A User's Guide to Fortran Programming in IRAF
The IMFORT Interface

Doug Tody

National Optical Astronomy Observatories*
September 1986

ABSTRACT

The IMFORT interface is a Fortran programming environment suitable for general Fortran programming, with special emphasis on batch image processing. IMFORT is intended for use primarily by the scientist/user who occasionally needs to write a program for their own personal use, but who does not program often enough to make it worthwhile learning a larger, more complex but fully featured programming environment. IMFORT is therefore a small interface which is easy to learn and use, and which relies heavily upon host system (non-IRAF) facilities which the user is assumed to already be familiar with. Facilities are provided for accessing command line arguments, reading and writing IRAF images, and returning output to the CL. Provisions are made for editing, compiling, linking, and debugging programs without need to leave the IRAF environment, making use of familiar host system editing and debugging tools wherever possible.

*Operated by the Association of Universities for Research in Astronomy, Inc. under cooperative agreement with the National Science Foundation.

Contents

A User's Guide to Fortran Programming in IRAF
The IMFORT Interface

Doug Tody

National Optical Astronomy Observatories*
September 1986

1. Introduction

The IMFORT interface is a library of Fortran callable subroutines which can be called from a host system Fortran program to perform such operations as fetching the arguments given on the command line when the task was invoked, or accessing the header or pixel information in an IRAF image (bulk data frame). Since the result is a host program rather than an IRAF program, only limited access to the facilities provided by the runtime IRAF system is possible, but on the other hand one has full access to the facilities provided by the host system. Programs which use IMFORT may be run as ordinary host system programs outside of IRAF, or may be interfaced to the IRAF command language (CL) as CL callable tasks. Within the IRAF environment these user written, non-IRAF tasks behave much like ordinary IRAF tasks, allowing background execution, use of i/o redirection and pipes, evaluation of expressions on the command line, programmed execution in scripts, and so on.

1.1. Who Should Use IMFORT

The most significant feature of the IMFORT interface is that it is designed for use by *host* Fortran programs. The scientist/user will often already be using such programs when IRAF becomes available. IMFORT allows these pre-existing programs to be modified to work within the IRAF environment with a minimum of effort and with minimum changes to the existing program. The only alternative is to rework these programs as *IRAF* programs, but few existing Fortran programs could (or should) survive such a transition without being completely rewritten. If the program in question is useful enough such a rewrite might be warranted, but in most cases this will not be practical, hence something like the IMFORT interface is clearly needed to keep these old programs alive until they are no longer needed.

The second goal of the IMFORT interface is to provide a way for the user to add their own programs to IRAF without having to invest a lot of time learning the full blown IRAF programming environment. IMFORT makes it possible for the user to begin writing useful programs within hours of their first exposure to the system. It is possible that the IMFORT interface will provide all the capability that some users will ever need, especially when supplemented by other (non-IRAF) Fortran callable libraries available on the local host machine. Programs developed in this way are bound to have portability and other problems, but it should be up to the developer and user of the software to decide whether these problems are worth worrying about. IMFORT is simply a *tool*, to be used as one sees fit; there is no attempt to dictate to the user how they should write their programs.

The alternative to IMFORT, if applications programming within IRAF is the goal, is the IRAF SPP/VOS programming environment. The SPP/VOS programming environment is a fully featured scientific programming environment which carefully addresses all the software engineering issues avoided by IMFORT. The VOS is a large and complex environment and therefore takes longer to learn than IMFORT, but it provides all the facilities needed by large applications hence is *easier* to use than simpler interfaces like IMFORT, if one is faced with the already difficult task of coding a large program or package. Furthermore, the SPP/VOS

*Operated by the Association of Universities for Research in Astronomy, Inc. under cooperative agreement with the National Science Foundation.

environment fully addresses the problems of portability and device independence, critical issues for applications which must be supported and used simultaneously on a range of machines over a period of years, during which time the software is likely to be continually evolving. An overview of the SPP/VOS programming environment is given in *The IRAF Data Reduction and Analysis System*, February 1986, by the author.

In summary, IMFORT is intended for use to interface old Fortran programs to IRAF with a minimum of effort, and as an entry level programming environment which new users can learn to use in a few hours. Experienced users, professional programmers, and developers of large applications will find that they can accomplish more with less effort once they have learned to use the more fully featured SPP/VOS programming environment.

2. Getting Started

Although programs which use IMFORT can and often will be invoked from the host system command interpreter, it is likely that such programs will also be used interactively in combination with the tasks provided by the standard IRAF system. For example, the IRAF graphics and image display facilities are likely to be used to examine the results of an image operation performed by a user written Fortran/IMFORT program. Indeed, the standard IRAF tasks are likely to be used for testing new IMFORT programs as well as reducing data with old ones, so we shall assume that software development will take place from within the IRAF environment. Since IRAF provides full access to the facilities of the host system at all times, there is little reason not to work from within the IRAF environment.

As a first step, let's see what is required to enter, compile, link, and execute a small Fortran program which does nothing more than print the message `hello, world!` on the terminal. We shall assume that the reader has read the *CL User's Guide* and is already familiar with basic CL command entry, the OS escape facility, the editor interface and so on. The first step is to call up the editor to enter the program into a file:

```
cl> edit hello.f
```

Note that the filename extension is ".f", which is what IRAF uses for Fortran files. The extension will be mapped into the local host system equivalent when IRAF communicates with the host system, but when working in the IRAF environment the IRAF name should be used.

Once in the editor, enter the following program text:

```
program hello
write (*,*) 'hello, world!'
stop
end
```

The next step is to compile and link the `hello` program. This is done by the command *fc* (*f*ortran-*c*ompile), which produces an object file `hello.o` and an executable program file `hello.e`. Note that the *fc* task is defined in the default *user* package in your `LOGIN.CL` file, hence a *mkiraf* may be required to regenerate the `LOGIN.CL` file if the file is old or has been modified.

```
cl> fc hello.f
```

Since the `hello` program is a host Fortran program, it can be executed immediately with an OS escape, e.g., `!hello.e` on UNIX, or `!run hello` on VMS. A better approach if the task has command line arguments is to use the IRAF *foreign task* facility to define the program as a new IRAF task, as we shall see in the next section.

2.1. Example 1: Plotting a function

As a slightly more complicated example, let's construct a program to compute and plot a function using command line arguments to input the function parameters, with output consisting of a simple ASCII table sampling the computed function. Our example computes the Planck function, which gives the emissivity of a blackbody as a function of wavelength and temperature. The sample program is shown in Figure 1. Source code for this and all other examples in this paper may be found in the IRAF directory `imfort$tasks`.

```
c PLANCK -- Compute the Planck blackbody radiation distribution for a
c given temperature and wavelength region.
c
c       usage:  planck temperature lambda1 lambda2
c
c The temperature is specified in degrees Kelvin and the wavelength
c region in microns (1u=10000A).  100 [x,y] data points defining the
c curve are output.
c ------------------------------------------------------------------------

        program planck

        character*80    errmsg
        integer         nargs, ier, i
        real            w1, w2, dw, cm, t
        real            xv(100), yv(100)

c --- Get the temperature in degrees kelvin.
        call clargr (1, t, ier)
        if (ier .ne. 0) then
            write (*, '('' temperature (degrees kelvin): '',$)')
            read (*,*) t
        endif

c --- Get the wavelength region to be computed.
        call clnarg (nargs)
        if (nargs .ge. 3) then
            call clargr (2, w1, ier)
            if (ier .ne. 0) goto 91
            call clargr (3, w2, ier)
            if (ier .ne. 0) goto 91
        else
            write (*, '('' start wavelength (microns): '',$)')
            read (*,*) w1
            write (*, '('' end wavelength (microns): '',$)')
            read (*,*) w2
        endif

c --- Compute the blackbody curve.
        dw = (w2 - w1) / 99.0
        do 10 i = 1, 100
            xv(i) = ((i-1) * dw) + w1
            cm = xv(i) * 1.0E-4
            yv(i) = (3.74185E-5 * (cm ** -5)) /
     *          (2.71828 ** (1.43883 / (cm * t)) - 1.0)
 10     continue

c --- Print the curve as a table.
        do 20 i = 1, 100
            write (*, '(f7.4, g12.4)') xv(i), yv(i)
 20     continue

        stop

c --- Error exit.
 91     call imemsg (ier, errmsg)
        write (*, '('' Error: '', a80)') errmsg
        stop
        end
```

Figure 1. Sample program to compute the Planck function‡

‡The trailing `$` carriage control code used in the format strings in the `WRITE` statements in this and the other sample Fortran programs is nonstandard Fortran and may not be available on all host machines. Its function is to defeat the carriage-return linefeed so that the user's response may be entered on the same line as the prompt.

This example serves to demonstrate the use of the IMFORT *clarg* procedures to fetch the command line arguments, and the use of i/o redirection to capture the output to generate the plot. The command line to an IMFORT program consists of a sequence of arguments delimited by spaces or tabs. The subroutine *clnarg* returns the number of arguments present on the command line when the task was called. The *clargr*, *clargi*, etc. procedures fetch and decode the values of the individual arguments. Virtually all IMFORT procedures include an integer output variable *ier* in their argument list; a zero status indicates success, anything else indicates failure and the actual error code identifies the cause of the problem. The *imemsg* procedure may be called to convert IMFORT error codes into error message strings, as in the example.

Once the program has been entered and compiled and linked with *fc*, we must declare the program as a foreign task to the CL. If this is not done the program can still be run via an OS escape, but none of the advanced CL features will be available, e.g., background execution, command line expression evaluation, i/o redirection, and so on. The technique used to declare a foreign task is machine dependent since it depends upon the syntax of the host command interpreter. For example, to declare the new CL foreign task *planck* on a UNIX system, we enter the following command:

```
cl> task $planck = $planck.e
```

The same thing can be achieved on a VMS system with the following declaration (it can be simplified by moving the VMS foreign task declaration to your `LOGIN.COM` file):

```
cl> task $planck = "$planck:==\$disk:[dir...]planck.exe!planck"
```

The `$` characters are required to tell the CL that the new task does not have a parameter file, and is a foreign task rather than a regular IRAF task. The `!` in the VMS example is used to delimit multiple DCL commands; the command shown defines the DCL foreign task *planck* and then executes it. The use of the *task* statement to declare foreign tasks is discussed in detail in §3.3.

We have written the program in such a way that the arguments will be queried for if not given on the command line, so if we enter only the name of the command, an interaction such as the following will occur:

```
cl> planck
temperature (degrees kelvin): 3000
start wavelength (microns): .1
end wavelength (microns): 4
```

Note that if the output of the *planck* task is redirected this input mechanism will *not* work, since the queries will be redirected along with the output. Hence if we use a pipe to capture the output, as in the following example, the arguments must be given on the command line.

```
cl> planck 3000 0.1 4.0 | graph
```

This command will compute and plot the emissivity for a 3000 degree kelvin blackbody from 0.1 to 4.0 microns (1000 to 40000 angstroms).

An interesting alternative way to implement the above program would be to output the function curve as a line in an image, rather than as a table of numbers. For example, a two dimensional image could be generated wherein each line corresponds to a different temperature. *Graph* or *implot* could then be used to plot curves or overplot families of curves; this would be more efficient than the technique employed in our sample program. Image access via IMFORT is illustrated in our next example.

2.2. Example 2: Compute the range of pixel values in an image

The program shown in Figure 2 opens the named image, examines each line in the image to determine the minimum and maximum pixel values, keeping a running tally until the entire image has been examined (there is no provision for detecting and ignoring bad pixels in the image). The newly computed minimum and maximum pixel values are then updated in the image header as well as printed on the standard output.

```
c MINMAX -- Compute the minimum and maximum pixel values in an image.
c The new values are printed as well as updated in the image header.
c
c       usage:  minmax image
c ----------------------------------------------------------------------

        program minmax

        character*80    image, errmsg
        real            pix(4096), dmin, dmax, vmin, vmax
        integer         im, axlen(7), naxis, dtype, ier, j

c --- Get image name.
        call clargc (1, image, ier)
        if (ier .ne. 0) then
            write (*, '(' enter image name: ',$)')
            read (*,*) image
        endif

c --- Open the image for readwrite access (we need to update the header).
        call imopen (image, 3, im, ier)
        if (ier .ne. 0) goto 91
        call imgsiz (im, axlen, naxis, dtype, ier)
        if (ier .ne. 0) goto 91

c --- Read through the image and compute the limiting pixel values.
        do 10 j = 1, axlen(2)
            call imgl2r (im, pix, j, ier)
            if (ier .ne. 0) goto 91
            call alimr (pix, axlen(1), vmin, vmax)
            if (j .eq. 1) then
                dmin = vmin
                dmax = vmax
            else
                dmin = min (dmin, vmin)
                dmax = max (dmax, vmax)
            endif
 10     continue

c --- Update the image header.
        call impkwr (im, 'datamin', dmin, ier)
        if (ier .ne. 0) goto 91
        call impkwr (im, 'datamax', dmax, ier)
        if (ier .ne. 0) goto 91

c --- Clean up.
        call imclos (im, ier)
        if (ier .ne. 0) goto 91
        write (*, '(a20, 2 g12.5)') image, dmin, dmax
        stop

c --- Error exit.
 91     call imemsg (ier, errmsg)
        write (*, '(' Error: ', a80)') errmsg
        stop
        end
```

Figure 2. Compute the min and max pixel values in an image

The program as written can only deal with images of one or two dimensions, of pixel type short (16 bit integer) or real (32 bit floating), with a line length not to exceed 4096 pixels per line. We could easily change the program to deal with images of up to three dimensions, but the IMFORT interface does not provide dynamic memory allocation facilities so there is always going to be an upper limit on the line length if we use the simple get line i/o procedure *imgl2r*, as shown. The use of fixed size buffers simplifies the program, however, and is not expected to be a serious problem in most IMFORT applications.

The *alimr* subroutine in the previous example is from the IRAF VOPS (vector operators) package. The function of *alimr* is to compute the limiting (min and max) pixel values in a vector of type real, the function type being indicated by the *lim*, and the pixel datatype by the *r*. The VOPS package provides many other such vector operators, and is discussed further in § 4.4.

2.3. Example 3: Copy an image

Our final example (Figure 3) shows how to create a new image as a copy of some existing image. This can be used as a template to create any binary image operator, i.e., any program which computes some transformation upon an existing image, writing a new image as output.

By now the functioning of this procedure should be self evident. The only thing here which is at all subtle is the subroutine *imopnc*, used to open (create) a new copy of an existing image. The open new copy operation creates a new image the same size and datatype as the old image, and copies the image header of the old image to the new image. Any user keywords in the header of the old image will be automatically passed to the new image, without requiring that the calling program have explicit knowledge of the contents of the image header.

Note that the program is written to work only on pixels of type real, hence will be inefficient if used to copy images of type short-integer. A more efficient approach for a general image copy operator would be to add a conditional test on the variable `dtype`, executing a different copy-loop for each datatype, to avoid having to convert from integer to real and back again when copying a short-integer image. The short-integer equivalents of *imgl3r* (get line, 3 dim image, type real) and *impl3r* (put line, 3 dim image, type real) are called *imgl3s* and *impl3s*.

The program as written will work for images of up to three dimensions, even though it is written to deal with only the three dimensional case. This works because the length of the "unused" axes in an image is set to one when the image is created. A program passed an image of higher dimension than it is written for will also work, but will not process all of the data. IMFORT does not support image sections, so only the first few lines of the image will be accessible to such a program.

Additional useful examples of Fortran programs using IMFORT are given in `imfort$tasks`. These include utility programs to make test images, print the contents of an image header, print the values of the pixels in a subraster, and so on. You may wish to copy the source for these to your own workspace for use as is, or for use as templates to construct similar programs.

```
c IMCOPY -- Copy an image.  Works for images of up to three dimensions
c with a pixel type of short or real and up to 4096 pixels per line.
c
c       usage:  imcopy oldimage newimage
c ----------------------------------------------------------------------

        program imcopy

        real            pix(4096)
        character*80    oimage, nimage, errmsg
        integer         ncols, nlines, nbands, j, k, oim, nim
        integer         ier, axlen(7), naxis, dtype, nargs

c --- Get command line arguments.
        call clnarg (nargs)
        if (nargs .eq. 2) then
            call clargc (1, oimage, ier)
            if (ier .ne. 0) goto 91
            call clargc (2, nimage, ier)
            if (ier .ne. 0) goto 91
        else
            write (*, '('' input image: '',$)')
            read (*,*) oimage
            write (*, '('' output image: '',$)')
            read (*,*) nimage
        endif

c --- Open the input image and create a new-copy output image.
        call imopen (oimage, 1, oim, ier)
        if (ier .ne. 0) goto 91
        call imopnc (nimage, oim, nim, ier)
        if (ier .ne. 0) goto 91

c --- Determine the size and pixel type of the image being copied.
        call imgsiz (oim, axlen, naxis, dtype, ier)
        if (ier .ne. 0) goto 91

        ncols  = axlen(1)
        nlines = axlen(2)
        nbands = axlen(3)

c --- Copy the image.
       do 15 k = 1, nbands
           do 10 j = 1, nlines
               call imgl3r (oim, pix, j, k, ier)
               if (ier .ne. 0) goto 91
               call impl3r (nim, pix, j, k, ier)
               if (ier .ne. 0) goto 91
 10         continue
 15     continue

c --- Clean up.
        call imclos (oim, ier)
        if (ier .ne. 0) goto 91
        call imclos (nim, ier)
        if (ier .ne. 0) goto 91

        stop

c --- Error actions.
 91     call imemsg (ier, errmsg)
        write (*, '('' Error: '', a80)') errmsg
        stop
        end
```

Figure 3. Image copy program

3. The IMFORT Programming Environment

IRAF provides a small programming environment for the development of host Fortran programs using the IMFORT interface. This environment consists of the general CL tools, e.g., the editor, the *page* and *lprint* tasks, etc., plus a few special tools, namely, the *fc* compile/link utility and the foreign task facility. In this section we discuss these special tools and facilities. Information is also provided for linking to the IMFORT libraries if program development is to take place at the host system level.

The classic third generation program development cycle (ignoring such minor details as designing the software) is edit — compile/link — debug. The edit phase uses the CL *edit* task, an interface to the host system editor of choice. The compile/link phase is performed by the *fc* utility. The debug phase is optional and is generally only necessary for large programs. The host system debug tool is used; while IRAF does not provide a special interface to the host debug tool, one can easily be constructed using the foreign task facility if desired.

Programs which use the IMFORT interface are inevitably host system dependent to some degree, since they are host programs. In the interests of providing the user with concrete examples, the discussion in this section must therefore delve into the specifics of certain host operating systems. We have chosen to use UNIX and VMS in the examples, since most IRAF implementations run on one or the other of these operating systems. The ties between the IMFORT programming environment and the host system are quite simple, however, so it should not be difficult to see how to modify the examples for a different host.

3.1. The FC Compile/Link Utility

The *fc* utility provides a consistent, machine independent interface to the host system compiler and linker which is convenient and easy to use. In addition, *fc* provides a means for linking host programs with the IRAF libraries without having to type a lot, and without having to build host command scripts. All of the IRAF libraries are accessible via *fc*, not just IMFORT (`lib$libimfort.a`) and the IRAF system libraries used by IMFORT, but all the other IRAF libraries as well, e.g., the math libraries.

The default action of *fc* is to compile and link the files listed on the command line, i.e., source files in various languages, object modules, and libraries. Any source files are first turned into object modules, then the objects are linked in the order given, searching any libraries in the order in which they are encountered on the command line (the IMFORT libraries are searched automatically, after any libraries listed on the command line). By default, the root name of the new executable will be the same as that of the first file listed on the command line; a different name may be assigned with the *-o* switch if desired.

The syntax of the *fc* command is as follows:

`fc [`*switches*`] `*file*` [`*file ...*`] [-o `*exefile*`]`

The most interesting switches are as follows:

-c Compile but do not link.

-l*library*
: Link to the named IRAF library. On a UNIX host this switch may also be used to reference the UNIX libraries. The *-llibrary* reference should be given in the file list at the point at which you want the library to be searched. The *-l* causes *fc* to look in a set of standard places for the named library; user libraries should be referenced directly by the filename of the library.

-o *exefile*
: Override the default name for the executable file produced by the linker.

-x Compile and link for debugging.

Since the *fc* command line can contain many different types of objects, a filename extension is required to identify the object type. The IRAF filename extensions *must* be used; these are listed in the table below.

IRAF Filename Extensions	
extn	*usage*
.a	object library
.c	C source file
.e	executable
.f	Fortran source file
.o	object module
.s	Assembler source file
.x	SPP source file

The *fc* utility is easy to learn and use. Here are a few examples illustrating the most common usage of the utility. To compile and link the Fortran program `prog.f`, producing the executable program `prog.e`:

```
cl> fc prog.f
```

To compile the file `util.f` to produce the object `util.o`, without linking anything:

```
cl> fc -c util.f
```

To link `prog.o` and `util.o`, producing the executable program `prog.e`:

```
cl> fc prog.o util.o
```

To do the same thing, producing an executable named `foo.e` instead of `prog.e`:

```
cl> fc prog.o util.o -o foo.e
```

To compile and link `prog.f` for debugging:

```
cl> fc -x prog.f
```

To link `prog.o` with the IRAF library `lib$libdeboor.a` (the DeBoor spline package), producing the executable `prog.e` as output:

```
cl> fc prog.o -ldeboor
```

To do the same thing, spooling the output in the file `spool` and running the whole thing in the background:

```
cl> fc prog.o -ldeboor >& spool &
```

To link instead with the library `libfoo.a`, in the current directory (note that in this case the library is a module and not a switch):

```
cl> fc prog.o libfoo.a
```

Just about any combination of switches and modules that makes sense will work. The order of libraries in the argument list is important, as they will be searched in the order in which they are listed on the command line.

The *fc* utility is actually just a front-end to the standard IRAF compiler *xc*, as we shall see in §3.3. See the manual page for *xc* for additional information.

3.2. Host Level Linking to the IMFORT Libraries

In some cases it may be desirable to use host system facilities to compile and link programs which use the IMFORT interface. The procedure for doing this is host dependent and is completely up to the user, who no doubt will already have a preferred technique worked out. All one needs to know in this situation are the names of the libraries to be linked, and the order in which they are to be linked. The libraries are as follows, using the IRAF filenames for the libraries. All the libraries listed are referenced internally by the IMFORT code hence are required.

lib$libimfort.a	IMFORT itself
lib$libsys.a	Contains certain pure code modules used by IMFORT
lib$libvops.a	The VOPS vector operators library
hlib$libos.a	The IRAF kernel (i/o primitives)

The host pathnames of these libraries will probably be evident, given the host pathname of the IRAF root directory (*lib* is a subdirectory of the IRAF root directory). If in doubt, the *osfn* intrinsic function may be used while in the CL to print the host pathname of the desired library. For example,

```
cl> = osfn ("lib$libimfort.a")
```

will cause the CL to print the host pathname of the main IMFORT library.

3.3. Calling Host Programs from the CL

Since Fortran programs which use IMFORT are host programs rather than IRAF programs, the CL *foreign task* interface is used to connect the programs to the CL as CL callable tasks. The foreign task interface may also be used to provide custom CL task interfaces to other host system utilities, e.g., the debugger or the librarian.

The function of the *task* statement in the CL is to make a new task known to the CL. The CL must know the name of the new task, the name of the package to which it is to be added, whether or not the new task has a parameter file, the type of task being defined, and the name of the file in which the task resides. At present new tasks are always added to the "current" package. The possible types of tasks are normal IRAF executable tasks, CL script tasks, and foreign tasks. Our interest here is only in the forms of the task statement used to declare foreign tasks. There are two such forms at present. The simplest is the following:

```
task $taskname [, $taskname...] = $foreign
```

This form is used when the command to be sent to the host system to run the task is identical to the name by which the task is known to the CL. Note that any number of new tasks may be declared at one time with this form of the task statement. The `$` prefixing each *taskname* tells the CL that the task does not have a parameter file. The `$foreign` tells the CL that the new tasks are foreign tasks and that the host command is the same as *taskname*. For example, most systems have a system utility *mail* which is used to read or send electronic mail. To declare the *mail* task as an IRAF foreign task, we could enter the following declaration, and then just call the *mail* task from within the CL like any other IRAF task.

```
task $mail = $foreign
```

The more general form of the foreign task statement is shown below. The host command string must be quoted if it contains blanks or any other special characters; `$` is a reserved character and must be escaped to be included in the command sent to the host system.

```
task $taskname = $host_command_string
```

In this form of the task statement, the command to be sent to the host system to execute the

new IRAF task may be any string. For example, on a VMS host, we might want to define the *mail* task so that outgoing messages are always composed in the editor. This could be set up by adding the `/EDIT` switch to the command sent to VMS:

```
task $mail = $mail/edit
```

Foreign task statements which reference user-written Fortran programs often refer to the program by its filename. For the task to work regardless of the current directory, either the full pathname of the executable file must be given, or some provision must be made at the host command interpreter level to ensure that the task can be found.

When a foreign task is called from the CL, the CL builds up the command string to be sent to the host command interpreter by converting each command line argument to a string and appending it to *host_command_string* preceded by a space. This is the principal difference between the foreign task interface and the low level OS escape facility: in the case of a foreign task, the command line is fully parsed, permitting general expression evaluation, i/o redirection, background execution, minimum match abbreviations, and so on.

In most cases this simple method of composing the command to be sent to the host system is sufficient. There are occasional cases, however, where it is desirable to *embed* the command line arguments somewhere in the string to be sent to the host system. A special *argument substitution* notation is provided for this purpose. In this form of the task statement, *host_command_string* contains special symbols which are replaced by the CL command line arguments to form the final host command string. These special symbols are defined in the table below.

$0	replaced by *taskname*
$1, $2, ..., $9	replaced by the indicated argument string
$*	replaced by the entire argument list
$(N)	use host equivalent of filename argument N (1-9 or *)

An example of this form of the task statement is the *fc* task discussed in §3.1. As we noted earlier, *fc* is merely a front-end to the more general IRAF HSI command/link utility *xc*. In fact, *fc* is implemented as a foreign task defined in the default *user* package in the `LOGIN.CL` file. The task declaration used to define *fc* is shown below. The task statement shown is for UNIX; the VMS version is identical except that the `-O` switch must be quoted else DCL will convert it to lower case. In general, foreign task statements are necessarily machine dependent, since their function is to send a command to the host system.

```
task $fc = "$xc -h -O $* -limfort -lsys -lvops -los"
```

The argument substitution facility is particularly useful when the host command template consists of several statements to be executed by the host command interpreter in sequence each time the CL foreign task is called. In this case, a delimiter character of some sort is required to delimit the host command interpreter statements. Once again, this is host system dependent, since the delimiter character to be used is defined by the syntax of the host command interpreter. On UNIX systems the command delimiter character is semicolon (‘**;**’). VMS DCL does not allow multiple statements to be given on a single command line, but the IRAF interface to DCL does, using the exclamation character (‘**!**’), which is the comment character in DCL.

The `$()` form of argument substitution is useful for foreign tasks with one or more filename arguments. The indicated argument or arguments are taken to be IRAF virtual filenames, and are mapped into their host filename equivalents to build up the host command string. For example, assume that we have an IMFORT task *phead*, the function of which is to print the header of an image in FITS format on the standard output (there really is such a program - look in `imfort$tasks/phead.f`). We might declare the task as follows

(assuming that *phead* means something to the host system):

```
task $phead = "$phead $(*)"
```

We could then call the new task from within the CL to list the header of, for example, the standard test image `dev$pix`, and page the output:

```
cl> phead dev$pix | page
```

Or we could direct the output to the line printer:

```
cl> phead dev$pix | lpr
```

Filename translation is available for all forms of argument substitution symbols, e.g., `$(1)`, `$(2)`, `$(*)`, and so on; merely add the parenthesis.

It is suggested that new foreign task statements, if not typed in interactively, be added to the *user* package in your `LOGIN.CL` file, so that the definitions are not discarded when you log out of the CL or exit a package. If you want to make the new tasks available to other IRAF users they can be added to the *local* package by adding the task statements to the file `local$tasks/local.cl`. If this becomes unwieldy the next step is to define a new package and add it to the system; this is not difficult to do, but it is beyond the scope of this manual to explain how to do so.

3.3.1. Example 1 Revisited

Now that we are familiar with the details of the foreign task statement, it might be useful to review the examples of foreign task statements given in §2.1, which introduced the *planck* task. The UNIX example given was as follows:

```
cl> task $planck = $planck.e
```

This is fine, but only provided the *planck* task is called from the directory containing the executable. To enable the executable to be called from any directory we can use a UNIX pathname instead, e.g.,

```
cl> task $planck = $/usr/jones/iraf/tasks/planck.e
```

Alternatively, one could place all such tasks in a certain directory, and either define the pathname of the directory as a shell environment variable to be referenced in the task statement, or include the task's directory in the shell search path. There are many other possibilities, of course, but it would be inappropriate to enumerate them here.

The VMS example given earlier was the following:

```
cl> task $planck = "$planck:==\$disk:[dir...]planck.exe!planck"
```

The command string at the right actually consists of two separate DCL commands separated by the VMS/IRAF DCL command delimiter '**!**'. If we invent a pathname for the executable, we can write down the the first command:

```
$ planck :== $usr\$2:[jones.iraf.tasks]planck.exe
```

This is a DCL command which defines the new DCL foreign task *planck*. We could shorten the CL foreign task statement by moving the DCL declaration to our DCL `LOGIN.COM` file; this has the additional benefit of allowing the task to be called directly from DCL, but is not as self-contained. If this were done the CL task statement could be shortened to the following.

```
cl> task $planck = $foreign
```

The same thing could be accomplished in Berkeley UNIX by defining a cshell *alias* for the task in the user's `.cshrc` file.

3.4. Debugging IMFORT Programs

Programs written and called from within the IRAF environment can be debugged using the host system debug facility without any inconvenience. The details of how to use the debugger are highly dependent upon the host system since the debugger is a host facility, but a few examples should help the reader understand what is involved.

Berkeley UNIX provides two debug tools, the assembly language debugger *adb* and the source language debugger *dbx*. Both are implemented as UNIX tasks and are called from within the IRAF environment as tasks, with the name of the program to be debugged as a command line argument (this example assumes that *adb* is a defined foreign task):

```
cl> adb planck.e
```

The program is then run with a debugger command, passing any command line arguments to the program as part of the debugger run-program command. Programs do not have to be compiled in any special way to be debugged with *adb*; programs should be compiled with *fc -x* to be debugged with *dbx*.

In VMS, the debugger is not a separate task but rather a shareable image which is linked directly into the program to be debugged. To debug a program, the program must first be linked with *fc -x*. The program is then run by simply calling it in the usual way from the CL, with any arguments given on the command line. When the program runs it comes up initially in the debugger, and a debugger command (*go*) is required to execute the user program. Note that if the program is run directly with `run/debug` there is no provision for passing an argument list to the task.

3.5. Calling IMFORT from Languages other than Fortran

Although our discussion and examples have concentrated exclusively on the use of the IMFORT library in host Fortran programs, the library is in fact language independent, i.e., it uses only low level, language independent system facilities and can therefore be called from any language available on the host system. The method by which Fortran subroutines and functions are called from another language, e.g., C or assembler, is highly machine dependent and it would be inappropriate for us to go into the details here. Note that *fc* may be used to compile and link C or assembler programs as well as Fortran programs.

3.6. Avoiding Library Name Collisions

Any program which uses IMFORT is being linked against the main IRAF system libraries, which together contain some thousands of external procedure names. Only a few hundred of these are likely to be linked into a host program, but there is always the chance that a user program module will have the same external name as one of the modules in the IRAF libraries. If such a library collision should occur, at best one would get an error message from the linker, and at worst one would end up with a program which fails mysteriously at run time.

At present there is no utility which can search a user program for externals and cross check these against the list of externals in the IRAF system libraries. A database of external names is however available in the file `lib$names`; this contains a sorted list of all the Fortran callable external names defined by procedures in the *imfort*, *ex*, *sys*, *vops*, and *os* libraries (the *ex* library is however not searched when linking IMFORT programs).

The *match* task may be used to check individual user external names against the name list, or a host utility may be used for the same purpose. For example, to determine if the module *subnam* is present in any of the IRAF system libraries:

```
cl> match subnam lib$names
```

The names database is also useful for finding the names of all the procedures sharing a particular package prefix. For example,

```
cl> match "^cl" lib$names | table
```

will find all the procedures whose names begin with the prefix "cl" and print them as a table (the *lists* package must be loaded first).

4. The IMFORT Library

In this section we survey the procedures provided by the IMFORT interface, grouped according to the function they perform. There are currently four main groups: the command line access procedures, the image access procedures, the vector operators (VOPS), and a small binary file i/o package. With the exception of the VOPS procedures, all of the IMFORT routines were written especially for IMFORT and are not called in standard IRAF programs. The VOPS procedures are standard IRAF procedures, but are included in the IMFORT interface because they are coded at a sufficiently low level that they can be linked into any program, and they tend to be useful in image processing applications such as IMFORT is designed for.

The ANSI Fortran-77 standard requires that all names in Fortran programs have six or fewer characters. To eliminate guesswork, the names of all the IMFORT procedures are exactly six characters long and the names adhere to a **naming convention**. The first one or two characters in each name identify the package or group to which the procedure belongs, e.g., *cl* for the command line access package, *im* for the image access package, and so on. The package prefix is followed by the function name, and lastly a datatype code identifying the datatype upon which the procedure operates, in cases where multiple versions of the procedure are available for a range of datatypes.

package_prefix // function_code // type_suffix

The type suffix codes have already been introduced in the examples. They are the same as are used throughout IRAF. The full set is **[bcsilrdx]**, as illustrated in the following table (not all are used in the IMFORT procedures).

Standard IRAF Datatypes

suffix	*name*	*code*	*typical fortran equivalent*
b	bool	1	`LOGICAL`
c	char	2	`INTEGER*2` (non-ANSI)
s	short	3	`INTEGER*2` (non-ANSI)
i	int	4	`INTEGER`
l	long	5	`INTEGER*4` (non-ANSI)
r	real	6	`REAL`
d	double	7	`DOUBLE PRECISION`
x	complex	8	`COMPLEX`

The actual mapping of IRAF datatypes into host system datatypes is machine dependent, i.e., *short* may not map into INTEGER*2 on all machines. This should not matter since the datatype in which data is physically stored internally is hidden from user programs by the IMFORT interface.

In cases where multiple versions of a procedure are available for operands of different datatypes, a special nomenclature is used to refer to the class as a whole. For example,

```
clarg[cird] (argno, [cird]val, ier)
```

denotes the set of four procedures *clargc*, *clargi*, *clargr*, and *clargd*. The datatype of the output operand (*cval*, *ival*, etc.) must match the type specified by the procedure name.

With the exception of the low level binary file i/o procedures (BFIO), all IMFORT procedures are implemented as subroutines rather than functions, for reasons of consistency and to avoid problems with mistyping of undeclared functions by the Fortran compiler.

4.1. Command Line Access

The command line access procedures are used to decode the arguments present on the command line when the IMFORT program was invoked. This works both when the program is called from the IRAF CL, and when the program is called from the host system command interpreter. The command line access procedures are summarized in Figure 4, below.

```
      clnarg (nargs)
      clrawc (outstr, ier)
clarg[cird] (argno, [cird]val, ier)
```

Figure 4. Command Line Access Procedures

The *clnarg* procedure returns the number of command line arguments; zero is returned if an error occurs or if there were no command line arguments. The *clargc*, *clargi*, etc., procedures are used to fetch and decode the individual arguments; *clargc* returns a character string, *clargi* returns an integer, and so on. A nonzero *ier* status indicates either that the command line did not contain the indexed argument, or that the argument could not be decoded in the manner specified. Character string arguments must be quoted on the command line if they contain any blanks or tabs, otherwise quoting is not necessary. The rarely used *clrawc* procedure returns the entire raw command line as a string.

4.2. Image Access

The image access procedures form the bulk of the IMFORT interface. There are three main categories of image access procedures, namely, the general image management procedures (open, close, create, get size, etc.), the header access procedures (used to get and put the values of header keywords), and the pixel i/o procedures, used to read and write image data.

IMFORT currently supports images of up to three dimensions, of type short-integer or real. There is no builtin limit on the size of an image, although the size of image a particular program can deal with is normally limited by the size of a statically allocated buffer in the user program. IMFORT does not map IRAF virtual filenames, hence host dependent names must be used when running a program which uses IMFORT.

IMFORT currently supports only the OIF image format, and images must be of type short-integer or real. Since normal IRAF programs support images of up to seven disk datatypes with a dimensionality of up to seven, as well as completely different image formats than that expected by IMFORT (e.g., STF), if you are not careful IRAF can create images which IMFORT programs cannot read (don't omit the error checking!). In normal use, however, types short-integer and real are by far the most common and images with more than two dimensions are rare, so these are not expected to be serious limitations.

4.2.1. General Image Access Procedures

The general image access and management procedures are listed in Figure 5. An image must be opened with *imopen* or *imopnc* before header access or pixel i/o can occur. The image open procedures return an *image descriptor* (an integer magic number) which uniquely identifies the image in all subsequent accesses until the image is closed. When the operation is completed, an image must be closed with *imclos* to flush any buffered output, update the image header, and free any resources associated with the image descriptor. The maximum number of images which can be open at any one time is limited by the maximum number of open file descriptors permitted by the host operating system.

New images are created with *imopnc* and *imcrea*. The *imopnc* procedure creates a new copy of an existing image, copying the header of the old image to the new image but not the data. The new copy image must be the same size and datatype as the old image. For complete control over the attributes of a new image the *imcrea* procedure must be used. The *imopnc* operation is equivalent to an *imopen* followed by an *imgsiz* to determine the size and datatype of the old image, followed by an *imcrea* to create the new image, followed by an *imhcpy* to copy the header of the old image to the new image and then two *imclos* calls to close both images.

Note that *imgsiz always returns seven elements in the output array axlen*, regardless of the actual dimensionality of the image; this is so that current programs will continue to work in the future if IMFORT is extended to support images of dimensionality higher than three. Images may be deleted with *imdele*, or renamed with *imrnam*; the latter may also be used to move an image to a different directory. The *imflsh* procedure is used to flush any buffered output pixel data to an image opened for writing.

```
imopen (image, acmode, im, ier)       acmode: 1=RO,3=RW
imopnc (nimage, oim, nim, ier)        acmode: always RW
imclos (im, ier)

imcrea (image, axlen, naxis, dtype, ier)
imdele (image, ier)
imrnam (oldnam, newnam, ier)

imflsh (im, ier)
imgsiz (im, axlen, naxis, dtype, ier)
imhcpy (oim, nim, ier)
impixf (im, pixfd, pixfil, pixoff, szline, ier)
```

Figure 5. General Image Access Procedures

The *impixf* procedure may be used to obtain the physical attributes of the pixel file, i.e., the pixel file name, the one-indexed *char* offset to the first pixel, and the physical line length of an image as stored in the pixel file (the image lines may be aligned on device block boundaries). These parameters may be used to bypass the IMFORT pixel i/o procedures to directly access the pixels if desired (aside from the blocking of lines to fill device blocks, the pixels are stored as in a Fortran array). The BFIO file descriptor of the open pixel file is also returned, allowing direct access to the pixel file via BFIO if desired. If lower level (e.g., host system) i/o facilities are to be used, *bfclos* or *imclos* should be called to close the pixel file before reopening it with the foreign i/o system.

Direct access to the pixel file is not recommended since it makes a program dependent upon the details of how the pixels are stored on disk; such a program may not work with future versions of the IMFORT interface, nor with implementations of the IMFORT interface for different (non-OIF) physical image storage formats. Direct access may be warranted when performing a minimum modification hack of an old program to make it work in the IRAF environment, or in applications with unusually demanding performance requirements, where the (usually negligible) overhead of the BFIO buffer is unacceptable. Note that in many applications, the reduction in disk accesses provided by the large BFIO buffer outweighs the additional cpu cycles required for memory to memory copies into and out of the buffer.

4.2.2. Image Header Keyword Access

The image header contains a small number of standard fields plus an arbitrary number of user or application defined fields. Each image has its own header and IMFORT does not in itself make any association between the header parameters of different images. The header access procedures are summarized in Figure 6. Note that the *imgsiz* procedure described in the previous section is the most convenient way to obtain the size and datatype of an open image, although the same thing can be achieved by a series of calls to obtain the values of the individual keywords, using the procedures described in this section.

```
      imacck (im, keyw, ier)
      imaddk (im, keyw, dtype, comm, ier)
      imdelk (im, keyw, ier)
      imtypk (im, keyw, dtype, comm, ier)

imakw[bcdir] (im, keyw, [bcdir]val, comm, ier)
imgkw[bcdir] (im, keyw, [bcdir]val, ier)
impkw[bcdir] (im, keyw, [bcdir]val, ier)

      imokwl (im, patstr, sortit, kwl, ier)
      imgnkw (kwl, outstr, ier)
      imckwl (kwl, ier)
```

Figure 6. Image Header Access Procedures

Both the standard and user defined header parameters may be accessed via the procedures introduced in this section. The *imacck* procedure tests for the existence of the named keyword, returning a zero *ier* if the keyword exists. New keywords may be added to the image header with *imaddk*, and old keywords may be deleted with *imdelk*. The datatype of a keyword may be determined with *imtypk*. The attributes of a keyword are its name, datatype, value, and an optional comment string describing the significance of the parameter. The comment string is normally invisible except when the header is listed, but may be set when a new keyword is added to the header, or fetched with *imtypk*.

The most commonly used procedures are likely to be the *imgkw* and *impkw* families of procedures, used to get and put the values of named keywords; these procedures require that the keyword already be present in the header. The *imakw* procedures should be used instead of the *impkw* procedures if it is desired that a keyword be automatically added to the header if not found, before setting the new value. Automatic datatype conversion is performed if the requested datatype does not match the actual datatype of the keyword.

The *keyword list* package is the only way to obtain information from the header without knowing in advance the names of the header keywords. The *imokwl* procedure opens a keyword list consisting of all header keywords matching the given pattern, returning a *list descriptor* to be used as input to the other procedures in the package. Successive keyword *names* are returned in calls to *imgnkw*; a nonzero *ier* is returned when the end of the list is reached. The keyword name is typically used as input to other procedures such as *imtypk* or one of the *imgkw* procedures to obtain further information about the keyword. A keyword list should be closed with *imckwl* when it is no longer needed to free system resources associated with the list descriptor.

Standard Image Header User Keywords		
name	*datatype*	*description*
naxis	int	number of axes (dimensionality)
naxis[1:3]	int	length of each axis, pixels
pixtype	int	pixel datatype
datamin	real	minimum pixel value
datamax	real	maximum pixel value
ctime	int	image creation time
mtime	int	image modification time
limtime	int	time min/max last updated
title	string	image title string (for plots etc.)

The keyword list pattern string follows the usual IRAF conventions; some useful patterns are "∗", which matches the entire header, and "i_", which matches only the standard header keywords (the standard header keywords are really named "i_naxis", "i_pixtype", etc., although the "i_" may be omitted in most cases). A pattern which does not include any pattern matching metacharacters is taken to be a prefix string, matching all keywords whose names start with the pattern string.

An image must be opened with read-write access for header updates to have any effect. An attempt to update a header without write permission will not produce an error status return until *imclos* is called to update the header on disk (and close the image).

4.2.3. Image Pixel Access

The IMFORT image pixel i/o procedures are used to get and put entire image lines to N-dimensional images, or to get and put N-dimensional subrasters to N-dimensional images. In all cases the caller supplies a buffer into which the pixels are to be put, or from which the pixels are to be taken. The pixel i/o procedures are summarized in Figure 7.

As shown in the figure, there are four main classes of pixel i/o procedures, the get-line, put-line, get-section, and put-section procedures. The get-line and put-line procedures are special cases of the get/put section procedures, provided for programming convenience in the usual line by line sequential image operator (they are also slightly more efficient than the subraster procedures for line by line i/o). It is illegal to reference out of bounds and *i1* must be less than or equal to *i2* (IMFORT will not flip lines); the remaining subscripts may be swapped if desired. Access may be completely random if desired, but sequential access (in storage order) implies fewer buffer faults and is more efficient.

```
im[gp]l1[rs] (im, buf, ier)
im[gp]l2[rs] (im, buf, lineno, ier)
im[gp]l3[rs] (im, buf, lineno, bandno, ier)
im[gp]s1[rs] (im, buf, i1, i2, ier)
im[gp]s2[rs] (im, buf, i1, i2, j1, j2, ier)
im[gp]s3[rs] (im, buf, i1, i2, j1, j2, k1, k2, ier)
```

Figure 7. Image Pixel I/O Procedures

Type short and type real versions of each i/o procedure are provided. The type real procedures may be used to access images of either type short or type real, with automatic datatype conversion being provided if the disk and program datatypes do not match. The type short-integer i/o procedures may only be used with type short images.

The user who is familiar with the type of image i/o interface which maps the pixel array into virtual memory may wonder why IMFORT uses the more old fashioned buffered technique. There are two major reasons why this approach was chosen. Firstly, the virtual memory mapping technique, in common use on VMS systems, is *not portable*. On a host which does not support the mapping of file segments into paged memory, the entire image must be copied into paged memory when the image is opened, then copied again when the image operation takes place, then copied once again from memory to disk when the image is closed. Needless to say this is very inefficient, particularly for large images, and some of our applications deal with images 2048 or even 6000 pixels square.

Even on a machine that supports mapping of file segments into memory, mapped access will probably not be efficient for sequential access to large images, since it causes the system to page heavily; data pages which will never be used again fill up the system page caches, displacing text pages that must then be paged back in. This happens on even the best systems, and on a system that does not implement virtual memory efficiently, performance may suffer greatly.

A less obvious reason is that mapping the image directly into memory violates the principle of *data independence*, i.e., a program which uses this type of interface has a builtin dependence on the particular physical image storage format in use when the program was developed. This rules out even such simple interface features as automatic datatype conversion, and prevents the expansion of the interface in the future, e.g., to provide such attractive features as an image section capability (as in the real IRAF image interface), network access to images stored on a remote node, support for pixel storage schemes other than line storage mode (e.g., isotropic mappings or sparse image storage), and so on.

The majority of image operations are either sequential whole-image operations or operations upon subrasters, and are just as easily programmed with a buffered interface as with a memory mapped interface. The very serious drawbacks of the memory mapped interface dictate that it not be used except in special applications that must randomly access individual pixels in an image too large to be read in as a subraster.

4.3. Error Handling

The IMFORT error handling mechanism is extremely simple. All procedures in which an error condition can occur return a nonzero *ier* error code if an error occurs. The value of *ier* identifies which of many possible errors actually occurred. These error codes may be converted into error message strings with the following procedure:

```
imemsg (ier, errmsg)
```

It is suggested that every main program contain an error handling section at the end of the program which calls *imemsg* and halts program execution with an informative error message, as in the examples in §2. This is especially helpful when debugging new programs.

4.4. Vector Operators

The vector operators (VOPS) package is a subroutine library implementing a large number of primitive operations upon one dimensional vectors of any datatype. Some of the operations implemented by the VOPS routines are non-trivial to implement, in which case the justification for a library subroutine is clear. Even in the simplest cases, however, the use of a VOPS procedure is advantageous because it provides scope for optimizing all programs which use the VOPS operator, without having to modify the calling programs. For example, if the host machine has vector hardware or special machine instructions (e.g., the block move and bitfield instructions of the VAX), the VOPS operator can be optimized in a machine dependent way to take advantage of the special capabilities of the hardware, without compromising the portability of the applications software using the procedure.

The VOPS procedures adhere to the naming convention described in §4. The package prefix is *a*, the function code is always three characters, and the remaining one or two characters define the datatype or types upon which the procedure operates. For example, *aaddr* performs a vector add upon type real operands. If the character *k* is added to the three character function name, one of the operands will be a scalar. For example, *aaddkr* adds a scalar to a vector, with both the scalar and the vector being of type real.

Most vector operators operate upon operands of a single datatype: one notable exception is the *acht* (change datatype) operator, used to convert a vector from one datatype to another. For example, *achtbi* will unpack each byte in a byte array into an integer in the output array, providing a capability that cannot be implemented in portable Fortran. Any datatype suffix characters may be substituted for the *bi*, to convert a vector from any datatype to any other datatype.

In general, there are are three main classes of vector operators, the *unary* operators, the *binary* operators, and the *projection* operators. The unary operators perform some operation upon a single input vector, producing an output vector as the result. The binary operators perform some operation upon two input vectors, producing an output vector as the result. The projection operators compute some function of a single input vector, producing a scalar function value (rather than a vector) as the result. Unary operators typically have three arguments, binary operators four, and projection operators two arguments and one output function value. For example, *aabsi* is the unary absolute value vector operator, type integer (here, *a* is the input vector, *b* is the output vector, and *npix* is the number of vector elements):

```
aabsi (a, b, npix)
```

A typical example of a binary operator is the vector add operator, *aaddr*. Here, *a* and *b* are the input vectors, and *c* is the output vector:

```
aaddr (a, b, c, npix)
```

In all cases except where the output vector contains fewer elements than one of the input vectors, the output vector may be the same as one of the input vectors. A full range of datatypes are provided for each vector operator, except that there are no boolean vector operators (integer is used instead), and *char* and *complex* are only partially implemented, since they are not sensible datatypes for many vector operations. In any case, the VOPS *char* is the SPP char and should be avoided in Fortran programs.

Once these rules are understood, the calling sequence of a particular VOPS operator can usually be predicted with little effort. The more complex operators, of course, may have special arguments, and some study is typically required to determine their exact function and how they are used. A list of the VOPS operators currently provided is given below (the datatype suffix characters must be added to the names shown to form the full procedure names).

```
  aabs - Absolute value of a vector
  aadd - Add two vectors
 aaddk - Add a vector and a scalar
  aand - Bitwise boolean AND of two vectors
 aandk - Bitwise boolean AND of a vector and a scalar
  aavg - Compute the mean and standard deviation of a vector
  abav - Block average a vector
  abeq - Vector equals vector
 abeqk - Vector equals scalar
  abge - Vector greater than or equal to vector
 abgek - Vector greater than or equal to scalar
  abgt - Vector greater than vector
 abgtk - Vector greater than scalar
  able - Vector less than or equal to vector
```

ablek - Vector less than or equal to scalar
ablt - Vector less than vector
abltk - Vector less than scalar
abne - Vector not equal to vector
abnek - Vector not equal to scalar
abor - Bitwise boolean OR of two vectors
abork - Bitwise boolean OR of a vector and a scalar
absu - Block sum a vector
acht - Change datatype of a vector
acjgx - Complex conjugate of a complex vector
aclr - Clear (zero) a vector
acnv - Convolve two vectors
acnvr - Convolve a vector with a real kernel
adiv - Divide two vectors
adivk - Divide a vector by a scalar
adot - Dot product of two vectors
advz - Vector divide with divide by zero detection
aexp - Vector to a real vector exponent
aexpk - Vector to a real scalar exponent
afftr - Forward real discrete fourier transform
afftx - Forward complex discrete fourier transform
aglt - General piecewise linear transformation
ahgm - Accumulate the histogram of a series of vectors
ahiv - Compute the high (maximum) value of a vector
aiftr - Inverse real discrete fourier transform
aiftx - Inverse complex discrete fourier transform
aimg - Imaginary part of a complex vector
alim - Compute the limits (minimum and maximum values) of a vector
alln - Natural logarithm of a vector
alog - Logarithm of a vector
alov - Compute the low (minimum) value of a vector
altr - Linear transformation of a vector
alui - Vector lookup and interpolate (linear)
alut - Vector transform via lookup table
amag - Magnitude of two vectors (sqrt of sum of squares)
amap - Linear mapping of a vector with clipping
amax - Vector maximum of two vectors
amaxk - Vector maximum of a vector and a scalar
amed - Median value of a vector
amed3 - Vector median of three vectors
amed4 - Vector median of four vectors
amed5 - Vector median of five vectors
amgs - Magnitude squared of two vectors (sum of squares)
amin - Vector minimum of two vectors
amink - Vector minimum of a vector and a scalar
amod - Modulus of two vectors
amodk - Modulus of a vector and a scalar
amov - Move (copy or shift) a vector
amovk - Move a scalar into a vector
amul - Multiply two vectors
amulk - Multiply a vector and a scalar
aneg - Negate a vector (change the sign of each pixel)
anot - Bitwise boolean NOT of a vector
apkx - Pack a complex vector given the real and imaginary parts

apol - Polynomial evaluation
apow - Vector to an integer vector power
apowk - Vector to an integer scalar power
arav - Mean and standard deviation of a vector with pixel rejection
arcp - Reciprocal of a scalar and a vector
arcz - Reciprocal with detection of divide by zero
arlt - Vector replace pixel if less than scalar
argt - Vector replace pixel if greater than scalar
asel - Vector select from two vectors based on boolean flag vector
asok - Selection of the Kth smallest element of a vector
asqr - Square root of a vector
asrt - Sort a vector in order of increasing pixel value
assq - Sum of squares of a vector
asub - Subtract two vectors
asubk - Subtract a scalar from a vector
asum - Sum of a vector
aupx - Unpack the real and imaginary parts of a complex vector
awsu - Weighted sum of two vectors
awvg - Mean and standard deviation of a windowed vector
axor - Bitwise boolean XOR (exclusive or) of two vectors
axork - Bitwise boolean XOR (exclusive or) of a vector and a scalar

A non-trivial example of the use of vector operators is the case of bilinear interpolation on a two dimensional image. The value of each pixel in the output image is a linear sum of the values of four pixels in the input image. The obvious solution is to set up a do-loop over the pixels in each line of the output image, computing the linear sum over four pixels from the input image for each pixel in the output line; this is repeated for each line in the output image.

The solution using the VOPS operators involves the *alui* (vector look up and interpolate) and *awsu* (weighted sum) vector operators. A lookup table defining the X-coordinate in the input image of each pixel in a line of the output image is first generated. Then, for each line of the output image, the two lines from the input image which will contribute to the output image line are extracted. *Alui* is used to interpolate each line in X, then *awsu* is used to form the weighted sum to interpolate in the Y direction. This technique is especially efficient when bilinear interpolation is being used to expand the image, in which case the *alui* interpolated X-vectors, for example, are computed once but then used to generate several lines of the output image by taking the weighted sum, a simple and fast operation. When moving sequentially up through the image, the high X-vector becomes the low X-vector for the next pair of input lines, hence only a single call to *alui* is required to set up the next region.

The point of this example is that many or most image operations can be expressed in terms of primitive one dimensional vector operations, regardless of the dimensionality of the image being operated upon. The resultant algorithm will often run more efficiently even on a conventional scalar machine than the equivalent nonvectorized code, and will probably run efficiently without modification on a vector machine.

Detailed specification sheets (manual pages) are not currently available for the VOPS procedures. A summary of the calling sequences is given in the file `vops$vops.syn`, which can be paged or printed by that name while in the CL, assuming that the system has not been stripped and that the sources are still on line. The lack of documentation is really not a problem for these operators, since they are all fairly simple, and it is easy to page the source file (in the *vops* directory) to determine the exact calling sequence. For example, to examine the source for *awsu*, type

```
cl> page vops$awsu.gx
```

to page the generic source, regardless of the specific datatype of interest. If you have trouble deciphering the generic source, use `xc -f file.x` to produce the Fortran translation of one of the type specific files in the subdirectories `vops$ak` and `vops$lz`.

4.5. Binary File I/O (BFIO)

The IMFORT binary file i/o package (BFIO) is a small package, written originally as an internal package for use by the IMFORT image i/o routines for accessing header and pixel files (the VOS FIO package could not be used in IMFORT without linking the entire IRAF/VOS runtime system into the Fortran program). Despite its original conception as an internal package, the package provides a useful capability and is portable, hence has been included in the IMFORT interface definition. Nonetheless, the user should be warned that BFIO is a fairly low level interface and some care is required to use it safely. If other suitable facilities are available it may be better to use those, although few interfaces will be found which are simpler or more efficient than BFIO for randomly accessing pre-existing or preallocated binary files.

The principal capability provided by BFIO is the ability to randomly access a binary file, reading or writing an arbitrary number of char-units of storage at any (one-indexed) char offset in the file. The file itself is a non-record structured file containing no embedded record manager information, hence is suitable for access by any program, including non-Fortran programs, and for export to other machines (this is usually not the case with a Fortran unformatted direct access file). Unlike the mainline IMFORT procedures, many of the BFIO procedures are integer functions returning a positive count value if the operation is successful (e.g., the number of char units of storage read or written), or a negative value if an error occurs. Zero is returned for a read at end of file.

```
        bfaloc (fname, nchars, status)
    fd = bfopen (fname, acmode, advice)    acmode: 1=RO,3=RW,5=NF
        bfclos (fd, status)                advice: 1=random,2=seq

nchars = bfread (fd, buf, nchars, offset)
nchars = bfwrit (fd, buf, nchars, offset)

nchars = bfbsiz (fd)
nchars = bffsiz (fd)
  chan = bfchan (fd)
  stat = bfflsh (fd)
```

Figure 8. Low Level Binary File I/O Procedures

BFIO binary files may be preallocated with *bfaloc*, or created with *bfopen* and then initialized by writing at the end of file. Preallocating a file is useful when the file size is known in advance, e.g., when creating the pixel file for a new image. The contents of a file allocated with *bfaloc* are uninitialized. To extend a file by writing at the end of file the file size must be known; the file size may be obtained by calling *bffsiz* on the open file.

Before i/o to a file can occur, the file must be opened with *bfopen*. The *bfopen* procedure returns as its function value an integer *file descriptor* which is used to refer to the file in all subsequent accesses until the file is closed with *bfclos*. Binary data is read from the file with *bfread*, and written to the file with *bfwrit*. Any amount of data may be read or written in a single call to *bfread* or *bfwrit*. All user level i/o is synchronous and data is buffered

internally by BFIO to minimize disk transfers and provide for the blocking and deblocking of data into device blocks. Any buffered output data may be flushed to disk with *bfflsh*. The function *bfchan* returns the descriptor of the raw i/o channel as required by the IRAF binary file driver.

BFIO manages an internal buffer, necessary for efficient sequential i/o and to hide the device block size from the user program. Larger buffers are desirable for sequential i/o on large files; smaller buffers are best for small files or for randomly accessing large files. The buffer size may be set at *bfopen* time with the *advice* parameter. An *advice* value of 1 implies random access and causes a small buffer to be allocated; a value of 2 implies sequential access and causes a large buffer to be allocated. Any other value is taken to be the actual buffer size in chars, but care must be used since the value specified must be some multiple of the device block size, and less than the maximum transfer size permitted by the kernel file driver. Note that when writing at end of file, the full contents of the internal buffer will be written, even if the entire buffer contents were not written into in a *bfwrit* call. The buffer size in chars is returned by *bfbsiz*.

Since BFIO is a low level interface, the file offset must always be specified when reading from or writing to the file, even when the file is being accessed sequentially. Contrary to what one might think, file offsets are one-indexed in the Fortran tradition, and are specified in units of *chars*. Do not confuse *char* with the Fortran `CHARACTER`; *char* is the fundamental unit of storage in IRAF, the smallest datum which can be accessed as an integer quantity with the host Fortran compiler, normally `INTEGER*2` (16 bits or two bytes on all current IRAF hosts).

Appendix: Manual Pages for the Imfort Procedures

This section presents the ‘‘manual pages’’ for the IMFORT and BFIO procedures. The manual pages present the exact technical specifications of each procedure, i.e., the procedure name and arguments (not necessarily obvious in the case of a typed family of procedures), the datatypes and dimensions of the arguments, and a precise description of the operation of the procedure. Each procedure is presented on a separate page for ease of reference.

The following conventions have been devised to organize the information presented in this section:

- The manual pages are presented in alphabetical order indexed by the procedure name.
- A single manual page is used to present an entire family of procedures which differ only in the datatype of their primary operand. The name on the manual page is the generic name of the family, e.g., *clargi*, *clargr*, etc., are described in the manual page *clarg*.
- In some cases it makes sense to describe several related procedures with a single manual page. An example is the keyword-list package, consisting of the procedures *imokwl*, *imgnkw*, and *imckwl*. In such a case, since the procedures have different names the manual page for the group is duplicated for each procedure in the group, so that the user will not have to guess which name the manual page is filed under.
- The *synopsis* section of each manual page defines the calling sequence of each procedure, the datatypes and dimensions of the arguments, and notes whether each argument is an input argument (`#I`) or an output argument (`#O`).
- The *return value* section describes the conditions required for successful execution of the procedure, normally indicated by a zero status in *ier*. A symbolic list of the possible error codes is also given. The numeric values of these error codes are defined in `imfort$imfort.h` and in `lib$syserr.h`, but the exact numeric codes should be used only for debugging purposes or passed on to the *imemsg* procedure to get the error message string. The numeric error codes are likely to change in future versions of the interface hence their values should not be "wired into" programs.

Manual pages for the VOPS procedures are not included since VOPS is not really part of the IMFORT interface, and it is not yet clear if the VOPS procedures are complex enough to justify the production of individual manual pages.