** - XRT graph. Proprietary widget set from the KL Group.
- **LIB_XRT_TABLE** - Table (spreadsheet-like). Proprietary widget set from the KL Group.

2.3 Using the Generated VMS Build File

This section describes the options you can use to control the build process using the generated VMS build file

The VMS 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 VMS build file is executed with the "`@` " 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 lowercase on the command line), and the lower case letters are optional. Most options can be abbreviated.

The primary options are:

- **COMPile**: Compiles some or all 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.
- **INST**all: 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 if 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 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.
- **SYS**tem: 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).

Some primaries, such as STD, ALL, and SYStem, are combinations of other primaries. The secondary options apply in these cases as well. 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, give the secondary option NORMal.

It is rarely 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 passed to the compiler (only "`/debug`" is passed to the VMS MACRO assembler).
- **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 passed 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 passed to the linker.
- **MAPALL**: Creates a full link map file. The option "`/full`" is passed 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. Valid 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. But the capability exists 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 how to obtain a full compile listing with cross-reference from a pair of modules:

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

2.4 Using the Generated UNIX makefile

Since most VICAR programmers may not be familiar with `make`, this Section briefly describes how to use it. For more details on using `make`, see the documentation for it.

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 designed to figure out what needs to be built based on what has changed, and to build only changed files. 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 small, this should not cause a problem, but just require 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". The generated makefile is always named " *file*.make" where *file* is the name of the application. Pass the " `-f`" option to `make` to specify the name of the makefile.

If you are developing a program, you may rename the file "makefile" (or create a symbolic link) so `make` will find it automatically. 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 be compiled. It could be the executable name, in which case all the compiles and links necessary to create the executable are performed. There are special targets, such as “all” or “clean.src”, that cause other 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.

There are fewer options on the UNIX 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 debugging on these machines, 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, 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, SUBROUTIN E, and PROCEDURE. The system target performs a clean.src operation, which will delete the source code (since it just came from the COMfile, 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. 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. Valid for PROGRAM and SUBROUTINE.
- **clean.src**: Deletes the source code, imakefile, and makefiles. It is intended mainly for system builds,

where the source code can be deleted after the build because it is maintained in the COMfile. 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 messagefile, 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 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:

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

3. Application Packer

A typical VICAR application program, procedure, or SUBLIB subroutine is composed of several files. These 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 their ".com" extension. 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. Use vunpack to extract files from the COM file.

3.1 vpack

The vpack program accepts a list of parameters:

```
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`" (i makefile) option should be used instead. Machine-specific build files are allowed only under rare 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 module s, "`-u`" must be present; for

portable modules, it must not be.

If you call the vpack program with no parameters, it will report a syntax error and give 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 are altered or deleted. vpack's output is the COM (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 specified.

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
```

An include file (view.h) is considered a "source" file. Multiple file names can be separated by commas or spaces or both. If you arrange the command on several lines, you would need a continuation character appropriate for the operating system you are using.

The vpack program can be executed in two 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
```

The preferred method for executing the vpack program also uses the repack file, but as a parameter to the program:

```
vpack view.repack
```

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 header.
- DCL code which parses command-line options in the VMS environment and sets up the necessary conditions to execute the user's instructions. For example, if the user typed "`@file.com source` ", the source files from the COM file would be created.
- 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: source file(s), PDF file(s), etc. Each section consists of a label header ("$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 utilized by the generated build file: COMPILE, ALL, SYSTEM, CLEAN, etc.). The vpack program is designed 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 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 in formation on the use of vimake and the build options, see 2 vimake (PAGE 3).

```
$!*********************************************************
$!
$! 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 private, unpack the PDF and DOC files.
$!   STD         Build a private, and unpack the PDF file(s).
$!   SYStem      Build the system with the CLEAN option, and
$!               unpack the PDF and DOC files.
$!   CLEAN       Clean (delete/purge) code parts, see 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 in for compatibility).
$!   PDF         Only the PDF file is created.
$!   TEST        Only the test files are created.
$!   IMAKE       Only IMAKE file (w/ VIMAKE program) created.
$!   DOC         Only the documentation files are created.
$!
$!   The default is use the STD parameter if non 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 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, purge executable
(default)
$!   SRC        Delete source and make files
$!
$!*********************************************************
$!
$ write sys$output "*** module view ***"
```

The following options are 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.

Many of these options for build will not typically be used. The recommended 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. The COM file may be edited directly without

unpacking, in which case these options could be useful.

- **COMP**ile: 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:
- **DEB**ug: 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:

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

Secondary options are separated by commas, no spaces.

The command above 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**: 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 additional processing of the documentation files.
- **STD**: Carries out the same actions as the ALL command, with the exception that documentation files are not unpacked and generated. STD is the default option for portable modules (without the “`-u`” flag for vpack).
- **SYS**tem: 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, while 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 passed to vpack.

- **CLEAN**: 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. Make sure the COM file is up to date before deleting the files that make it up.

The remaining COM file options select the files 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 repackfile 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 text file, it is editable by the user. We recommend you follow the normal procedure of unpacking the file(s), making and testing the necessary modifications, then using vpack to rebuild the COM file.

3.2 vunpack

The vunpack program unpacks files from a COM file created by vpack. Its primary use is on UNIX systems, where the COM file is not self-extracting, though it may be used in VMS.

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
```

If you call the vunpack program with no parameters, the program will report an error and will tell you the proper syntax to use.

The first parameter tells vunpack which COM file to extract files from. 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 2.3 Using the Generated VMS Build File (PAGE 21) or 2.4 Using the Generated UNIX makefile (PAGE 23) for details.

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

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

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

If you need all of the source files and the imakefile from VIEW.COM:

```
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.

3.3 Test Routines

Test routines are required for every VICAR class 2 (R2LIB) application and subroutine. Programs and procedures typically have a procedure PDF as the test routine.

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, and have its own imakefile and its own PDF.

Subroutines often have four test files :the test program, the imakefile, the test program PDF, and the test PDF that calls it. These files are under the TEST file category in vpack. To run the test, first unpack the TEST files, run vimake on the test imakefile, build the program, and execute the test PDF. 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.

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. Run `diff` on the new log versus the old log, to make sure nothing changed.

Checking the logs may be difficult because TAE automatically puts `usage` statistics 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 can hide any real differences.

For this reason, the TAE global variable $AUTOUSAGE has been added to control the automatic printing of usage statistics. It takes three values: “BATCH”, “ALL”, or ”NONE”. Use the TAE command “`refgbl $AUTOUSAGE`” before the “body” statement, then set it with a standard “let” command. The values are described below:

- **BATCH**: The default. 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**: Automatic usage statistics are printed after *every* program execution, in both batch and interactive modes.
- **NONE**: Suppresses automatic usage statistics in both batch and interactive modes. With “NONE” set, the only way to get the statistics is via the `usage` command.

We recommend that $AUTOUSAGE be set to ”NONE” in every test script, to avoid unneeded usage statistics.

It is possible that the results of a test involving floating-point calculations may be differ on various 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.

4. Appendix A: About This Document

This manual combines material from a previous manual: “VICAR Porting Guide” JPL D-9395, 1994, http://www-mipl.jpl.nasa.gov/portguide/portguide.html, written by Robert G. Deen, Robert.G.Deen@jpl.nasa.gov.

4.1 Document Source

This document was written in the Microsoft Word program. Any changes or additions to it must be made in the original Word document. This is available in two versions: Word native (binary): http://www-mipl.jpl.nasa.gov/buildapps/Build_VICAR_Apps.wrd and Microsoft RTF (Rich Text Format): http://www-mipl.jpl.nasa.gov//buildapps/Build_VICAR_Apps.rtf. RTF documents are ASCII text with embedded formatting commands, and can be imported by many word processing programs.

An Adobe PDF version of this document at: http://www-mipl.jpl.nasa.gov/buildapps/Build_VICAR_Apps.pdf is available for easy printing or viewing using the Adobe Acrobat Reader: http://www.adobe.com/prodindex/acrobat/readstep.html.

4.2 Generating HTML Version

The HTML version of this manual is generated automatically from the RTF version by the RTFtoHTML filter program, available from: http://www.sunpack.com/RTF/. You must have a modified version of the html-trn parameter file: http://rushmore.JPL.NASA.GOV/buildapps/html-trn in the directory containing the RTFtoHTML program.

Use this command line to generate the HTML version of the manual:

```
./rtftohtml -x -c -h2 -T "Building VICAR Applications"
Build_VICAR_Applications.rtf
```

4.3 Changing or Adding to this Document

Any editing done in this document may result in page numbers or section numbers changing. These changes are made automatically only if you use cross-references. If you don't undertand how to use cross-references, please consult the manual or the online help.

The HTML generation step can only be successful if you use one of a limited number of styles RTFtoHTML understands.The RTFtoHTML manual: http://www.sunpack.com/RTF/guide.htm has complete information.

Since page number references are not useful in HTML, all references of the form: (PAGE 233) must be formatted with the small caps attribute. The html-trn file is modified so that RTFtoHTML will not translate any text in small caps.

4.3.1 Styles used in this Document

Below is a list of styles used in this manual. If you use any other styles, please consult the RTFtoHTML manual: http://www.sunpack.com/RTF/guide.htm and examine the html-trn parameter file to make sure the additional styles will not break the HTML generation process.

- Heading 1 - top level heading
- Heading 2 - second level heading
- Heading 3 - third level heading
- Normal - text in Times font without first-line indent
- Normal Indent - text in Times font with first-line indent, no indent in HTML
- bullet list - first level item list, not numbered
- bullet list 1 - second level item list, not numbered
- pre - preformatted text in Courier font, used for code or computer file names
- glossary - acronym or definition lists
- Caption - table caption

4.3.2 Formatting Hints and Kinks

The html-trn is modified for HTML generation to:

- Discard any text that is has the text format "small caps". This can be found in the Fonts menu item in the Format menu.
- Add extra line breaks (
) after paragraphs and list items.