An Introductory User's Guide to IRAF SPP Programming

Rob Seaman

National Optical Astronomy Observatories†
Tucson, Arizona 85726

This document presents a tutorial and overview of programming in the IRAF Subset Pre-Processor (SPP) language using the various procedure calls that constitute the IRAF Virtual Operating System (VOS). Some of the advantages of programming in SPP are portability from machine to machine, durability as computers and operating systems evolve, access to IRAF data structures such as Command Language parameters and the various image formats, and access to large libraries of useful routines of every description in both source code and compiled versions.

While a full exposition of the IRAF SPP/VOS programming environment is beyond the scope of this guide, the author hopes to communicate enough of the substance of SPP programming to get an interested scientist or programmer over the first few hurdles. An associated IRAF package is available from NOAO that contains the examples from this document. This package can be modified and extended by the user to avoid the monotony and mistakes of duplicating the author's efforts.

Prepared for V2.10 of the Image Reduction and Analysis Facility (IRAF).

Please send comments to seaman@noao.edu.

October 30, 1992

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

Table of Contents

1. **Introduction** 1
   - Typographical Notes
   - IRAF File Name Conventions

2. **Compiling and Executing an SPP Task** 3
   - IRAF Online Help

3. **Tasks with Parameters** 7
   - The Fibonnaci Task
   - Imreplace (First Version)
   - Imreplace (Second Version)

4. **An Advanced Example** 17

5. **Modifying an Existing Task** 25
   - Summary

6. **The IRAF VOS Interfaces** 41
   - The Command Language I/O Interface (*clio*)
   - The File I/O Interface (*fio*)
   - The Image I/O Interface (*imio*)
   - The Memory I/O Interface (*memio*)
   - The Graphics I/O Interface (*gio*)
   - Vector Operators (*vops*)
   - Miscellaneous (*etc*)
   - The Formatted I/O Interface (*fmtio*)
   - Intrinsic Functions
   - Other Interfaces and Library Routines
   - Arguments and Return Values

7. **Making an IRAF Package** 55
   - Installing the **examples** Package
   - Adding a Task to the Package

Appendix I: **Topics Not Discussed**

Appendix II: **References**

Appendix III: **Help Pages for the Examples**

Appendix IV: **An SPP/VOS Quick Reference Card**

An Introductory User's Guide to IRAF SPP Programming

Rob Seaman

National Optical Astronomy Observatories†
Tucson, Arizona 85726

1. Introduction

There are several good reasons to write IRAF programs in SPP. The single reason most often offered is portability from machine to machine. This is a good reason, but of course the push in the computer industry for standards and "open systems" has made portability across computers less critical than it once was. It remains an important consideration, however, since even small differences between operating systems or compilers can result in subtle problems that must be resolved in order to maintain a port of IRAF to a given computer, for instance. In other words, not all astronomers own Sun workstations, and equally to the point, a Sun 3 is a very different computer than a SparcStation.

Another good reason is the *chronological* portability that has given IRAF time to mature and to adapt to the changing machines and operating systems of the last decade. Carefully written SPP programs are extremely durable and rarely become obsolete.

Since IRAF is a mature system, the immediate benefit for the programmer is access to well developed data structures such as IRAF Command Language parameters and image file formats and to large libraries of useful and well tested system and mathematical routines of every description. You don't have to re-invent the wheel when programming in SPP.

At this point, we have referred to IRAF SPP programs without defining the terms. IRAF is the *Image Reduction and Analysis Facility* that has been developed by and is maintained by NOAO. SPP is the *Subset Pre-Processor* language, which serves as a buffer against the vagaries of any particular computer programming language implementation. Rather than attempt to summarize the reasoning behind the selection of SPP for IRAF, the reader is directed to the document, *The Role of the Preprocessor*, by Doug Tody.

The SPP language is only a third of the recipe that was used to achieve IRAF's high degree of portability and thus, to a large measure, its success.

The second ingredient is the IRAF VOS (or *Virtual Operating System*). An SPP *task* never accesses the computer's operating system directly, but rather uses the facilities of the VOS, which in turn talks to the particular computer's operating system using a small interface known as the IRAF *kernel*. It is the VOS which includes the very large number of useful routines that were mentioned above. It is more correct to call an IRAF program an "SPP/VOS" program than an "SPP" program. The complexity that can vex new IRAF programmers is the size of the VOS, not the rather simple and basically familiar nature of the SPP language itself.

The third ingredient is the set of IRAF bootstrap utilities that is supplied for each supported computer. These utilities include the SPP compiler, **xc**, and the **mkpkg** program which provides abilities that are similar to, but enhanced relative to the usual Unix **make** command (and, of course, **mkpkg** works on non-Unix machines, *e.g.*, VMS Vaxes, as well). The bootstrap utilities are not part of IRAF, *per se*, but are coded in C with the normal level of portability that can be achieved with a careful design, *i.e.*, moderate code changes are usually required for the IRAF Group to support a new machine or operating system version.

---

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

One last bit of business: the word "task" was used above. An IRAF *task* is a program that is meant to run in the environment provided by the IRAF Command Language, or *CL*. Tasks fall into three general categories: compiled SPP programs, interpreted CL *scripts*, and imported *foreign* tasks. You can read about scripts in *An Introductory User's Guide to IRAF Scripts*, by Ed Anderson and Rob Seaman, and you can read about foreign tasks, and also about the *IMFORT* library of routines for accessing IRAF images from host Fortran (or C) programs, in *A User's Guide to Fortran Programming in IRAF — The IMFORT Interface*, by Doug Tody.

1.1. Typographical Notes

- Text in the `Courier` (*i.e.*, `teletype`) font indicates programs that are contained in a file, information typed by the computer, or the names of variables, of procedures, or of parts of the SPP language.
- Text in the **`Courier Bold`** font indicates what the user types on the keyboard.
- Text in the *`Courier Italic`* font indicates an identifier that should be substituted with the appropriate text string value.
- Text in the *Roman Italic* font is used for emphasis or for the names of files.
- Text in the **Roman Bold** font is used for the names of IRAF tasks and packages, host programs, section and example headings, and for highlighting important points.
- Examples are labeled with parenthesized numbers, *e.g.*, (1), at the end of some lines. These are used solely to reference the notes that follow and are not part of the examples. Corresponding labels are included in the comments of the **examples** package.

1.2. IRAF File Name Conventions

- Files containing SPP source code always have an extension of *.x*. SPP files that contain entry point procedures for IRAF tasks often (but not always) have *t_* prepended.
- IRAF makes it possible to use both SPP and non-SPP code (for instance, Fortran or C) in a single executable. Fortran programs have an extension of *.f* when accessed from within IRAF. C programs have an extension of *.c*.
- IRAF makes use of a number of *header* or *include* files which define constants and data structures. These files have an extension of *.h*.
- Compiled object files have an extension (inside of IRAF) of *.o*, object library files have an extension of *.a*, and linked executable files typically have an extension of *.e*,
- IRAF CL scripts have an extension of *.cl*, and parameter files an extension of *.par*.
- Files associated with IRAF help have extensions such as *.hlp, .hd, .men*, and *.mip*.
- Cursor loop tasks often provide access to online "keystroke" help files. These files have an extension of *.key*.
- Subdirectories in the IRAF tree that contain source code are often called *src*. Those containing documentation are often called *doc*. Library subdirectories are called *lib*. Directories containing executable binary files are called *bin* or *bin.*`arch`, where *`arch`* specifies a particular machine architecture.
- IRAF source code directories each contain a *mkpkg* file which describes how to compile, link, and install the code in that directory.
- Due to IRAF filename mapping, the actual host level names may differ. For example under VMS, Fortran programs have an extension of *.for*, object files have an extension of *.obj*, executables have an extension of *.exe*, and object libraries an extension of *.olb*. The names within IRAF are as described above, however.

2. Compiling and Executing an SPP Task

We will jump right in with an example. In SPP, the customary "Hello, world!" program looks something like this:

```
# HELLO -- Sample program introducing SPP.

task hello = t_hello_world

procedure t_hello_world ()

begin
        call printf ("Hello, world!\n")
end
```

The first step is to edit the program above into the file *hello.x* using your favorite editor. You could also simply retrieve the file from the *examples.tar.Z* archive file as described in §7.

The second step is to compile the program into the executable file *hello.e*:

```
cl> xc hello.x
```

The third step is to declare the new task (the name for an IRAF program) so that it is accessible from within IRAF:

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

The task can now be run from the CL prompt:

```
cl> hello
Hello, world!
```

Even as simple as it is, **hello** is a fully featured IRAF task that can be used with such IRAF facilities as output redirection:

```
cl> hello > filename
```

and IRAF job control, for example, to run the task in the background:

```
cl> hello &
```

or to measure its execution time:

```
cl> $hello
Hello, world!
Time (hello) 0.017 0:00 99%
```

As you read this guide, keep in mind that much of the power of SPP/VOS programming is implicit, requiring little or no action on the part of the programmer beyond following the rules.

We will now go back over the steps involved in creating an IRAF SPP task in more detail. A few general comments are in order:

- **Semicolons are NOT used to terminate each statement**, although a statement consisting only of a "`;`" can be used to indicate a null statement as in the C language. This may be useful, for instance, in the body of a simple loop.
- **Blank lines are permitted** (and encouraged!) to make programs more readable.
- **A statement will be continued onto the next line if it ends with a comma, an operator, or a backslash (\).**
- **Indentation is free form** and is left to the design or whim of the programmer. One could do worse than to emulate the style of the examples in this guide, which adhere to the IRAF project standards as described in *IRAF Standards and Conventions*, by Elwood Downey, *et. al.*

There are also comments specific to every line of the program:

```
# HELLO -- Sample program introducing SPP.                  (1)

task hello = t_hello_world                                  (2)

procedure t_hello_world ()                                  (3)

begin                                                       (4)
        call printf ("Hello, world!\n")                     (5)
end                                                         (4)
```

(1) **Comments begin with the `#` character and extend to the end of the line.**

(2) SPP programs are not intended to be run outside of IRAF†, but rather as *tasks* under the Command Language. The SPP `task` statement links an automatically generated command interpreter (called `sys_runtask`) into your program. The interpreter allows communication between the IRAF CL and the program. **There may be more than one task within a single IRAF executable.** This is one reason that the jargon term *task* is used, rather than just calling them programs.

(3) The `procedure` statement is used to declare all subroutines and functions in SPP. Usually the main procedure for a task is given a name that begins with *t_*. It was this name, `t_hello_world`, that was referred to in the preceding `task` statement, while the outside world, *e.g.*, the CL, knows the task by the name `hello`.

(4) The beginning and ending of the executable portion of an SPP procedure are indicated by the `begin` and `end` statements.

(5) Untyped procedure calls (but not function calls) require that the `call` keyword be used when they are referenced (contrast this to C language usage). In this case the procedure `printf` is being called to print a string literal, `"Hello, world!\n"`, to the standard output (the newline character is specified within the string by "`\n`"). All IRAF tasks have the usual Unix-like standard input, standard output, and standard error (or `STDIN`, `STDOUT`, and `STDERR`) files predefined and opened for access with no hoopla.

---

†However, standalone execution from the host operating system prompt is supported for special purpose applications. See Appendix I for more details.

Compiling the program also requires some explanation. On a Sun, the **xc** command will generate something like the following running output:

```
cl> xc hello.x                                          (1)
hello.x:                                                (2)
   sys_runtask:
   t_hello_world:
hello.f:                                                (3)
        sysruk:                                         (4)
        thelld:                                         (4)
link:
```

(1) The IRAF **xc** compiler is used to compile SPP programs. It may also be used to compile C or Fortran programs. It is an IRAF bootstrap program and may be used either inside or outside IRAF. When no command line switches are specified, the SPP file(s) will be both compiled and linked. The resulting executable file is named by replacing the file extension with a *.e*, in this case, *hello.e*.

(2) SPP source files must have an extension of *.x*, C source files must have an extension of *.c*, and Fortran source files must have an extension of *.f*. These are the extensions used within the IRAF environment — the host file extensions may be subject to IRAF filename mapping, for instance Fortran files have a *.for* extension under VMS, but are known inside of IRAF by a *.f* extension. Read the IRAF help page for **xc** for more information.

(3) The SPP language is currently implemented as a preprocessor for Fortran (on most machines), hence the name *Subset Pre-Processor* language.†

(4) To allow the use of long names for variables and procedures, names that are longer than six characters (all that are permitted in Fortran 66) are translated into six or fewer characters. The rule is that the first five characters and the last character of the name are contracted. Underscores are ignored in the count. Given these two rules, `sys_runtask` contracts to `sysruk` and `t_hello_world` contracts to `thelld`. This scheme is normally transparent to the programmer, but **you must choose variable and procedure names that map to unique identifiers**.

A better way to declare the **hello** task to the CL is to specify the full pathname to the executable. If this is not done, the task will only work when you are in the directory containing the compiled program. For example, the command:

```
cl> task $hello = home$spp/hello.e
```

will let **hello** work no matter what directory is current. This assumes that the executable file is located in the subdirectory, *spp*, of your IRAF login directory, *home$*. Adjust the pathname if your directory is different. The dollar sign (`$`) that is prepended to the task name, `$hello`, tells the CL that this task does not have an associated parameter file. To allow **hello** to be automatically defined the next time you log into IRAF, place the `task` statement into either your *login.cl* or *loginuser.cl* file.

†Actually it is a two step preprocessor, producing a dialect of that venerable (and highly stable) language, RATFOR, as the first intermediate step. The Fortran that is produced as the second step is a vanilla subset of the Fortran 66 standard that has been shown to be digestible to even the strangest of host Fortran compilers. Note that the compilation process is quite speedy, even with all the conversions going on, and that no runtime speed penalty is imposed, especially with the use of modern optimizers.

2.1. IRAF Online Help

At this point, it is appropriate to introduce some of the pertinent online help information that is available to you. Both the **xc** and **task** commands that were mentioned above have IRAF help pages that you may find useful. Use the IRAF **help** or **phelp** commands to access these:

```
cl> help xc    or    cl> phelp task
```

(The **phelp** command allows scrolling the help page backward.) Two other help pages to become familiar with are those for **help** itself (`cl> help help`) and for the **references** task. The **help** task can be used to do other useful things, as well as simply listing out a "help" page, such as viewing the IRAF source code for a task. You will see examples of this later on. The **references** task is useful for locating IRAF tasks that are related to a particular keyword:

```
cl> references cache
          cache - Cache parameter files, or print the current cache list
      flprcache - Flush the process cache
      mkttydata - Build cache for termcap/graphcap device entries
        prcache - Show process cache, or lock a process into the cache
```

The first time you use **references**, you should specify the parameter `updquick=yes` (abbreviated `upd+` on the command line below) to speed up later searches dramatically:

```
cl> refer upd+
generating new quick-reference file uparm$quick.ref...
```

There are also several general topic help pages that can be of use as background for SPP programmers. A few of these are **parameters**, **cursors**, and **commands**. The **softools** package and the **xtools** library contain help pages on several different topics:

```
cl> help softools
        generic - Preprocess a generic source file
     hdbexamine - Examine a help database
          lroff - Lroff (line-roff) text formatter
       mkhelpdb - Make (compile) a help database
      mkmanpage - Make a manual page
          mkpkg - Make or update an object library or package
         mktags - Tag all procedure declarations in a set of files
      mkttydata - Build cache for termcap/graphcap device entries
          rmbin - Find/delete binary files in subdirectories
        rmfiles - Find/delete files in subdirectories
           rtar - Read a TAR format archive file
           wtar - Write a TAR format archive file
             xc - Compile and/or link a program
          xyacc - Build an SPP language parser
```

Other documentation is scattered throughout the system. One good place to look is in the *iraf$doc* directory, although you may be overwhelmed by the details. For instance, to examine the *crib.hlp* file in that directory:

```
cl> help doc$crib.hlp file+
```

The IRAF help database will be discussed in §7. Not all help files are installed in this database, for instance, any help files that you may be writing for tasks that you are working on. The `file+` specification will let you examine the formatted output from these files.

3. Tasks with Parameters

The examples in this section will use progressively more of the facilities of the IRAF VOS. The VOS, or *Virtual Operating System*, consists of a very large number of machine independent procedures (organized into several coherent interfaces) that supply much more than the normal facilities of a host computer operating system. By programming on top of the VOS, the program is insulated from the peculiarities of the particular machine upon which IRAF is running, for instance, the identical IRAF source code can run under VMS and Unix.

An example of a VOS interface that is particularly useful is the Command Language Input/Output interface, or CLIO. Any task that is meant for production use will likely require CL parameters for specifying the critical information that the task needs at runtime. The CL parameter mechanism supplies a much more flexible replacement for the simple command line switches of many Unix or VMS programs, for the question and answer interrogation of the user by other programs, and for the cryptic configuration files that still other programs rely on.

3.1. The Fibonnaci Task

The first example will have just a single parameter. The **fibonnaci** task prints out the first N terms of the Fibonnaci sequence calculated using two different algorithms. The SPP source code may be typed into the file *fibonnaci.x* as listed below, or like the “Hello, world!” example and all examples in this guide, can be found in the **examples** package tar file as described in §7.

examples$src/fibonnaci.x:

```
# FIBONNACI -- print the first N Fibonnaci numbers.

task fibonnaci = t_fibonnaci

include <mach.h>                                                (1)

define  MAX_TERMS       50                                      (2)

procedure t_fibonnaci ()

int     nterms, fib, farray[MAX_TERMS], n                       (3)
double  dfib, phi

int     clgeti()                                                (4)

begin
        nterms = min (clgeti ("nterms"), MAX_TERMS)             (5)
        phi = (1 + sqrt (5.d0)) / 2                             (6)

        call printf (" N\t Algebraic\tSequence\n")

        do n = 1, nterms {                                      (7)
            dfib = phi ** n / sqrt (5.d0)                       (8)
            if (dfib > MAX_INT)
                break
            fib = nint (dfib)

            if (n <= 2)                                         (9)
                farray[n] = 1
            else
                farray[n] = farray[n-1] + farray[n-2]

            call printf ("%2d\t%10d\t%d\n")                    (10)
                call pargi (n)
                call pargi (fib)
                call pargi (farray[n])
        }
end
```

There are a number of points to go over for even such a brief task as this:

(1) We are going to need to know the maximum size of an integer on whatever computer the task is compiled on. The `<mach.h>` system include file contains this and other information about the particular IRAF machine. The angle brackets (<>) indicate that the *xc* compiler should look in the standard IRAF location (typically the *lib$* or *hlib$* directories) to find the include (or "header") file.

(2) On a typical modern machine with 32 bit integers, integer precision is exhausted before the 50*th* Fibonnaci number is reached (N=46 to be precise). Rather than explicitly declare the size of any needed arrays or loop control variables to be 50, it is better to define a symbolic constant to do the job. Better yet would be to dynamically allocate an array of the correct size, as needed. Dynamic memory management is discussed later.

(3) **SPP requires that all variables be declared.**

The iterative algorithm uses the integer array `farray[]` to contain the successive terms of the Fibonnaci sequence. **Array definitions and references are indicated with square brackets.**

(4) External functions (but not "subroutines") must be declared to be the proper datatype by the calling procedure. The intrinsic math functions should *not* be declared.

(5) The task begins by requesting the number of Fibonnaci terms desired. The `nterms` parameter is declared as a *query* parameter in the file *fibonnaci.par*, described below. That `nterms` is a query mode parameter means that unless the value for `nterms` is given by the user on the command line, an interactive prompt will be generated asking for the information. Information that the task must have that is likely to change from one execution of the task to the next should be specified by a query parameter.

Getting the query parameters should typically be the first thing that a task does. This avoids making the user wait while the task executes up to the point that a parameter value is actually needed.

Note that we explicitly limit the value of `nterms` to the maximum allowed. We don't have to limit the minimum since the `do` loop later in the program will be skipped for values of nterms less than one. This constraint on the input is just a precaution since the allowable range of values for the `nterms` parameter is specified in the associated parameter file.

(6) **SPP intrinsic functions are generic, meaning the same name is used for the function no matter the datatype that is returned, which is (typically) the same as the datatype of the arguments.** To preserve full precision for large integer values of the Fibonnaci numbers, the algorithm requires double precision accuracy.

Note that for intrinsic functions that take more than one argument, all arguments should be the same datatype to ensure portability. A complete list of the supported SPP intrinsic functions and their arguments can be found in §6 and on the *SPP/VOS Quick Reference Card*.

(7) SPP supports several control loop structures, including the C-like `while`, `for`, and `repeat...until` constructs, and also the Fortran-like `do` loop. The `do` loop is the most straightforward and should typically compile and optimize to the fastest object code. Note that SPP control structures only apply to the block of code that follows immediately. This is either a single (but perhaps compound, *e.g.*, another `do` loop) statement or a block surrounded by braces (`{}`).

(8) In case you are interested, the actual closed form Fibonnaci expression is:

$$\sqrt{5} \cdot F(N) = \phi^N - (-\phi)^{-N} \qquad \textit{where} \qquad \phi = \frac{1 + \sqrt{5}}{2}$$

The second term quickly approaches zero and may be neglected for all positive values of N, if the result is passed through the `nint` (nearest integer) function. The golden mean, ϕ, is the root of the equation: $x^2 - x - 1 = 0$.

Note how the double precision result is explicitly checked to verify that it will fit into an integer variable on the particular machine before the result is rounded to the nearest integer.

(9) SPP uses C like boolean operators: `<`, `<=`, `>`, `>=`, `==`, `!=`, `!`, `&&`, and `||`,. Again, note that SPP control structures apply either to the single following statement or to a compound statement block surrounded by braces (`{}`).

(10) The SPP `printf` formatting directives are similar in form to C language usage, but the particular directives are different. You should be careful to only consult the IRAF documentation when fiddling with your output formatting. A list of the formatting directives can be found in §6 and on the *SPP/VOS Quick Reference Card*.

The current SPP `printf` implementation is incomplete. Instead of supplying a variable number of arguments directly to `printf` to specify the different values to be formatted and printed, the arguments are supplied in subsequent `parg_` calls, one per formatting directive. The `parg_` calls should follow immediately after the corresponding `printf` and should be indented one level. These rules are not absolutely required by the preprocessor, but are good standards to follow.

The parameter declaration file that is associated with *fibonnaci.x* should be named *fibonnaci.par*. You can either edit the following single line into that file in your work directory (for instance, *home$spp*), or you can copy it from the **examples** package.

examples$src/fibonnaci.par:

```
nterms,i,a,,1,50,Number of terms in the Fibonnaci sequence
```

Each line of a parameter file consists of seven comma separated fields. These are:

- name of the parameter
- parameter datatype:
  integer (`i`), real (`r`), boolean (`b`), or a character string (`s`)†
- access mode:
  hidden (`h`), query (`q`), or automatic (`a`)
- default value
- minimum allowed value (also enumerated values)
- maximum allowed value
- interactive prompt

The parameter file can be considered a disk representation of a runtime data structure. Before your task is executed the CL has the responsibility for maintaining this data structure. While your task executes these parameter values can be examined and even modified by your program. The ensuing behavior depends on access type (*hidden*, *query*, or *automatic*) and on whether the parameter was specified on the command line or not. For more details, read the IRAF **parameter** help page.

†More exotic types are `gcur` for the graphics cursor, `imcur` for the image cursor, `struct` for a character string that scans to include the rest of a line of input, `ukey` for unbuffered or *raw* input, and `file` for a string that is constrained to be a legal file name (this is currently just the same as a normal string). Any of the basic datatypes can be declared as *list directed* by prepending an asterisk (`*`) to the type. A list directed parameter may be assigned the name of a file from which to read the desired value for the parameter.

The command to compile the task is very similar to the “Hello, world!” example:

```
cl> xc fibonnaci.x
```

However, the command to declare the task to the CL is slightly different for a task that has a parameter file. There should be no initial dollar sign (`$`) prepended to the task name in the `task` statement:

```
cl> task fibonnaci = home$spp/fibonnaci.e
```

In addition, the parameter file, *fibonnaci.par*, must be placed in the same directory as specified for the executable so that the CL can locate it.

The task can be run in two ways. By specifying the `nterms` parameter on the command line:

```
cl> fibonnaci 8
 N       Algebraic      Sequence
 1               1      1
 2               1      1
 3               2      2
 4               3      3
 5               5      5
 6               8      8
 7              13      13
 8              21      21
```

Or by allowing the task to prompt you for the value of `nterms`:

```
cl> fibonnaci
Number of terms in the Fibonnaci sequence (1:50) (12): 2
 N       Algebraic      Sequence
 1               1      1
 2               1      1
```

Note that the prompt includes the range of acceptable input values as well as the default value that was learned from the previous execution of the task.

3.2. Imreplace (First Version)

The next two examples are variants of the **imreplace** task from the **proto** package. This task allows the user to specify a new value for all pixels in an image whose current value lies between a lower and an upper threshold. Our first version of the task requires the user to enter specific pixel thresholds, rather than allowing a threshold to default to the image minimum or maximum. Features will be added between the two examples. The **proto** package version of this task lacks some capabilities that we will implement in our examples. On the other hand, for the purposes of the examples we have simplified the handling of multiple image datatypes.

The first version of the task is in the file *examples$src/imreplace1.x* in the *example.tar.Z* file. The associated parameter declaration file is contained in *examples$src/imreplace1.par* and will be listed first.

examples$src/imreplace1.par:

```
# Parameters for the imreplace task.  [FIRST VERSION]
image,s,a,,,,The image to be modified
value,r,a,,,,Constant for the replacement operation
lower,r,h,,,,Lower limit of the replacement window
upper,r,h,,,,Upper limit of the replacement window
```

examples$src/imreplace1.x:

```
# IMREPLACE [FIRST VERSION] -- replace pixel values within thresholds.

task imreplace1 = t_imreplace1

include <imhdr.h>                                                (1)

procedure t_imreplace1 ()

char    image[SZ_FNAME]                                          (2)
real    value, lower, upper

pointer im, buf1, buf2
int     npix, nlines, linenum

pointer immap(), imgl2r(), impl2r()
real    clgetr()

begin
        call clgstr ("image", image, SZ_FNAME)

        value = clgetr ("value")                                 (3)
        lower = clgetr ("lower")
        upper = clgetr ("upper")

        im = immap (image, READ_WRITE, 0)                        (4)

        npix = IM_LEN(im,1)                                      (5)
        nlines = IM_LEN(im,2)

        do linenum = 1, nlines {
            buf1 = imgl2r (im, linenum)                          (6)
            buf2 = impl2r (im, linenum)

            call arepr (Memr[buf1], Memr[buf2], npix,            (7)
                lower, upper, value)
        }

        call imunmap (im)                                        (8)
end


# AREPR -- Copy a to b, replacing values between floor & ceiling.

procedure arepr (a, b, n, floor, ceiling, newval)                (9)

real    a[n]                    #I input array                   (10)
real    b[n]                    #O output array
int     n                       #I size of the arrays
real    floor, ceiling          #I replacement limits
real    newval                  #I replacement constant

int     i

begin
        do i = 1, n
            if (a[i] >= floor && a[i] <= ceiling)
                b[i] = newval
            else
                b[i] = a[i]
end
```

Notes on this example:

(1) We will need to know the size of the input image. The way to obtain this information is to directly access the proper fields in the data structure that represents the image header. The `<imhdr.h>` system include file contains the necessary macro definitions to find out all sorts of useful information about an image.

(2) The task will need the name of the desired image, so we declare a character array (or *string*) to contain the name. You may never need to know it, but IRAF character strings are represented by *null* delimited character arrays. An individual IRAF character is typically two machine bytes in length for portability reasons.

The length of the character string is specified using the predefined SPP macro `SZ_FNAME`. The value for `SZ_FNAME` (nominally 63 characters) is set in the include file `<iraf.h>` and is chosen to represent the maximum length of a filename. The file `<iraf.h>` (really *hlib$iraf.h*) is automatically included in all SPP programs.

(3) After the query parameters are obtained, the task should get the rest of the parameters. This should occur at the very beginning of the task to establish the context for the particular execution of the task as early as possible. In most cases, only the topmost procedure of a task should access the task parameters.

(4) In IRAF terms an image file is not *opened* for I/O access, it is *mapped*. The value returned by the `immap` function is a pointer to a data structure (the *image descriptor*) that contains the information that is needed to read or write both the image header and the image pixels.

(5) The `IM_LEN` macro is used to access the image descriptor and obtain the length of the two axes of the image. As with other macro languages, it is a good idea to omit any whitespace from the macro expression.

Note that the task assumes that the input image is two dimensional.

(6) These two statements are responsible for obtaining pointers to the image input and output buffers. The input buffer will be initialized to the specific line of the image referenced in the call to the function `imgl2r`. The output buffer, once obtained, may be written to at will. The pixels will be actually written to the file either the next time `impl2r` is referenced or when the image descriptor is *unmapped*.

(7) We have now reached the nitty-gritty of the task. The procedure `arepr` will actually copy the pixels from the input buffer to the output buffer with the specified replacements. The mechanism that SPP uses to dereference pointers utilizes a special predefined set of arrays `Memr[]`, `Memi[]`, `Memc[]`, and so on for pointers to *real*, *integer*, or *character* buffers, respectively. A good way to handle such buffers so as to avoid pointer arithmetic is to immediately dereference them and pass them to a called procedure for further processing. This allows the buffers to be accessed as normal arrays within the called procedure.

Recall that SPP allows automatic continuation to the succeeding line when a statement ends in a comma or an operator.

SPP procedures *must* be referenced with a `call` statement. SPP functions *must* be referenced within an expression, for instance, in an assignment statement.

(8) When you are finished with the image descriptor, the memory should be returned to the system for others to use. The Command Language keeps a cache of sleeping processes around for future use. Unlike a host level program which will terminate after execution, an IRAF process may be invoked and reinvoked many times over many hours. **You *must* clean up memory at the end of each task, and often at the end of each procedure.**

(9) The `arepr` procedure† is declared. The arguments of SPP procedures are passed by reference as in Fortran. The procedure receives the *addresses* of the actual arguments, not the *values* as in C. This has the usual repercussions that are encountered in Fortran programs. It is important to keep in mind which of the arguments are purely for input, which are purely for output, and which are used for both (referred to as *update* arguments).

When the procedure or function is referenced within a calling procedure, the programmer must guarantee that no output argument (*i.e.*, an argument whose value will be modified when the procedure is done) is specified as a constant. This also applies to *update* arguments, whose initial values are also critical to the routine.

Within the procedure, the programmer should be careful to never assign a new value to a purely input argument. Ideally, an output argument should be referenced only once in some expression or assignment near the bottom of the procedure.

(10) The arguments to the procedure are declared. The size of an array may be passed as an argument as in Fortran. A handy convention is to supply a descriptive comment with each argument declaration of the procedure. An initial `I`, `O`, or `U` in the individual comments indicates whether each is an input, output, or update argument.

3.3. Imreplace (Second Version)

We will now revise the task to include significantly more functionality for the user. The most apparent change will be to include support for input and output image *templates*. The user will be able to specify a *list* of images to be operated on and, at the same time, the input images will not be overwritten on output. A second change will allow the user to specify a single lower or upper bound, above or below which all pixels will be replaced. A third change allows the user to replace the pixels *outside* the specified range. This will occur when the lower bound that is specified is larger than the upper bound.

Note that there are three changes to the parameter file. First, an output image parameter was added, and the input image parameter was renamed from `image` to `input` to reflect the new ability to specify image templates and lists. Second, the `lower` and `upper` parameters have been provided with default `INDEF` values which indicate that the corresponding bound is, in effect, the minimum or maximum value in the image. Lastly, the mode of the `value` parameter has been changed from “a” (for automatic) to “h” (for hidden), and the parameter now has a default value of `0` (zero). This is purely a judgement call on the part of the programmer. The other changes emphasized the capabilities of specifying the replacement limits, and so the specification of the replacement *value* has been correspondingly de-emphasized.

examples$src/imreplace2.par:

```
# Parameters for the imreplace task.  [SECOND VERSION]
input,s,a,,,,List of input images
output,s,a,,,,List of output images
value,r,h,0.,,,Constant for the replacement operation
lower,r,h,INDEF,,,Lower limit of the replacement window
upper,r,h,INDEF,,,Upper limit of the replacement window
```

†The `arepr` procedure has been named by the conventions of the IRAF Vector Operators (or VOPS) library. The `arepr` procedure is similar to a VOPS routine since it involves a well defined operation on a vector and uses only pure arithmetic. The initial “a” indicates that this procedure operates on an array (or vector). The terminal “r” indicates that this is a version that requires type `real` input and delivers `real` output. The total length of such names is limited to six characters, but five is preferable.

examples$src/imreplace2.x:

```
# IMREPLACE [SECOND VERSION] -- replace pixel values within thresholds.

task imreplace2 = t_imreplace2

include <imhdr.h>
include <error.h>                                                    (1)

procedure t_imreplace2 ()

pointer sp, inlist, outlist, input, output, im1, im2
real    value, lower, upper, rtemp
bool    outside

pointer imtopenp(), immap()
int     imtgetim(), imtlen()
real    clgetr()

errchk  immap                                                        (1)

begin
        call smark (sp)                                              (2)
        call salloc (input, SZ_FNAME, TY_CHAR)                       (2)
        call salloc (output, SZ_FNAME, TY_CHAR)

        inlist = imtopenp ("input")                                  (3)
        outlist = imtopenp ("output")

        if (imtlen (inlist) != imtlen (outlist)) {                   (3)
            call imtclose (outlist)
            call imtclose (inlist)
            call sfree (sp)
            call error (1, "input and output lists don't match")    (1)
        }

        value = clgetr ("value")
        lower = clgetr ("lower")
        upper = clgetr ("upper")

        if (IS_INDEFR(lower) || IS_INDEFR(upper))                    (4)
            outside = false
        else
            outside = (lower > upper)

        if (lower > upper) {
            rtemp = upper
            upper = lower
            lower = rtemp
        }

        while (imtgetim (inlist, Memc[input], SZ_FNAME) != EOF &&   (3)
            imtgetim (outlist, Memc[output], SZ_FNAME) != EOF) {

            iferr (im1 = immap (Memc[input], READ_ONLY, 0)) {        (1)
                call eprintf ("Problem opening input image:\n")      (1)
                call erract (EA_WARN)                                (1)
                next
            }

            iferr (im2 = immap (Memc[output], NEW_COPY, im1)) {      (5)
                call imunmap (im1)
                call eprintf ("Problem opening output image:\n")
                call erract (EA_WARN)
                next
            }

            call imrepr (im1, im2, value, lower, upper, outside)

            call imunmap (im2)                                       (5)
            call imunmap (im1)
        }

        call imtclose (outlist)                                      (3)
        call imtclose (inlist)
        call sfree (sp)                                              (2)
end
```

```
# IMREPR -- Replace (real) pixels between lower & upper by value.

procedure imrepr (im1, im2, value, lower, upper, outside)

pointer im1, im2                #I input & output image descriptors
real    value                   #I replacement value
real    lower, upper            #I range to be replaced
bool    outside                 #I Replace values *outside* the range?

long    v1[IM_MAXDIM], v2[IM_MAXDIM]                             (6)
pointer buf1, buf2
int     n

int     imgnlr(), impnlr()

begin
        call amovkl (long(1), v1, IM_MAXDIM)                     (6)
        call amovkl (long(1), v2, IM_MAXDIM)

        n = IM_LEN(im2,1)

        if (IS_INDEFR(lower) && IS_INDEFR(upper)) # replace all pixels
            while (impnlr(im2,buf2,v2) != EOF)                   (6)
                call amovkr (value, Memr[buf2], n)

        else if (IS_INDEFR(lower))      # replace all pixels below upper
            while (imgnlr(im1,buf1,v1) != EOF && impnlr(im2,buf2,v2) != EOF)
                call arler (Memr[buf1], Memr[buf2], n, upper, value)

        else if (IS_INDEFR(upper))      # replace all pixels above lower
            while (imgnlr(im1,buf1,v1) != EOF && impnlr(im2,buf2,v2) != EOF)
                call arger (Memr[buf1], Memr[buf2], n, lower, value)

        else    # replace pixels between lower & upper
            while (imgnlr(im1,buf1,v1) != EOF && impnlr(im2,buf2,v2) != EOF)
                if (outside) {
                    call arler (Memr[buf1], Memr[buf2], n, lower, value)
                    call arger (Memr[buf2], Memr[buf2], n, upper, value)
                } else
                    call arepr (Memr[buf1], Memr[buf2], n,
                        lower, upper, value)
end


# ARLER -- Copy a to b, replacing values <= floor.

procedure arler (a, b, n, floor, newval)

real    a[n]                    #I input array
real    b[n]                    #O output array
int     n                       #I size of the arrays
real    floor, newval           #I replacement limit & constant
int     i

begin
        do i = 1, n
            if (a[i] <= floor)
                b[i] = newval
            else
                b[i] = a[i]
end


# ARGER -- Copy a to b, replacing values >= ceiling.

procedure arger (a, b, n, ceiling, newval)

real    a[n]                    #I input array
real    b[n]                    #O output array
int     n                       #I size of the arrays
real    ceiling, newval         #I replacement limit & constant
int     i

begin
        do i = 1, n
            if (a[i] >= ceiling)
                b[i] = newval
            else
                b[i] = a[i]
end
```

Various comments on the new SPP/VOS features in this example follow. First, however, note that the `arepr` procedure is not shown in the example. It is identical to the version in *imreplace1.x*. As you may already have noticed, the files as listed in this guide are not absolutely identical to the files in the *examples.tar.Z* file. That compressed tar file, as described in §7, implements an example IRAF package of all the tasks in this guide. The specific change that is required to do that is to remove all of the SPP `task` statements from the individual files into a common file that is used to link the package executable. The individual `task` statements are present in the source files, but have been commented out. The only other difference between the examples in the guide and the files in the example package is that the `arepr` procedure is commented out in the file *imreplace2.x* to avoid a duplicate declaration.

(1) New to this example is the use of IRAF's error handling facilities. The basic SPP statements and procedure calls that support these facilities are: `error`, a procedure which generates an error condition and aborts the task, the `iferr` statement, which allows such an error condition to be intercepted and handled by the task itself, `erract`, which supplies an action for an outstanding error condition within an error handler, and `errchk`, which allows error conditions in called procedures to be passed back up to the parent. In addition, `eprintf` can be used to send messages to the `STDERR` output stream. The `error.h` system header definition file is included in this case to permit the use of the `EA_WARN` macro within the call to `erract`.

(2) IRAF programming in SPP/VOS relies heavily on dynamic memory allocation for the support of various large and small data structures. This example introduces the SPP/VOS memory stack facilities. Allocating character arrays and large data buffers on the stack can substantially reduce the size of IRAF executables, especially when linking to the IRAF shared library (which is the default on the Suns). A procedure that allocates dynamic memory on the stack will typically begin by marking the position of the stack pointer when the procedure is first entered using the `smark` procedure. This entry position is saved in a pointer variable almost universally called `sp`, which is used to restore the stack to its original size and positioning when the procedure is exited. This is done using the `sfree` procedure. Calls to `salloc` are interspersed within the body of the procedure to allocate different blocks of memory for different purposes. The first argument of the `salloc` procedure is set to a pointer to a newly allocated memory buffer of the indicated size and datatype. This pointer may be dereferenced (as above) by using the SPP `Mem_[]` array constructs.

(3) A very handy item in the IRAF bag of tricks is the ability to specify an entire list (or *template*) of images or files to be worked on by a task. Image templates are accessed using about a half a dozen procedures that are a part of the *Image I/O* (or IMIO) interface.

The user typically specifies an image template using a string parameter of the task. The name of the parameter is passed to the `imtopenp` function which returns an SPP pointer to a data structure (an *image template descriptor*) that contains the expanded list of images. This data structure is opaque to the programmer, which is to say that it is only accessed using the small set of routines in the image template "package". The programmer just needs to pass the structure pointer to the `imtlen` function, for instance, to determine the number of images that were specified in the template. In usual use, the image list is cycled through one image at a time using the `imtgetim` function, which fills an output string buffer with the next image in the list each time the function is referenced. When the end of the list is reached, `imtgetim` returns the value `EOF`, which is a symbolic constant available to all SPP programs whose specific (integer) value you should not need to know (it is defined in `<iraf.h>`).

After finishing with the image template descriptors, the task should close them to return the memory to the system. This is done with the `imtclose` procedure.

This version of **imreplace** uses both an *input* and an *output* image template. The implications of this for the user are discussed in the help page in Appendix III. The implications for the programmer are that both templates must typically represent the same number of images, and that the two lists are stepped through in unison.

(4) Note how the task's behavior is specified implicitly. If `lower` is greater than `upper` the task will replace the pixels *outside* the range. This is an added feature that requires no extra parameters to specify, and that should be clear to the user.

By considering the range of allowed values for a parameter, it is often possible to leverage increased functionality without adding parameters to specify them. Note that adding another parameter can confuse the issue more than making the ones the task already has do more. One example of making the parameters do more is using upper and lower limits, such as these, that are handled differently if the values are flopped. Another is the ability to specify an `INDEF` value (in this case the default) that indicates some special value as the minimum or maximum of the image.

Other possibilities for making the user interaction more robust are the enumeration of the allowed values of a parameter, the specification of an allowed range for a numerical parameter, the idea of a multivalued prompt (`yes|no|YES|NO`) that is used in several of the IRAF spectral reduction tasks, and using negative values for a nominally positive quantity to indicate, for example, medianing instead of averaging.

(5) This call to `immap` is creating a new copy of an existing image. The data structures of the new image *depend* on the corresponding data structures of the original image. There is no way to tell exactly *when* the original image will be accessed by the IRAF VOS. You could dig through the IRAF source code, but there is no guarantee that the IRAF code will not change. Since the new image depends on the data structures of the original, the original image should remain open the entire time that the new image is open. In particular, note that the new image is closed (using `imunmap`) *before* the original image.

(6) In the previous **imreplace** example, we used the `imgl2r` and `impl2r` procedures to retrieve input and output data buffers from a two dimensional image. The first version of the task should also handle one dimensional images correctly. However, due to the design of `imgl2r` and `impl2r`, it will only process the first band of a three (or higher) dimensional image.

To fully access all seven potential dimensions of an IRAF image a different set of IMIO routines must be used. In the second version of **imreplace**, we have chosen to use the `imgnlr` and `impnlr` routines. These two routines scan sequentially through the image, rather than relying on the indexed scheme used by routines such as `imgl2r` and `impl2r`. A seven dimensional *start vector* (of datatype *long integer*), typically named `v[]`, is used to index into the image. After each call to `imgnlr` or `impnlr`, the start vector is incremented to the next line of the image. The lines of the image are referenced in storage order with the leftmost subscript varying most quickly. The start vector can also be adjusted manually to access a specific line from the image (although it will still be incremented).

In all of these examples a different datatype, as expressed by the final letter of the procedure name (in this case, "`r`" for datatype *real*), could have been chosen, although the IMIO interface allows any datatype image to be accessed by any datatype procedure. The normal datatype conversions are performed on input and output from the image to the buffer (with the usual concern for truncating precision).

4. An Advanced Example

It is now time to mention several of the most useful facilities that the SPP/VOS programming environment has to offer. The following example, the **impix** task, packages a number of these up into one task, a simplified variation of the **imedit** task from the **tv** package. **Impix** implements a form of user interface known as a *cursor loop*. After some initial configuration (in this case, the image that is specified by the `image` parameter is displayed on the workstation or other image display), the task enters an endless loop where it waits to receive cursor input from the display. The task takes different actions depending on the specific keystrokes typed.

To understand an IRAF task, the best place to start is usually the data structures. As in the previous examples, a fundamental data structure for the **impix** task is its parameter file, which is the data structure that the user sees. **Impix** also has been designed to make use of the SPP macro based data structure mechanism. While the implementation of data structures in SPP is somewhat crude, the mechanism supports a large degree of functionality. Fortran programmers will perhaps be surprised to find that an old dialect, Fortran 66 (which is what current implementations of SPP are preprocessed into), of their language of choice can support complicated, “modern”, language constructs. On the other hand, C programmers (after having a few good chuckles) should take another look at the underlying functionality that is provided. Much of what C provides is also possible in SPP.

We will begin this example, therefore, with a discussion of features of the parameter file and the task data structures which are located in the two files *examples$src/impix.par* and *examples$src/impix.h*, respectively. Note that the contents of the header definition file (*impix.h*) would normally be prepended to the source file (*impix.x*) for such a small task, but since the header file mechanism is used throughout IRAF, one is included for its pedagogical value.

examples$src/impix.par:

```
image,s,a,,,,Image to be edited
peakup,b,h,yes,,,Peak up within the box?
localmin,b,h,no,,,"Peak on the local minimum, rather than maximum?"  (1)
replace,s,h,"median","constant|mean|median",,Replacement algorithm   (2)
constant,r,h,0.,,,Value for constant replacement
boxsize,i,h,5,1,,Size of the peaking & statistics box
imcur,*imcur,h,"",,,Image display cursor input
frame,i,h,1,1,4,Frame number for the image display
update,b,h,yes,,,Actually edit the image?
```

Comments on the parameter file:

(1) **Quote a parameter prompt if it contains commas or other special characters.**

(2) The minimum field of a parameter specification is also used to enumerate allowed values for string parameters. This is a way of constraining the input to a task before the task is ever executed, *i.e.*, the CL catches values that aren’t allowed for a particular parameter.

The CL parameter enumeration operates *before* the task is executed. It is even more important to limit the inputs to the task once it is started. The `clgwrd` or `strdic` functions (indicated below, in the source code) will obtain an index string from the CL or from a character string variable, respectively, and will look it up in a dictionary string. Only values from the dictionary, or their unique abbreviations, are allowed. The routine returns the numeric index of the item in the dictionary as well as the non-abbreviated string entry. It is usually the numeric index that is actually used by the program to control a `switch` or an `if` statement.

examples$src/impix.h:

```
define  KEYHELP         "examples$lib/scr/impix.key"                    (3)
define  DISPCMD         "display %s %d >& dev$null"                     (4)

define  SZ_NAME         16                      # allow some elbow room

# enumerate the colon commands
define  COMMANDS        "|eparam|peakup|localmin|replace|constant|boxsize"
define  EPARAM          1
define  PEAKUP          2
define  LOCALMIN        3
define  REPLACE         4
define  CONSTANT        5
define  BOXSIZE         6

# enumerate the minimum match choices for the replacement algorithm
define  RALGORITHMS     "|constant|mean|median"                         (2)
define  RCONSTANT       1
define  RMEAN           2
define  RMEDIAN         3

# data structure representing the current parameters

define  LEN_IP          (8 + SZ_NAME*SZ_CHAR/SZ_STRUCT + 1)             (5)

define  IP_IM           Memi[$1]
define  IP_PEAKUP       Memi[$1+1]                                      (6)
define  IP_LOCALMIN     Memi[$1+2]                                      (6)
define  IP_RALG         Memi[$1+3]
define  IP_CONSTANT     Memr[$1+4]
define  IP_BOXSIZE      Memi[$1+5]
define  IP_HALFBOX      Memi[$1+6]
define  IP_FRAME        Memi[$1+7]
define  IP_REPLACE      Memc[P2C($1+8)]                                 (7)
```

Various comments on the data structures:

(3) Ideally, every cursor loop should have an associated keystroke help file that can be displayed while the task is executing. A user does not have access to the help page for the task while the task executes, so without the keystroke help the user would be left to either guess the commands or to exit the task just to remember a particular key. The `pagefiles` routine (below) is very helpful for permitting the user to access this help information while the task executes. It can also be used to page through a list of the output generated by the task itself. A variant on the routine is `gpagefile`, which does about the same thing, but from within a graphics oriented task.

(4) A feature of **impix** is that it allows the specified image to be displayed from within the task. To implement this, it uses the `clcmdw` routine to actually call the **display** task. In general, `clcmdw` allows one IRAF task to be called from inside another. This is very useful but requires great care in its use. In particular, a task that uses this capability may well have to be updated with each IRAF release since the behavior of the external task, especially its parameters, may have been modified. Note that the form of the **display** command line that is compiled into **impix** is as brief as possible and assumes as little as possible about the external task (**display**). An alternative would be to supply this command line (along with formatting codes) as the default for an **impix** parameter. This allows the user to supply any possible command to display the image. Unfortunately, it also allows the user to supply many impossible commands, with unpredictable results.

Since only a bare bones **display** command is specified within the task, we will allow the `eparam` parameter editor to be called from within **impix** to allow the other **display** parameters to be selected.

The “`w`” in `clcmdw` indicates that the calling task will wait until the external task is finished before continuing.

Note that we are going to be reading the image cursor to obtain the pixel coordinates to operate on in a particular image. The task should first make sure that this image is actually the one being displayed, since the image cursor will happily allow you to point at one image while you modify another.

(5) The task will allocate a block of memory of length `LEN_IP` to contain the data structure. Simple arithmetic is permitted within SPP macros, in this case to calculate the required size of the block using the sizes of the individual data elements. Note that data structures are allocated in terms of `SZ_STRUCT`, which is identical (on current machines) to the size of an integer, `SZ_INT`, *i.e.*, four 8-bit bytes. As mentioned previously, the size of an SPP character, `SZ_CHAR`, is two bytes on current machines.

(6) It is potentially non-portable to store a boolean variable in a data structure. Different machines and different languages support different concepts of what a boolean is. Since the SPP implementation of data structures doesn’t hide the specification of the individual data elements from the programmer — in particular, the sizes of these elements — the portable way to store a logical variable in a data structure is to encode it as an integer.

(7) The `P2C` macro converts a pointer to structure element into a pointer to a character. This is necessary to allow a character string to be placed directly into a allocated data structure. An alternative strategy is to merely keep track of the pointers to strings that are maintained outside of the data structure. This, however, would require that the memory for the strings be allocated and deallocated explicitly outside the data structure. Note that there are corresponding macros for other datatype pointer conversions. These macros are defined in *hlib$iraf.h* and are thus available in all SPP programs.

examples$src/impix.x:

```
# IMPIX -- point at and edit pixels in an image.

task impix = t_impix

include <error.h>
include <fset.h>
include <imhdr.h>
include <chars.h>
include "impix.h"                                                (8)

procedure t_impix ()

pointer sp, image, cmd, im, ip
int     wcs, key, cx, cy
real    new, wx, wy, value
bool    redisplay

pointer immap(), ip_init()
int     clgcur(), ip_colon(), ip_peak()
real    ip_stats()
bool    clgetb()
pointer imps2r()

errchk  immap

begin
        call fseti (STDERR, F_FLUSHNL, YES)                      (9)

        call smark (sp)
        call salloc (image, SZ_FNAME, TY_CHAR)
        call salloc (cmd, SZ_LINE, TY_CHAR)

        call clgstr ("image", Memc[image], SZ_FNAME)

        iferr {
            if (clgetb ("update"))
                im = immap (Memc[image], READ_WRITE, 0)
            else
                im = immap (Memc[image], READ_ONLY, 0)           (10)
        } then {
            call sfree (sp)
            call erract (EA_ERROR)
        }

        if (IM_NDIM(im) != 2) {                                  (11)
            call imunmap (im)
            call sfree (sp)
            call error ("image (or image section) is not two dimensional")
        }

        ip = ip_init ()                                          (12)

        call ip_display (Memc[image], IP_FRAME(ip))              (4)
        redisplay = false

        # the heart of the task, the actual cursor loop          (13)

        while (clgcur ("imcur", wx, wy, wcs, key, Memc[cmd], SZ_LINE) != EOF) {
            switch (key) {

            # quit the loop (same as EOF)
            case 'q':
                break

            # redisplay the image
            case 'r':
                redisplay = true                                 (14)

            # report the statistics
            case 's':
                if (ip_peak (im, ip, wx, wy, cx, cy, value) == ERR)
                    next

                new = ip_stats (im, ip, cx, cy, value, YES)

            # zap (replace) the pixel
            case 'z':
                if (ip_peak (im, ip, wx, wy, cx, cy, value) == ERR)
                    next

                new = ip_stats (im, ip, cx, cy, value, NO)

                if (! IS_INDEFR(new)) {
                    Memr[imps2r(im, cx, cx, cy, cy)] = new
                    call imflush (im)                            (15)
                }

            # page keystroke help file
            case '?':
                call pagefiles (KEYHELP)                         (3)

            # colon commands are used to examine or change parameters
            case ':':
                redisplay = ip_colon (Memc[cmd], ip)

            # direct the user to the '?' help                    (16)
            default:
                call eprintf ("unknown command, type '?' for help\n")

            }

            if (redisplay) {
                call ip_display (Memc[image], IP_FRAME(ip))
                redisplay = false
            }
        }

        call mfree (ip, TY_STRUCT)                               (17)
        call imunmap (im)
        call sfree (sp)
end


# IP_INIT -- read the parameters and initialize the task structure.

pointer procedure ip_init ()

int     boxsize
pointer ip

int     clgeti(), clgwrd(), btoi()
real    clgetr()
bool    clgetb()

begin
        call malloc (ip, LEN_IP, TY_STRUCT)                      (5,17)

        IP_PEAKUP(ip) = btoi (clgetb ("peakup"))
        IP_LOCALMIN(ip) = btoi (clgetb ("localmin"))

        IP_RALG(ip) = clgwrd ("replace",
            IP_REPLACE(ip), SZ_NAME, RALGORITHMS)                (2)
        IP_CONSTANT(ip) = clgetr ("constant")

        boxsize = clgeti ("boxsize")
        if (mod (boxsize, 2) == 0) {
            boxsize = boxsize + 1
            call eprintf ("boxsize must be odd, using %d\n")
                call pargi (boxsize)
        }
        IP_BOXSIZE(ip) = boxsize
        IP_HALFBOX(ip) = (boxsize - 1) / 2

        IP_FRAME(ip) = clgeti ("frame")

        return (ip)
end


# IP_COLON -- handle the colon commands.

bool procedure ip_colon (cmd, ip)

char    cmd[ARB]                #I command line (excluding ':')  (18)
pointer ip                      #U task structure pointer

int     itemp
real    rtemp
```

```
char    stemp[SZ_NAME], command[SZ_NAME], arg[SZ_NAME]
bool    btemp, redisplay, changeit

int     strdic(), nscan()

begin
        # initialize the returned value and to handle a failed sscan
        redisplay = false
        command[1] = NULL

        call sscan (cmd)
            call gargwrd (command, SZ_NAME)
            call gargwrd (arg, SZ_NAME)   # limited to SZ_NAME even for numbers

        changeit = (nscan() == 2)

        switch (strdic (command, command, SZ_NAME, COMMANDS)) {
        case EPARAM:
                call clcmdw ("eparam display")                     (4)
                redisplay = true

        case PEAKUP:
            if (changeit) {
                call sscan (arg)
                    call gargb (btemp)

                if (nscan() != 1) {
                    call eprintf ("error in peakup: ‘%s’\n")
                        call pargstr (arg)
                } else
                    IP_PEAKUP(ip) = btemp
            }

            call printf ("peakup = %b\n")
                call pargb (IP_PEAKUP(ip))

        case LOCALMIN:
            if (changeit) {
                call sscan (arg)
                    call gargb (btemp)

                if (nscan() != 1) {
                    call eprintf ("error in localmin ‘%s’\n")
                        call pargstr (arg)
                } else
                    IP_LOCALMIN(ip) = btemp
            }

            call printf ("localmin = %b\n")
                call pargb (IP_LOCALMIN(ip))

        case REPLACE:
            if (changeit) {
                itemp = strdic (arg, stemp, SZ_NAME, RALGORITHMS)  (2)

                if (itemp <= 0) {
                    call eprintf ("error in replace string ‘%s’\n")
                        call pargstr (arg)
                } else {
                    IP_RALG(ip) = itemp
                    call strcpy (stemp, IP_REPLACE(ip), SZ_NAME)
                }
            }

            call printf ("replace = %s\n")
                call pargstr (IP_REPLACE(ip))

        case CONSTANT:
            if (changeit) {
                call sscan (arg)
                    call gargr (rtemp)

                if (nscan() != 1) {
                    call eprintf ("error in constant ‘%s’\n")
                        call pargstr (arg)
                } else
                    IP_CONSTANT(ip) = rtemp
            }

            call printf ("constant = %g\n")
```



```
                call pargr (IP_CONSTANT(ip))

        case BOXSIZE:
            if (changeit) {
                call sscan (arg)
                    call gargi (itemp)

                if (nscan() != 1) {
                    call eprintf ("error in boxsize ‘%s’\n")
                        call pargstr (arg)
                } else {
                    # round up to odd
                    itemp = abs (itemp)
                    IP_BOXSIZE(ip) = itemp + 1 - mod (itemp, 2)
                    IP_HALFBOX(ip) = (IP_BOXSIZE(ip) - 1) / 2
                }
            }

            call printf ("boxsize = %d\n")
                call pargi (IP_BOXSIZE(ip))

        default:                                                          (16)
            call printf ("unknown colon command ‘%s’, type ‘?’ for help\n")
                call pargstr (cmd)

        }

        return (redisplay)
end


# IP_DISPLAY -- construct a command line and display an image.

procedure ip_display (image, frame)

char    image[ARB]              #I image name to display
int     frame                   #I frame number to display it in

pointer sp, cmdline

begin
        call smark (sp)
        call salloc (cmdline, SZ_LINE, TY_CHAR)

        call sprintf (Memc[cmdline], SZ_LINE, DISPCMD)
            call pargstr (image)
            call pargi (frame)

        call clcmdw (Memc[cmdline])                                       (4)

        call sfree (sp)
end


# IP_PEAK -- peak up using the specified parameters.

int procedure ip_peak (im, ip, wx, wy, cx, cy, value)

pointer im                      #I image desciptor
pointer ip                      #I task structure pointer
real    wx, wy                  #I cursor (window) coordinates
int     cx, cy                  #O peaked coordinates
real    value

int     xp, yp, x1, x2, y1, y2, nx, ny

pointer imgs2r()

begin
        cx = nint (wx)
        cy = nint (wy)

        if (cx < 1 || cx > IM_LEN(im,1) || cy < 1 || cy > IM_LEN(im,2)) {
            call eprintf ("cursor is outside the image (%d,%d)\n")
                call pargi (cx)
                call pargi (cy)
            return (ERR)
        }

        if (IP_PEAKUP(ip) == NO)
```

```
            return (OK)

        x1 = max (cx - IP_HALFBOX(ip), 1)
        x2 = min (cx + IP_HALFBOX(ip), IM_LEN(im,1))
        y1 = max (cy - IP_HALFBOX(ip), 1)
        y2 = min (cy + IP_HALFBOX(ip), IM_LEN(im,2))

        nx = x2 - x1 + 1
        ny = y2 - y1 + 1

        call ip_ext (Memr[imgs2r(im,x1,x2,y1,y2)], nx, ny,
            xp, yp, value, IP_LOCALMIN(ip))

        cx = x1 + xp - 1
        cy = y1 + yp - 1

        return (OK)
end


# IP_EXT -- find the coordinates of the extreme (maximum or minimum) pixel.
# If there are multiple pixels having the extreme value, the first in
# storage order is reported with no warning.

procedure ip_ext (a, nx, ny, xp, yp, extreme, minimum)

real    a[nx,ny]                #I array of pixel values
int     nx, ny                  #I dimensions of the array
int     xp, yp                  #O coordinates of the extreme pixel
real    extreme                 #O the value of the extreme pixel
int     minimum                 #I search for minimum, rather than maximum

int     x, y

begin
        xp = 1
        yp = 1
        extreme = a[1,1]

        if (minimum == YES) {
            do y = 1, ny
                do x = 1, nx
                    if (a[x,y] < extreme) {
                        xp = x
                        yp = y
                        extreme = a[x,y]
                    }

        } else {
            do y = 1, ny
                do x = 1, nx
                    if (a[x,y] > extreme) {
                        xp = x
                        yp = y
                        extreme = a[x,y]
                    }

        }
end


# IP_STATS -- calculate (and optionally print) the statistics
# for the indicated subraster (box) of an image.

real procedure ip_stats (im, ip, x, y, current, verbose)

pointer im                      #I image descriptor
pointer ip                      #I task structure pointer
int     x, y                    #I coords of the box’s central pixel
real    current                 #I current value of this pixel
int     verbose                 #I Print the statistics on the STDOUT?

int     x1, x2, y1, y2, nx, ny
real    mean, median, sigma
pointer buf

pointer imgs2r()
real    amedr()

begin
```



```
        if (IP_RALG(ip) == RCONSTANT && verbose == NO)
            return (IP_CONSTANT(ip))

        x1 = max (x - IP_HALFBOX(ip), 1)
        x2 = min (x + IP_HALFBOX(ip), IM_LEN(im,1))
        y1 = max (y - IP_HALFBOX(ip), 1)
        y2 = min (y + IP_HALFBOX(ip), IM_LEN(im,2))

        nx = x2 - x1 + 1
        ny = y2 - y1 + 1

        buf = imgs2r (im, x1, x2, y1, y2)

        call aavgr (Memr[buf], nx*ny, mean, sigma)
        median = amedr (Memr[buf], nx*ny)

        if (verbose == YES) {
            call printf ("\n%s[%d:%d,%d:%d]:\n")
                call pargstr (IM_HDRFILE(im))
                call pargi (x1)
                call pargi (x2)
                call pargi (y1)
                call pargi (y2)

            call printf ("    mean = %8.6g, sigma = %8.6g")
                call pargr (mean)
                call pargr (sigma)

            call printf ("  median = %8.6g\n")
                call pargr (median)

            call printf (" current = %8.6g at (%d,%d)\n")
                call pargr (current)
                call pargi (x)
                call pargi (y)

            if (IP_RALG(ip) == RCONSTANT) {
                call printf ("constant = %8.6g (for pixel editing)\n")
                    call pargr (IP_CONSTANT(ip))
            }

            call flush (STDOUT)
        }

        switch (IP_RALG(ip)) {
        case RCONSTANT:
            return (IP_CONSTANT(ip))
        case RMEAN:
            return (mean)
        case RMEDIAN:
            return (median)
        }
end
```

More comments:

(8) Since the header file *impix.h* resides in the same directory as the source files, specify the quoted filename directly. This is in contrast to surrounding the filename with angle brackets (<>), which indicates to the compiler that the file should be searched for in the system include directories, *lib$* and *hlib$*.

(9) In previous examples we would explicitly flush the `STDOUT` or `STDERR` buffers when desired, sometimes after each printing statement, sometimes only implicitly when the task exits. The `fseti` call can be used to fiddle with the internal FIO parameters. In most cases this is unnecessary and downright undesirable, but the `F_FLUSHNL` option to automatically flush the output buffers when a newline is encountered is often useful. Note that it is not always a good idea to do this since the main idea of the buffering mechanism is to promote efficiency which is subverted when the buffer is never allowed to fill. In particular, a task that generates a long listing of output rather than frequent interactive updates should not use this option.

(10) Setting the access mode to `READ_ONLY` allows you to examine an image for which you don't have write permission.

(11) The task should explicitly check its input. If the logic of the task is limited to particular image size or dimensionality for some reason, this should be checked after the image is opened. It is the two dimensional nature of the display device that mandates the restriction in this case. The task could, for instance, have also been coded to step serially through multi-dimensional bands of a higher order image. Note that image sections are handled by the underlying IMIO interface. A user can specify a two dimensional section of a three (or higher) dimensional image and the task will be perfectly happy.

(12) Usually a task will only access its parameters within the main procedure. In a task that has a significant data structure, it is often proper to initialize this data structure from the task parameters within a subroutine that encapsulates the messy details.

(13) The main part of the task is a loop that keys off of the values returned from the image cursor. The `clgcur` routine retrieves the value of the specified parameter and assigns the result to the five subsequent fields, which are the X and Y coordinates of the cursor in the *world coordinate system* specified by `wcs`, the single `key` that was typed as well as any subsequent command string that the user may have happened to type following a colon (`:`). These are *colon commands*. The routine returns the number of fields that were actually read or `EOF` if there is no more cursor input.

(14) In many cases, several of the cursor commands will include overlapping sets of actions. It is often handy to merely set flags in the switch statement that distinguishes among the various cursor options, and then use the flags to finish the actions up at the end.

(15) This particular task randomly alternates pixel reading with pixel writing at the whim of the user. To keep the context of the image I/O valid, the task must explicitly flush the output buffer each time the image is written to.

This is unnecessary in most tasks since an image will typically be either read from or written to, but not both. Each time a new output buffer is requested using a routine such as `imps2r`, the previous contents of the output buffer are implicitly flushed. If the task doesn't alternate pixel reads with writes, the implicit flushing will never lose synchronization.

(16) **Always catch the case of an unknown command and tell the user how to get help**. This permits a user to jump into the task with almost no preparation. A "`?`" is the standard IRAF command for listing the available commands. Catching erroneous colon commands is especially important since many of them provide no other user feedback. If the default branch of the switch is not provided, a user might happily proceed under mistaken assumptions after typing an incorrect command. This might ruin his or her whole day.

(17) The `malloc` procedure is used to allocate a block of memory (from the *heap*) that will endure once the calling procedure exits. Contrast this to the `salloc` procedure whose allocated memory (from the *stack*) should be freed before the procedure exits. **At some point elsewhere in the program, the allocated block of memory must be freed by using the `mfree` procedure.**

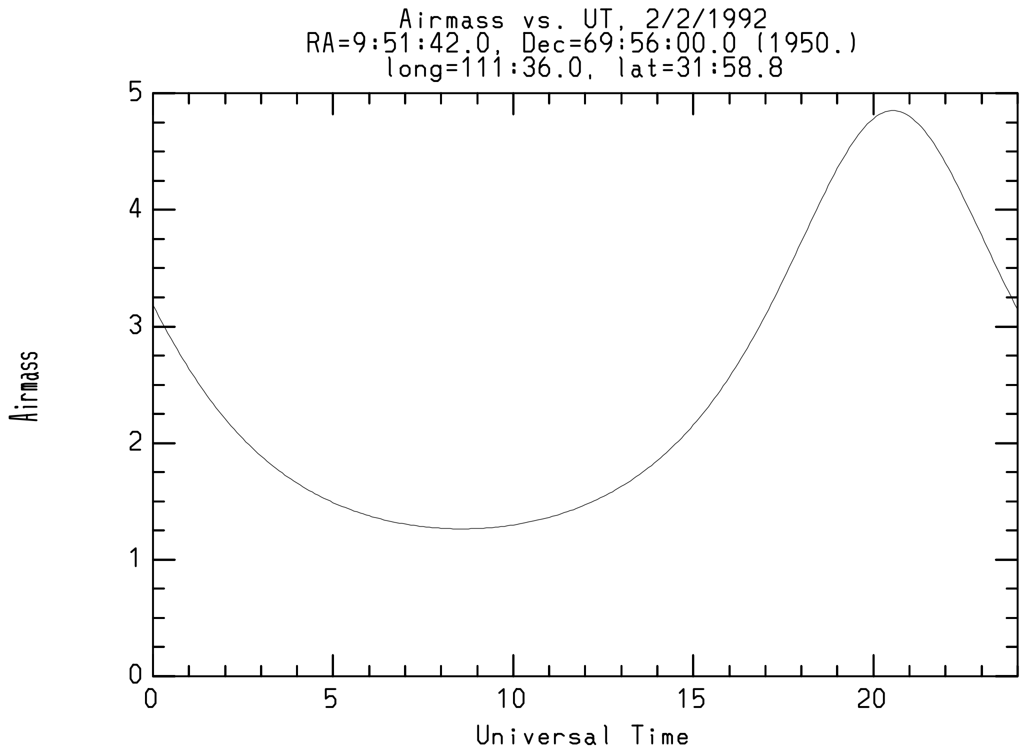
(18) The length of an array that is an argument of a procedure may be specified with the `ARB` keyword. This is a mere placeholder for the actual length of the array which is not known until the procedure is called. However, if the array is to be written into, or if the length of the array for reading is not indicated by a mechanism similar to the `NULL` termination of a character string, then the length of the array must still be passed into the procedure to constrain the I/O operations.

5. Modifying an Existing Task

There are an awful lot of tasks in IRAF and in associated externally developed packages such as STSDAS and PROS. It is quite likely that someone has already written a task similar to one that you may be considering writing. With all of this example code, users should be able to tweak tasks that are beyond their current programming plateau in order to achieve rather elegant results.

The example in this section borrows from the **airmass** task in the **astutil** package of IRAF and from the **prows** task in the **plot** package. The concept is to combine the two to produce a task that allows plotting the airmass of some astronomical object as a function of Universal Time for a given date at a given observatory.

To whet your appetite, here is a plot that was produced by the final version of the task. This is a plot of the airmass for M82 from Kitt Peak for Groundhog's Day in 1992. Note that the time zone of Arizona is Mountain Standard Time all year round, and that local midnight is therefore at 7 hours UT ($MST = UT - 7$). The airmass is constrained to small values for this plot since M82 is a circumpolar object. A non-circumpolar object will have a plot which goes off scale when the object sets.



While modifying **airmass**, we will discover that virtually everything has to be discarded from the task. However, a little extra rummaging about in **astutil** will reveal a procedure (also called `airmass`) in the **setairmass** task and a library of handy astronomical utilities in the *asttools* subdirectory that will serve our needs quite nicely.

General advice before we get started: **Don't modify tasks in the IRAF directory tree!** You should always copy files into your own area to work on them. There is also no reason to be logged into the IRAF account since all IRAF source files should be readable to the world. **Look before you leap!** You should exercise caution before diving into some code modification. Unless you carefully plan a line of attack at the beginning, an extensive modification can consume more time than starting from scratch.

(As you read through the remainder of §5, you may find it useful to refer from time to time to the summary on page 40.)

How to begin? Well, first just by having some idea of what it is one wants to do, and some idea of the tasks currently offered by IRAF. The **astutil** package seems like a likely place to look for algorithms to calculate the airmass as a function of the position of an object in the sky (*secant*(*Z*) or a variant), and the position of an object as a function of time (LST or UT). To start poking around:

```
cl> noao
      artdata       digiphot      mtlocal       proto         twodspec
      astrometry    focas         observatory   rv
      astutil       imred         onedspec      surfphot
no> astutil
      airmass     ccdtime     gratings    precess     setairmass
      asttimes    galactic    pdm         rvcorrect
as> help
        airmass - Compute the airmass at a given elevation above the horizon
       asttimes - Compute UT, Julian day, epoch, and siderial time
        ccdtime - Compute time required to observe star of given magnitude
       galactic - Convert ra, dec to galactic coordinates
       gratings - Compute and print grating parameters
            pdm - Find periods in light curves by Phase Dispersion Minimization
        precess - Precess a list of astronomical coordinates
      rvcorrect - Compute radial velocity corrections
     setairmass - Compute effective airmass and middle UT for an exposure
```

Two tasks, **airmass** and **setairmass**, seem promising. To look at the SPP source code†, you can either use the IRAF **help** task:

```
as> help airmass option=source
```

or you can examine the file directly:

```
as> cd astutil
as> dir
README              astutil.men         mkpkg               setairmass.par
Revisions           astutil.par         observatory.par     t_asttimes.x
airmass.par         ccdtime.par         pdm                 t_gratings.x
airmass.x           ccdtime.x           pdm.par             t_obs.x
asttimes.par        doc                 precess.par         t_rvcorrect.x
asttools            galactic.par        precess.x           t_setairmass.x
astutil.cl          galactic.x          rvcorrect.com       x_astutil.x
astutil.hd          gratings.par        rvcorrect.par
```

Which file contains the SPP program for the **airmass** task? At this point all we can do is make the logical guess that it is *airmass.x*. After examining the file, we will return to this question.

```
as> page airmass.x
```

Whether the **help** task is used to view the program, or whether the file is viewed directly does not matter in this case. In other cases of more complicated programs, the **help** task will only show the topmost procedure of the task. The other parts of such tasks, in other files, will have to be viewed individually.

†The example in this section relies on an *unstripped* IRAF system. Stripping (deleting) the source code and other non-runtime files is an option when an IRAF installation is configured. This should normally only be done if disk space on the computer is very limited. Users with stripped systems can still retrieve the final version of the **pairmass** task from the **examples** package (see §7).

astutil$airmass.x:

```
# AIRMASS -- Compute the airmass at a given elevation above the horizon.
# Airmass formulation from Allen "Astrophysical Quantities" 1973 p.125,133.

procedure t_airmass()

real    elevation, airmass, scale
real    x, radians_per_degree
bool    clgetb()
real    clgetr()
data    radians_per_degree /57.29577951D0/

begin
        # Get elevation in either degrees or radians and the scale factor
        # for the Earth's atmosphere.
        elevation = clgetr ("elevation")
        if (!clgetb ("radians"))
            elevation = elevation / radians_per_degree
        scale = clgetr ("scale")

        x  = scale * sin (elevation)
        airmass = sqrt (x**2 + 2*scale + 1) - x

        call printf ("airmass %.5g at an elevation of ")
            call pargr (airmass)
        call printf ("%.5g degrees (%.5g radians) above horizon\n")
            call pargr (elevation * radians_per_degree)
            call pargr (elevation)

        # Store airmass back in a parameter so that it can be accessed from
        # the CL.
        call clputr ("airmass", airmass)
end
```

To return to the question: how did we know that the source code for the task, **airmass**, was contained in the file *airmass.x*? Answer: we didn't, we merely guessed. Guessing the filename works quite well in practice, since files tend to be reasonably named. One added hint that doesn't apply in this case, but that often helps to decide where to begin to look, is that files containing task entry points often have a prepended *t_*, for instance, *t_setairmass.x*.

There is a better way, however. Every package has a *package definition script* in which the package's tasks and the context in which those tasks execute are defined. The standard name of this script is the name of the package with a *.cl* appended, and it should be located in the root directory of the package. In this case, the file is:

astutil$astutil.cl:

```
#{ Package script task for the ASTUTIL package.

package astutil

task    airmass,
        precess,
        galactic,
        gratings,
        pdm,
        asttimes,
        rvcorrect,
        setairmass,
        ccdtime         = "astutil$x_astutil.e"

clbye()
```

Tasks of all descriptions can be defined in such a script. These include script tasks and foreign tasks as well as SPP tasks. In this case, all of the tasks in the **astutil** package are contained in the executable file, *x_astutil.e*, nominally in the *astutil* directory. (IRAF actually places these files in separate *bin* directories, as will be discussed in §7.)

At this point, the **mkpkg** mechanism that IRAF uses to control the compilation and linking of such executables becomes pertinent. **Mkpkg** is an analog of the Unix **make** facility that includes support for the descent of directory trees and the maintenance of libraries. Like **make**, **mkpkg** operates from a simple text file that describes the necessary operations that are required to build some piece of software. This file is also called *mkpkg*, and for the **astutil** package is the following:

astutil$mkpkg:

```
# Make the ASTUTIL package.

$call   relink
$exit

update:
        $call   relink
        $call   install
        ;

relink:
        $set    LIBS = "-lxtools -lcurfit -lbev"

        $update libpkg.a
        $omake  x_astutil.x
        $link   x_astutil.o libpkg.a $(LIBS) -o xx_astutil.e
        ;

install:
        $move   xx_astutil.e noaobin$x_astutil.e
        ;

libpkg.a:
        @asttools
        @pdm

        airmass.x
        ccdtime.x
        galactic.x      <fset.h>
        precess.x       <fset.h>
        t_gratings.x    <error.h> <math.h>
        t_setairmass.x  <imhdr.h> <error.h>
        t_asttimes.x    <error.h>
        t_rvcorrect.x   rvcorrect.com <error.h>
        t_obs.x
        ;
```

The reader is directed to the help page for the **mkpkg** task and also to §7 for further information. As might be expected from our previous experience (that the output of **xc** derives its name from the input), the *mkpkg* file shows that *x_astutil.x* is the source file that is compiled and linked to create the *x_astutil.e* executable. The standard behavior for a *mkpkg* file, however, is to perform the compilation and linking as separate steps and to choose the final name explicitly. The initial linking is done in the source directory with the executable receiving a temporary name that includes an extra prepended *x*, as in *xx_astutil.e*. This two step process avoids clobbering the current version of the executable until the new version compiles and links cleanly. The extra *x* guarantees that a user will not inadvertently access the wrong version of the executable.

Next in this chain of logic is the *process definition file* (called by some the *task definition file*): **astutil$x_astutil.x**:

```
# Process definition of the ASTUTIL package.

task    precess   = t_precess,
        airmass   = t_airmass,
        ccdtime   = t_ccdtime,
        galactic  = t_galactic,
        gratings  = t_gratings,
        pdm       = t_pdm,
        rvcorrect = t_rvcorrect,
        setairmass= t_setairmass,
        asttimes  = t_asttimes,
        observatory = t_observatory
```

There are three fundamental reasons that this multi-line task statement (note the commas that implicitly continue the statement onto succeeding lines) is placed in a separate file. First, an IRAF executable can only have one task statement. Second, an IRAF package is normally arranged so that all of the compiled object code is kept in a single library file, almost universally named *libpkg.a*. The single exception is the compiled process definition file, in this case named *x_astutil.o*, which supplies the “main program” to which the package library is linked. Third, tasks are often added to a package for a new version of the system, and sometimes they are deleted. On occasion, tasks even move from package to package. Having a single file in a package in which all of the package’s tasks are brought together to be linked makes it easier to perform this maintenance.

We are nearing the end. The airmass task is seen to correspond to the procedure `t_airmass`, which we have already seen above. The final step in the logic, in case the initial guesses had failed to identify the correct file is to search for that procedure in the source files of the package. For instance:

```
as> match t_airmass *.x
airmass.x:procedure t_airmass()
x_astutil.x:    airmass   = t_airmass,
```

This confirms that the correct file is *airmass.x*.

After this detour, let’s return to the business at hand and examine the file. The **airmass** task actually does not seem very useful for our purposes, since all it does is convert an elevation above the horizon into an airmass. We need a routine that will return an airmass given the instantaneous location of the object in the sky, *i.e.*, given a set of quantities such as the right ascension, declination, sidereal time, and latitude. While it doesn’t do just what we want, the **airmass** task offers a place to start at least, so we will use it as such.

The first thing to do is to create an empty subdirectory in which to put our selection of files. These files will include the **airmass** task’s parameter file, *airmass.par* as well as *airmass.x*. In addition, we will need the *mkpkg* file:

```
as> mkdir home$pairmass
as> cd home$pairmass
as> copy astutil$airmass.x t_pairmass.x
as> copy astutil$airmass.par pairmass.par
as> copy astutil$x_astutil.x x_pairmass.x
as> copy astutil$mkpkg ./
```

Note that we have taken the opportunity to rename the source and parameter files to reflect the final taskname, **pairmass**.

The next step would normally be to add the task statement directly to the file *t_pairmass.x*, however in this case we will be patching together several source files so we will keep it separately in the file *x_pairmass.x*. After fiddling with the name of the procedure and the comment, the first few lines of *t_pairmass.x* become:

```
# PAIRMASS -- plot the airmass for a given RA, Dec on a given date.

procedure t_pairmass ()
```

While the file *x_pairmass.x* becomes simply:

```
# Process definition of the PAIRMASS task.

task pairmass = t_pairmass
```

We will eventually have to modify the parameter and *mkpkg* files, but first we should dig up the rest of the pieces of SPP code that we will be including in our task. We still need the central routine of our task, which will calculate the airmass given the location of an object in the sky. Recall that there were *two* tasks in the **astutil** package that dealt with airmasses. The second task was **setairmass**, in which we find just what we need, the function `airmass`:

```
as> help setairmass option=source
                              ...
include <math.h>
                              ...
# AIRMASS -- Compute airmass from DEC, LATITUDE and HA

# Airmass formulation from Allen "Astrophysical Quantities" 1973 p.125,133.
# and John Ball's book on Algorithms for the HP-45

double procedure airmass (ha, dec, lat)

double  ha, dec, lat, cos_zd, x

define  SCALE   750.0d0                 # Atmospheric scale height

begin
        if (IS_INDEFD (ha) || IS_INDEFD (dec) || IS_INDEFD (lat))
            call error (1, "Can't determine airmass")

        cos_zd = sin(DEGTORAD(lat)) * sin(DEGTORAD(dec)) +
                 cos(DEGTORAD(lat)) * cos(DEGTORAD(dec)) * cos(DEGTORAD(ha*15.))
        x  = SCALE * cos_zd

        return (sqrt (x**2 + 2*SCALE + 1) - x)
end
                              ...
```

The next step is to retrieve this procedure into a file in our work directory. There are many ways to do this using various text editors or window systems, but let's just copy the entire file and edit out everything but the `airmass` procedure. The file should also retain the statement `include <math.h>`, since the `DEGTORAD` macro is defined in that file.

```
as> copy astutil$t_setairmass.x airmass.x
as> edit airmass.x
```

We have chosen to reassign the name *airmass.x* to the file containing the new algorithm.

At this point, we have a main task procedure which will eventually need to be entirely rewritten and the `airmass` function which will actually calculate the airmass as the object moves across the sky. We still need a routine to make the plot and a number of utility procedures to manipulate the astronomical positions and times. Before we leave the **astutil** package, it turns out that the astronomical utility procedures are all packaged up and ready to go, once we find them. In the case of a package, such as **astutil**, in which several tasks require a number of related utilities, the procedures are often placed in a subdirectory. In this case that subdirectory is *asttools*:

```
as> cd astutil
as> dir asttools
PRECESS         astgalactic.x   asttimes.x      astvrotate.x precessmgb.x
README          astgaltoeq.x    astvbary.x      astvsun.x
astcoord.x      asthjd.x        astvorbit.x     mkpkg
astdsun.x       astprecess.x    astvr.x         precessgj.x
```

This subdirectory contains a number of useful routines. Examples of how to use them are to be found in the final version of **pairmass**, which is listed below, and of course in various tasks in the **astutil** package, itself. In §7 you will see how to use them in your own programs. In any case, let's copy them for later use:

```
as> cd asttools
as> mkdir home$pairmass/asttools
as> copy * home$pairmass/asttools/
```

We have extracted everything that we need from the **astutil** package. The last piece that we need is a plot routine, so let's go rummage about in the **plot** package:

```
as> plot
      calcomp     gkiextract  implot      pradprof    sgikern     velvect
      contour     gkimosaic   nsppkern    prow        showcap
      crtpict     graph       pcol        prows       stdgraph
      gkidecode   hafton      pcols       pvector     stdplot
      gkidir      imdkern     phistogram  sgidecode   surface
pl> help
                  ...
           prow - Plot a line (row) of an image
          prows - Plot the average of a range of image lines
                  ...
```

The description of the **prow** task suggests that it is about the simplest of the general purpose plotting tasks. A look at the source code shows that it is less than a page in length.

```
pl> help prow option=source
```

Better yet, the actual plot is constructed entirely with a call to a single routine:

```
# Now draw the vector to the screen.
call pr_draw_vector (Memc[image], Memr[x_vec], Memr[y_vec], ncols,
    zmin, zmax, row, row, Memc[wcslab], false)
```

But where is pr_draw_vector? Time for more rummaging:

```
pl> cd plot
pl> match pr_draw_vector *.x
t_prow.x:       call pr_draw_vector (Memc[image], ...
t_prows.x:      call pr_draw_vector (Memc[image], ...
t_prows.x:procedure pr_draw_vector (image,
```

So pr_draw_vector is in the file *t_prows.x*. Let's grab it and remove everything but the pr_draw_vector procedure itself:

```
pl> copy t_prows.x home$pairmass/drawvector.x
pl> cd home$pairmass
pl> edit drawvector.x
```

In this case, we also need to retain the include <gset.h> and include <mach.h> statements. How do we know this? Well, at some point the source code's use of VOS routines just simply has to be digested, but there is a general scheme that will help to deduce the include file dependencies.

First, IRAF programs typically include *.h* files in order to access SPP macros (declared using a define statement). Second, the standard practice is to capitalize macros, and to only capitalize macros. Finally, the standard IRAF *.h* include files are kept in either the *lib$* or the *hlib$* directories. So when borrowing a routine that is only part of some large file, as in the airmass or pr_draw_vector procedures, scan through the code to see if there are any capitalized macros that you don't recognize as being automatically defined (such as SZ_FNAME in *hlib$iraf.h*). To see if the macro name is found in any of the standard include files:

```
pl> match "EPSILONR" lib$*.h
pl> match "EPSILONR" hlib$*.h
hlib$mach.h:define      EPSILONR        (1.192e-7)   ...
hlib$mach.h:define      EPSILON         EPSILONR
pl> match "G_XNMAJOR" lib$*.h
lib$gset.h:define       G_XNMAJOR              111
```

So the procedure requires both <*mach.h*> and <*gset.h*>. (The other "G_" macros in pr_draw_vector are also found in <*gset.h*>.) This is not a foolproof method of deduction. Packages sometimes have their own include files, so the **match** command may also need to be executed in the package directory. When including such a file, use quotes ("") instead of angle braces (<>) as in: include "package.h". In this case, you would have to also copy the include file to your work directory.

Let's take a look at our copy of pr_draw_vector. We will first make a few changes. You may want to compare the new version, listed below, with the original:

```
pl> phelp prows option=source
```

The **phelp** task was used here to allow paging backwards as well as forwards through the source file (or any other help file, for that matter). Since our new task has no connection to the **prow** or **prows** tasks, the first change was to rename the procedure to simply draw_vector. Second, some variables have been given more logical names: image has become def_title and ncols has become simple n, for instance. Third, a block of code which was used to label the plot with information specific to the original tasks has been removed except for support for a more generic default title. Finally, the arguments to the procedure have been simplified and generalized. Simplified by removing the arguments that were used to pass the labeling information, and generalized by allowing the default X axis limits of the plot to be adjusted as well as the corresponding Y axis defaults.

examples$src/pairmass/drawvector.x:

```
# DRAW_VECTOR -- Draw the projected vector to the screen.

include <gset.h>
include <mach.h>

procedure draw_vector (def_title, xvec, yvec, n, xmin, xmax, ymin, ymax)

char    def_title[ARB]                          #I default plot title
real    xvec[n], yvec[n]                        #I vectors to plot
int     n                                       #I npts in vectors
real    xmin, xmax                              #I x vector min & max
real    ymin, ymax                              #I y vector min & max

pointer sp, gp
pointer device, marker, xlabel, ylabel, title, suffix
real    wx1, wx2, wy1, wy2, vx1, vx2, vy1, vy2, szm, tol
int     mode, imark
bool    pointmode

pointer gopen()
real    clgetr()                                                (1)
bool    clgetb(), streq()
int     btoi(), clgeti()

begin
        call smark (sp)
        call salloc (device, SZ_FNAME, TY_CHAR)
        call salloc (marker, SZ_FNAME, TY_CHAR)
        call salloc (xlabel, SZ_LINE,  TY_CHAR)
        call salloc (ylabel, SZ_LINE,  TY_CHAR)
        call salloc (title,  SZ_LINE,  TY_CHAR)
        call salloc (suffix, SZ_FNAME, TY_CHAR)

        call clgstr ("device", Memc[device], SZ_FNAME)          (2)
        mode = NEW_FILE
        if (clgetb ("append"))
            mode = APPEND

        gp = gopen (Memc[device], mode, STDGRAPH)               (3)
        tol = 10. * EPSILONR

        if (mode != APPEND) {
            # Establish window.
            wx1 = clgetr ("wx1")
            wx2 = clgetr ("wx2")
            wy1 = clgetr ("wy1")
            wy2 = clgetr ("wy2")

            # Set window limits to defaults if not specified by user.
            if ((wx2 - wx1) < tol) {                            (4)
                wx1 = xmin
                wx2 = xmax
            }

            if ((wy2 - wy1) < tol) {
                wy1 = ymin
                wy2 = ymax
            }

            call gswind (gp, wx1, wx2, wy1, wy2)                (5)

            # Establish viewport.
            vx1 = clgetr ("vx1")
            vx2 = clgetr ("vx2")
            vy1 = clgetr ("vy1")
            vy2 = clgetr ("vy2")

            # Set viewport only if specified by user.
            if ((vx2 - vx1) > tol && (vy2 - vy1) > tol)
                call gsview (gp, vx1, vx2, vy1, vy2)            (6)
            else {
                if (!clgetb ("fill"))
                    call gseti (gp, G_ASPECT, 1)                (7)
            }

            call clgstr ("xlabel", Memc[xlabel], SZ_LINE)
            call clgstr ("ylabel", Memc[ylabel], SZ_LINE)
            call clgstr ("title",  Memc[title],  SZ_LINE)

            if (streq (Memc[title], "default"))                  (8)
                call strcpy (def_title, Memc[title], SZ_LINE)

            call gseti (gp, G_XNMAJOR, clgeti ("majrx"))         (9)
            call gseti (gp, G_XNMINOR, clgeti ("minrx"))
            call gseti (gp, G_YNMAJOR, clgeti ("majry"))
            call gseti (gp, G_YNMINOR, clgeti ("minry"))

            call gseti (gp, G_ROUND, btoi (clgetb ("round")))

            if (clgetb ("logx"))
                call gseti (gp, G_XTRAN, GW_LOG)
            if (clgetb ("logy"))
                call gseti (gp, G_YTRAN, GW_LOG)

            # Draw axes using all this information.
            call glabax (gp, Memc[title], Memc[xlabel], Memc[ylabel]) (10)
        }

        pointmode = clgetb ("pointmode")
        if (pointmode) {
            call clgstr ("marker", Memc[marker], SZ_FNAME)
            szm = clgetr ("szmarker")
            call init_marker (Memc[marker], imark)               (11)
        }

        # Now to actually draw the plot.
        if (pointmode)
            call gpmark (gp, xvec, yvec, n, imark, szm, szm)     (12)
        else
            call gpline (gp, xvec, yvec, n)                      (13)

        call gflush (gp)
        call gclose (gp)                                         (14)
        call sfree (sp)
end
```

Comments on the indicated statements:

(1) It was previously mentioned that parameters should only be retrieved from the CL in the main procedure of a task. Well, here is special case where that isn't true. The immediate result of this is that we will have to retrieve the parameter file for the **prows** task into our work directory:

```
pl> concat plot$prows.par pairmass.par append+
```

(2) IRAF Graphics I/O (GIO) supports inumerable devices with no more fuss than you see here. The complications of how to get the same task to draw a plot to the terminal as well as a PostScript laser printer or a Calcomp or Versatec plotter is hidden not just from the user, but also from the programmer.

(3) Actually open the graphics device. This is similar to opening a file or an image in that a large data structure is allocated which is manipulated by the programmer through calls to procedures or references to macros that are defined in a *.h* include file. If `mode = append`, any previously existing plot will not be erased (or flushed to the plotter) before drawing the new plot. The final argument to `gopen` specifies the destination for the output metacode and should likely always be `STDGRAPH` for any programs that you write.

(4) This chunk of code chooses the axis limits for the plot. Note that the tolerance scheme that is used doesn't allow the user to default only the lower or upper limit separately. We could change this to do so, say by having an `INDEF` value for the `wx1`, `wx2`, `wy1`, or `wy2` parameters indicate a default value (the defaults are passed as arguments to `draw_vector`), but this sort of user interface change should be considered carefully before being implemented. The user of our new task will presumably be experienced with IRAF and may not take kindly to having parameters that are familiar from other tasks behave differently in this task.

(5) Actually specify the axis limits for the plot.

(6) Specify the viewport limits for the device. The viewport is the area of the plotting device (for instance, the graphics terminal) in which the plot will be drawn. The coordinate system is *Normalized Device Coordinates*, or NDC, which run from zero at one edge of the device to one at the opposite edge.

(7) The `gseti` routine is used to set various parameters of the plot, in this case, whether the aspect ratio of the plot will be forced to unity for the specified device, rather than filling the available viewport.

(8) The `draw_vector` procedure allows users the freedom to specify a title for the plot as a parameter of the calling task. This is useful, but for a special purpose task the user should not be forced to choose a title, since sufficient information is available within the task to label the average plot sensibly. Therefore `draw_vector` also provides for a default title to be passed as an argument of the procedure.

(9) Set more parameters of the plot. In this case, the values are the number of major and minor tick marks that will be generated for for each axis of the plot. The axis labeling heuristic takes these values as advice that may be ignored if it sees fit.

(10) Actually draw the axes.

(11) The `init_marker` procedure is discussed below.

(12) The plot can either consist of individual points or of connect-the-dot type line segments, depending on whether `pointmode` is true or false. The `gpmark` procedure will draw individual marks of the specified type and size at the set of coordinates specified by `xvec` and `yvec`.

(13) The `gpline` procedure draws a line connecting successive points specified in `xvec` and `yvec`. Note that the order of the points is important since no sorting is performed. In both the case of `gpmark` and of `gpline`, marks and line segments that extend beyond the bounds of the plot (as defined by either the axis limits or the viewport limits) are suppressed.

(14) These two statements flush the plot to the device and close the graphics descriptor. The former is actually unnecessary since closing the graphics subsystem also flushes the buffer.

What is the `init_marker` procedure? Well, it turns out that it is a rather peculiar routine that more-or-less duplicates the functionality of the `strdic` function that was introduced in §4, but for the specific needs of the plot package:

```
pl> cd plot
pl> match init_marker *.x
initmarker.x:procedure init_marker (marker, imark)
t_graph.x:          call init_marker (marker, marker_type)
t_pcols.x:            call init_marker (Memc[marker], imark)
t_prows.x:            call init_marker (Memc[marker], imark)
t_pvector.x:            call init_marker (Memc[marker], imark)
cl> type initmarker.x
# Copyright(c) 1986 Association of Universities for Research in Astronomy Inc.

include <gset.h>

# INIT_MARKER -- Returns integers code for marker type string.

procedure init_marker (marker, imark)

char    marker[SZ_FNAME]        # Marker type as a string
int     imark                   # Integer code for marker - returned

bool    streq()

begin
        if (streq (marker, "point"))
            imark = GM_POINT
        else if (streq (marker,   "box"))
            imark = GM_BOX
        else if (streq (marker,  "plus"))
            imark = GM_PLUS
        else if (streq (marker, "cross"))
            imark = GM_CROSS
        else if (streq (marker, "circle"))
            imark = GM_CIRCLE
        else if (streq (marker, "hebar"))
            imark = GM_HEBAR
        else if (streq (marker, "vebar"))
            imark = GM_VEBAR
        else if (streq (marker, "hline"))
            imark = GM_HLINE
        else if (streq (marker, "vline"))
            imark = GM_VLINE
        else if (streq (marker, "diamond"))
            imark = GM_DIAMOND
        else {
            call eprintf ("Unrecognized marker type, using 'box'\n")
            imark = GM_BOX
        }
end
```

What should we do with *initmarker.x*? On the surface, it would seem simple to dispense with the file by replacing the call to `init_marker` with a corresponding call to `strdic`. This turns out not to be the case, however, since the integer codes in this instance are powers of two (see *lib$gset.h*), rather than the sequential integers returned by `strdic`. One could imagine a rococo scheme to generate one from the other, but why bother? We are trying to achieve our goal with as little fuss as possible. It is easier to simply grab the existing code and use it, than it is to digest the code and write a simplified version for our own task. The point at which one starts fiddling too much with what already exists is more than likely the point at which one should start over from scratch. So our next step is to grab *initmarker.x*:

```
pl> copy initmarker.x home$pairmass/
pl> cd home$pairmass
```

We are almost to the end of this example. What remains is merely to write the main routine for **pairmass** and to put the whole thing together! This sounds worse than it is. The task really just consists of reading some parameters that express some position in the sky (and of the observatory) and a date of interest. These positions and times are then precessed and converted into the proper format for the `airmass` routine which then will be looped over for a range of times throughout the desired day. The results are collected into arrays which are passed to `draw_vector`. On the next page is the main procedure of the **pairmass** task, followed by the somewhat more dense parameter file.

examples$src/pairmass/t_pairmass.x:

```
# PAIRMASS -- plot the airmass for a given RA & Dec on a given date.

procedure t_pairmass()

pointer sp, ut, air, buf
double  ra, dec, epoch, ra0, dec0, epoch0
double  longitude, latitude, st, ha, utd, resolution
int     day, month, year, nsteps, i
real    amin, amax

double  clgetd(), ast_mst(), airmass()                         (1)
bool    clgetb()
int     clgeti()

begin
        ra0 = clgetd ("ra")
        dec0 = clgetd ("dec")
        epoch0 = clgetd ("epoch")

        year = clgeti ("year")
        month = clgeti ("month")
        day = clgeti ("day")

        longitude = clgetd ("longitude")                       (2)
        latitude = clgetd ("latitude")

        resolution = clgetd ("resolution")
        nsteps = nint (24 * resolution) + 1                    (3)

        call smark (sp)
        call salloc (ut, nsteps, TY_REAL)
        call salloc (air, nsteps, TY_REAL)
        call salloc (buf, SZ_LINE, TY_CHAR)

        call ast_date_to_epoch (year, month, day, 12.d0, epoch)      (4)
        call ast_precess (ra0, dec0, epoch0, ra, dec, epoch)         (4)

        do i = 1, nsteps {
            utd = (i-1) / resolution
            call ast_date_to_epoch (year, month, day, utd, epoch)    (4)
            st = ast_mst (epoch, longitude)                          (4)
            ha = st - ra

            Memr[ut+i-1] = real (utd)                                (5)
            Memr[air+i-1] = real (airmass (ha, dec, latitude))
        }

        if (clgetb ("listout")) {                                    (6)
            do i = 1, nsteps {
                call printf ("%6.0m%8.4f\n")
                    call pargr (Memr[ut+i-1])
                    call pargr (Memr[air+i-1])
            }
        } else {
            call sprintf (Memc[buf], SZ_LINE,                        (7)
                "Airmass vs. UT, %d/%d/%d\nRA=%h, Dec=%h (%g)\nlong=%m, lat=%m")

                call pargi (month)
                call pargi (day)
                call pargi (year)
                call pargd (ra0)
                call pargd (dec0)
                call pargd (epoch0)
                call pargd (longitude)
                call pargd (latitude)

            call alimr (Memr[air], nsteps, amin, amax)
            call draw_vector (Memc[buf], Memr[ut], Memr[air],        (8)
                nsteps, 0., 24., amin, amax)
        }

        call sfree (sp)
end
```

examples$src/pairmass.par:

```
ra,r,h,,0.,24.,Right Ascension of the object
dec,r,h,,-90.,90.,Declination of the object
epoch,r,h,INDEF,,,Epoch of the coordinates

year,i,h,,,,Year of the observation
month,i,h,,1,12,Numerical month specification
day,i,h,,1,31,Day of the month

longitude,r,h,111.6,0.,360.,Longitude of the observatory
latitude,r,h,31.98,-90.,90.,Latitude of the observatory

resolution,r,h,4,0.25,,Number of UT points per hour
listout,b,h,no,,,"List, rather than plot, the airmass vs. UT"

wx1,r,h,0.,,,left user x-coord if not autoscaling                (9)
wx2,r,h,0.,,,right user x-coord if not autoscaling
wy1,r,h,0.,,,lower user y-coord if not autoscaling
wy2,r,h,5.,,,upper user y-coord if not autoscaling
pointmode,b,h,no,,,plot points instead of lines
marker,s,h,"box",\
  "point|box|plus|cross|circle|hebar|vebar|hline|vline|diamond",\
  ,point marker character
szmarker,r,h,5E-3,,,marker size (0 for list input)
logx,b,h,no,,,log scale x-axis
logy,b,h,no,,,log scale y-axis
xlabel,s,h,"Universal Time",,,x-axis label
ylabel,s,h,"Airmass",,,y-axis label
title,s,h,"default",,,title for plot
vx1,r,h,0.,,,left limit of device window (ndc coords)
vx2,r,h,0.,,,right limit of device window (ndc coords)
vy1,r,h,0.,,,bottom limit of device window (ndc coords)
vy2,r,h,0.,,,upper limit of device window (ndc coords)
majrx,i,h,5,,,number of major divisions along x grid
minrx,i,h,5,,,number of minor divisions along x grid
majry,i,h,5,,,number of major divisions along y grid
minry,i,h,5,,,number of minor divisions along y grid
round,b,h,no,,,round axes to nice values
fill,b,h,yes,,,fill device viewport regardless of aspect ratio?
append,b,h,no,,,append to existing plot
device,s,h,"stdgraph",,,output device
```

A few final comments on the source code for the task itself:

(1) A frequent error when using library procedures is to fail to declare the correct datatype for the procedure arguments or for external functions. Many of the **asttools** routines are double precision and have double precision arguments.

(2) A recent introduction into the NOAO umbrella of packages is the notion of an observatory database. Specific observatory dependent information, such as the longitude and latitude, can be kept in the database. The source code for the facility is in the file *xtools$obsdb.x*. You may use these routines and others in the **xtools** library by simply linking to it. If you are running **xc** directly, just append "`-lxtools`" to the compiler command line. If you are using the IRAF **mkpkg** facility (see below) to manage the compilation and linking, add `-lxtools` to the `LIBS` definition, as in "`$set LIBS = "-lxtools"`".

Examples of the use of the observatory database are scattered throughout the NOAO packages, for example in the **setairmass** and **observatory** tasks.

(3) The task could have asked for `nsteps` directly, but the step `resolution` was chosen as the more natural quantity for users to specify. You should carefully consider what the most graceful set of parameters is for the user.

(4) Here are the specific **asttools** routines that the **pairmass** task needs. A description of the routines can be found in §6 and the calling sequences on the *SPP/VOS Quick Reference Card*. Note that by simply adding your own tasks to the **example** package, as described in §7, the **asttools** are immediately accessable within your tasks.

(5) The `draw_vector` procedure requires *real* arrays as arguments, but the **asttools** routines produce *double precision* values. The `real` intrinsic function explicitly performs the conversion. SPP would also perform the datatype conversion automatically within the assignment statement, but it is clearer to do it explicitly.

Also note the pointer dereferencing that corresponds to a simple array element specification. A one indexed array reference is constructed from a zero indexed pointer reference by subtracting one. **IRAF arrays are one indexed.**

(6) A useful option for plotting tasks is to allow the usual plot output to be redirected to a text listing of the points. This allows the user to interact with the plot data in almost any desired way, for instance as input to **mongo**.

(7) As can be seen in the plot on page 25, the title string can be made to extend over multiple lines. This is done by simply breaking the title string with newlines.

(8) Actually call `draw_vector`. As with the **asttools** routines, note that by adding your tasks to the **example** package, they may call `draw_vector` themselves since the compiled routine is kept in the package library.

(9) While it was deemed to be outside the scope of this document to describe the *parameter set* (or *pset*) mechanism of the IRAF CL, this would clearly be a good place to use an external pset to maintain the graphics parameters of the task. Psets are described in the document *Named External Parameter Sets in the CL*, by Doug Tody. Examples of their use are scattered throughout the newer packages, for instance within the **apphot** package.

We are now left to staple it all together. The key to making this work is to come up with the proper *mkpkg* file to automate the chore:

home$pairmass/mkpkg:

```
# Make the PAIRMASS task.

relink:                                                          (1)
        $set    LIBS = ""                                        (2)

        $update libpkg.a                                         (3)
        $omake  x_pairmass.x                                     (4)
        $link   x_pairmass.o libpkg.a $(LIBS) -o xx_pairmass.e   (5)
        ;                                                        (1)

libpkg.a:                                                        (6)
        @asttools
        airmass.x       <math.h>                                 (7)
        drawvector.x    <gset.h> <mach.h>
        initmarker.x    <gset.h>
        t_pairmass.x    <gset.h> <mach.h> <math.h>
        ;
```

A few final comments on the mkpkg file (also see the help page):

(1) A *mkpkg* file often has several internal modules that may be referenced on the command line. A module begins with a name terminated by a colon (`:`), and ends with a line consisting of a single semicolon (`;`). The command **`mkpkg relink`** will rebuild the *x_pairmass.e* executable, for instance, while the command **`mkpkg libpkg.a`** will rebuild only the library. If no module is specified, the first one encountered in the file will be executed, in most cases this is equivalent to a `mkpkg relink`.

(2) Declare any external libraries that are needed to link the task executable. In this case there are none. In other cases in which you borrow software and it may not be obvious whether the libraries specified in the package's *mkpkg* file are needed for your particular task, first try linking without any of the libraries. If there are unresolved references add libraries one by one from the list until the error messages disappear.

(3) There are a number of **mkpkg** directives to perform useful chores. Directives begin with a dollar sign (`$`) and are described in the **mkpkg** help page. The `$update` directive specifies that the a library module, described elsewhere in the file, should be processed by compiling and archiving the listed source files and subdirectories.

(4) Compile a single file. The process definition file *x_pairmass.x* is kept outside the package library to allow the executable to be easily linked.

(5) Link the list of files and libraries into the task (or package) executable.

(6) Entire subdirectory trees may be compiled and archived into the package library. This is a very useful feature (as anyone who has used the X windows **imake** facility will attest). The entire IRAF directory tree contains a web of *mkpkg* files that reference other files further down in the tree. This is true both for library subdirectories, as here, and for triggering updates of whole subpackages.

(7) Simply specify that a single source file be compiled and archived into the package library. A list of dependency files (typically header files) follows the source filename on the same line. The file will be recompiled only if the archived object file is older than the source file or a dependency file, or if the object doesn't yet exist.

The command to compile and link the task is now simply:

```
cl> cd home$pairmass
cl> mkpkg
```

And to declare the task to the CL:

```
cl> task pairmass = home$pairmass/xx_pairmass.e
```

Finally your new task is ready for use. The command line to make the plot on the first page of this section was:

```
cl> pairmass ra=9:51:42 dec=69:56 epoch=1950 \
>>> year=1992 month=2 day=2
```

Note that this is a fully functional task. You may call up the IRAF *cursor mode* using the command `= gcur`, for instance. A useful addition to the task might be to graft on a graphics cursor loop similar to the image cursor loop in the **impix** example.

5.1. Summary

(0) **Don't modify tasks in the IRAF directory tree!**

(1) Decide what you want to do. Investigate what IRAF has to offer.

(2) Make a working directory and copy the source and parameter files to that directory.

- Examine the package script (*e.g.*, *astutil.cl*) to determine whether a task is a compiled SPP task, a script, or a foreign task.
- Examine the *mkpkg* file to understand the architecture of the package and tasks.
- Examine the process definition file (*x_astutil.x*) to find the names of the task entry point routines.
- Search the source code for various procedure and macro names using the **match** task (or various host commands).

(3) If there is more than one source file, copy the *mkpkg* file and the process definition file (*x_astutil.x* in this examples) to your working directory. If there is only one file, you may find it easier to edit the task statement directly into that file.

(4) Copy any other "utility" source and parameter files, perhaps from different packages. (Don't forget any required *.h* header files!) In this example, the files are *t_setairmass.x*, *it_prows.x*, *initmarker.x*, and the entire *asttools* subdirectory. Files are borrowed from both the *astutil* and *plot* packages (and directories).

(5) Stitch the source code together into a coherent task. This may be a one line change to some algorithm, or may be a complete rewrite as in this example.

(6) Modify the package "glue" to support your new task. This may involve changes to the *mkpkg* and process definition files. We have not mentioned help pages yet, but you may also want to modify a preexisting help page to document your local modifications. This will be described in §7.

(7) Declare the task to the CL and test it. A typical sequence of actions while developing a task is to edit the source files, execute **mkpkg** which will only update the out-of-date files, test the results, and repeat.

(8) Install the task. This may be as simple as editing the task statement into your *loginuser.cl* file, or may involve steps to make the task known to other users. These will be described in §7.

6. The IRAF VOS Interfaces

The biggest difficulty that you will likely have getting into SPP programming is the very large number of IRAF *Virtual Operating System* routines to choose from. It can be difficult to figure out where to look for a particular routine to perform even an everyday chore such as writing to a file, let alone to do something complicated like manipulating a *World Coordinate System* (WCS). This section will cover the most frequently used routines from the more frequently used VOS interfaces. Some routines (*e.g.*, for asynchronous binary input/output) as well as some entire interfaces (*e.g.*, the WCS interface) will of necessity be omitted.

The most important thing to keep in mind is that the IRAF directory structure mirrors its functionality. The source code for the IRAF VOS is contained under the *sys$* directory and, for example, the FMTIO (*formatted input/output*) interface is defined in *fmtio$ = sys$fmtio/*. If you find yourself completely baffled about some point of VOS usage, a tour through the source code for that part of the VOS may be in order. There are often useful comments located in the individual source files or even entire system design documents (usually in a subdirectory called *doc*). The individual component routines of an interface are typically located in files that have the same name (or a simple variant) as the routine. Note that not all, or even most, of the routines in an interface directory are meant to used outside of the interface, itself. Many of the routines comprising a typical interface are internal to that interface, and should not be called by user programs. To do so would be to “violate the interface”, leaving such compromised programs open to future problems ranging from the cryptic to the catastrophic, should the interface ever be reorganized internally.

As you read through the descriptions of the individual routines, you may wish to consult §6.11, *Arguments & Return Values*, for general comments regarding their datatypes.

6.1. The Command Language I/O Interface (*clio*)

The CLIO (*Command Language I/O*) interface is used to exchange information between a task and the IRAF Command Language, typically by getting or setting the value of a CL parameter. The routines in this interface are usually invoked by the top level procedure of a task as the first step in executing the task. A sequence of `clgstr` and `clget_` calls (to get string valued and typed parameters, respectively) is used to supply the task with the values of query and hidden parameters that the user has selected within the Command Language.

The central set of routines in CLIO allows a task to query the CL for the values of parameters, or conversely to instruct the CL to set those values. The single most often used routine is `clgstr`, which requests the value of a *string* parameter. Note that routines that write to a character string must also be supplied with the maximum number of characters that the string allows. There are several different typed `clget_` routines, where the underscore (_) indicates that a specific letter should be substituted, such as `i` for an `integer` parameter, or `r` for a `real`. More arcane parameter queries are possible, such as using `clgwrd` to expand an abbreviated keyword from a dictionary string. Task usually use the index into that dictionary, rather than the keyword, itself. The `clgcur` routine can be used to query a cursor typed parameter (actually a *list directed cursor* parameter) in order to cause the hardware graphics or image cursor to be read. Finally, note the usual restriction that a background task cannot write into its parameter file. This means that the `clpstr` and `clput_` routines should be used with care.

	`clgstr (param, string, maxch)`	**Get a string parameter.**
`value =`	`clget_ (param)`	**Get a typed parameter.**
`stat =`	`clgwrd (param, keyword, maxch, dict)`	**Look up a parameter in a dictionary.**
`stat =`	`clgcur (param, x,y,wcs, key,cmd,maxch)`	**Read a cursor parameter.**
	`clpstr (param, string)`	**Output a string parameter.**
	`clput_ (param, value)`	**Output a typed parameter.**

Many IRAF tasks use the following set of routines to manage parameters that represent *file templates* (also called *file lists*). The actual file template mechanism is a part of the FIO interface, but these routines provide a simplified means of accessing it. The basic idea is to expand a template specification (provided as the value of a parameter) into a internal data structure (referenced by the pointer `list`, below), and then to retrieve individual file names from that list. Note that these routines specifically support *file* templates, while *image* templates (including the image section notation) are managed by another set of routines described under IMIO, below. One comment: when using `clpopni`, the task will automatically sense when the `STDIN` has been redirected and will open the `STDIN` as the sole member of `list`.

`list =`	`clpopni (param)`	**Open a file template or the `STDIN`.**
`list =`	`clpopns (param)`	**Open a sorted template.**
`list =`	`clpopnu (param)`	**Open an unsorted template.**
	`clpcls (list)`	**Close a file template.**
`nfiles =`	`clplen (list)`	**Return the number of files in a list.**
`stat =`	`clgfil (list, string, maxch)`	**Get the next file name.**

Every interface has a number of useful utility routines. One such routine in CLIO that can be particularly useful (but also potentially quite dangerous to use) is `clcmdw`, which will execute another IRAF task from within your compiled task. The danger is that your program will break if the external task is changed - say in a future release of IRAF. The `clcmd` variant will execute a CL command line and immediately return without waiting for the completion of that command. Due to the restraints of nonsynchronous programming, this is not typically a good idea. A more fundamental problem with using these routines is with the internal “plumbing” that may be needed to connect (in whatever way) to the external task. This plumbing is unlikely to mesh very well with the internal structure of your task.

`clcmdw (command)`	**Send a command to the CL and wait.**
`clcmd (command)`	**Send a command to the CL, return.**

6.2. The File I/O Interface (*fio*)

The FIO (*File I/O*) interface is used to access the host file system. The chores include opening and closing files, reading and writing binary and raw character (unformatted) data, testing the accessibility of files, and expanding filename templates. It also includes more technical facilities such as seeking to particular offsets within files, parsing the different fields of host pathnames, and managing the IRAF virtual filename mapping scheme. Reading and writing formatted data is handled by the FMTIO interface.

As with other systems and languages, a file must be *opened* before it can be read from or written to. When you are finished with the file it should be explicitly closed - you can’t rely on IRAF doing this for you implicitly when the task exits. FIO buffers data as it is being written. Use the `flush` routine to force the data to be actually written to disk, rather than just to the buffer. If this is not done, FIO will empty the buffer only when it fills (unless `fseti` has been used to reconfigure the buffering, see below).

`fd =`	`open (fname, mode, type)`	**Open a file for I/O.**
	`close (fd)`	**Close a file when finished.**
	`flush (fd)`	**Flush the buffers immediately.**

Binary data is read or written using `read` or `write`. The `maxch` parameter specifies the maximum number of SPP `char`’s to be transferred, either into the internal buffer or onto disk. A read often terminates before `maxch` characters are read, for instance when `EOF` is reached or when reading from a terminal. A write will always transfer exactly `maxch` SPP `char`’s unless an error occurs. Note that the units of data transferred are SPP `char`’s and that a `char` is two bytes in SPP. Only even numbers of bytes can be read or written using `read` or `write`. Lower level IRAF facilities must be used to get around this restriction, although this is usually not necessary. Datatypes other than `char` can be transferred by providing a buffer array of a different type, although `maxch` must still give the size of the array in units of `char`'s.

`stat =`	`read (fd, buffer, maxch)`	**Binary byte stream input.**
	`write (fd, buffer, maxch)`	**Binary byte stream output.**

IRAF interfaces are typically internally configurable. It is usually a bad idea for the user to fiddle with various options, many of which are chosen to tune the interface's performance. Should it be disirable, the `fseti` and `fstat_` routines provide the functionality to fiddle FIO. One case in which it may, indeed, be useful is setting the `F_FLUSHNL` option to cause an output buffer to be flushed after every line of output: `call fseti (STDOUT, F_FLUSHNL, YES)`. This should only be done for a task which involves a lot of user interaction. More "batch" oriented tasks should rely on the normal FIO buffering to optimize efficiency.

	`fseti (fd, param, value)`	**Set a FIO option.**
`stat =`	`fstat_ (fd, param)`	**Get the status of an open file.**

These are some of the more useful FIO utility routines. Each is fairly self-explanatory. The `access` routine tests whether a file is accessable under various modes (*e.g.*, `READ_ONLY`) or whether the file is a certain type (*e.g.*, `TEXT_FILE`). The modes and types are listed in the file *hlib$iraf.h*. `Delete` and `rename` do what they say. The `mktemp` procedure generates the name of a file that is guaranteed (at the time the procedure is called) to be nonexistent on the computer. The `protect` procedure causes a file to be protected from deletion. The actual protection mechanism varies from system to system. Under VMS the deletion permission is removed from the file. Under Unix (where deletion is controlled by the file permission of the directory in which the file resides) a similar protection is provided by establishing a hidden link to the file. This link provides safety against deletion within IRAF, but does nothing to protect a file under Unix (other than to provide another directory entry that must be removed to actually delete the file).

`stat =`	`access (fname, mode, type)`	**Can the file be accessed?**
	`delete (fname)`	**Delete the named file.**
	`rename (old_fname, new_fname)`	**Rename a file.**
	`mktemp (root, fname, maxch)`	**Make a temporary file name .**
`stat =`	`protect (fname, action)`	**Protect or unprotect a file.**

These two routines provide unformatted line oriented text I/O to a file. On input, `linebuf` must be at least `SZ_LINE` characters long.

`stat =`	`getline (fd, linebuf)`	**Get a line of text from a file.**
	`putline (fd, linebuf)`	**Output a line of text to a file.**

6.3. The Image I/O Interface (*imio*)

The IMIO (*Image I/O*) interface allows IRAF images of various underlying formats to be accessed at both the pixel and header levels. There are also routines supporting image sections and templates. IMIO protects the user from needing any knowledge of the particular format that IRAF uses to store its image data. In fact, IRAF images can be stored in any of a number of formats depending on the particular type of data being manipulated. All of the different formats (including the native IRAF *Original Image Format* (OIF), the *Space Telescope Format* (STF), the *Quick Position Ordered Event* format (QPOE), and the *Pixel List Format* (PLF) are transparently accessible to the user via the exact same IMIO routines. When a task is executed that uses the IMIO routines, the IRAF environment variable, *imtype* can be set to determine the datatype of any newly created output images. This can also be specified on the command line by providing the correct image extension, *e.g.*, *imh* for the typical (ground based) OIF image, or *qp*. for a ROSAT xray image.

The major conceptual hurdle to using these routines is due to the optimization of the interface for accessing large number of pixels within each individual I/O operation. One does not *open* an image for reading or writing, one *maps* it using `immap` to make copies in memory of the complicated data structures within the disk image file, and to establish an efficient pixel buffering mechanism. The image header data structures are defined in the system include file *imhdr.h*. The most straightforward way to access such information as the dimensionality or size of an image is to reference the individual data structure fields using the macros defined in this header file. Instances of this are to be found in various examples from this document. When you are finished, each image should be *unmapped* using `imunmap`, ideally in the reverse of the order that they were opened.

`im =`	`immap (image, mode, hdr_arg)`	**Map ("open") an image.**
	`imunmap (im)`	**Unmap ("close") an image.**

One of the most used IMIO utility routines is `imflush`, which causes the image buffers to be flushed to disk. This is important to maintain synchronization between the reading and writing of an image that is open with a mode of `READ_WRITE`. One other utility worth mentioning is `imdelete` which should be used if you need to delete an image from within a task. A simple delete is not good enough since an image typically consists of more than one file. Note that an image must be closed (unmapped) before deleting it.

`imflush (im)`	**Flush the image buffers.**
`imdelete (image)`	**Delete an (unmapped) image.**

There are numerous routines that actually read or write the image pixels. They differ by the datatype of the supplied buffers, by the assumed dimensionality of the image, by whether the pixels are returned line-by-line versus by entire image sections, and by whether the pixel access is sequential or random. Rather than describe each of the many routines in gory detail, we will present an overview of these general characteristics, and rely on the example code to clarify issues of usage.

Each of the three pairs of routines described below actually represents six times as many datatype specific routines. The underscore (_) in the name of each routine is to be replaced by one of the set of letters [`silrdx`] indicating the six datatypes (in order): `short`, `integer`, `long`, `real`, `double`, or `complex`. Other datatype images may be available for special purposes, but are not generally supported within IRAF. Also a `complex` image may be awkward to handle due to a dirth of `complex` typed utility procedures in other interfaces.

In general, most IRAF image operations involve (signed) `short` or `real` images for raw or processed data, respectively, from CCDs and similar detectors. `Double` precision images are usually too costly in terms of disk space and often are not justified by the precision of the input images or of the applied operations. Astronomical detectors typically don't have the dynamic range to justify the `integer` or `long` datatypes.

Finally, note that the datatype of the image being accessed does not have to match the datatype of the IMIO routines being used. The "normal" automatic datatype conversions will be made, both on input and output. For this reason and given the discussion above, the examples in this document have used the `real` datatype versions of the various pixel input and output routines. This will result in truncated precision (at the very least) if the tasks are used with `double` or `complex` images and potentially with `long` integer images. As an alternative, more robust IRAF tasks (such as **imarith**) allow the user to choose the datatypes of both the output image and of the internal calculation that is performed.

The following suite of routines (36 in all) provides random access, line oriented input / output for one, two, or three dimensional images (`N = 1, 2, or 3`) of the various datatypes. While it is possible to access images of different dimensionality than the routine being used, this access is limited by the mismatch, for instance, reading a three dimensional image using `imgl2r` is limited to the first band of the image. On the other hand, when reading (or writing) a two dimensional image using `imgl3r` (or `impl3r`) the programmer is responsible for ensuring that the task does not reference outside the first band of the image (to avoid a `Pixel subscript out of bounds` error).

```
buf =    imglN_ (im [,line [,band]])          Get a line from an image, N=[123].
         implN_ (im [,line [,band]])          Output a line to an image, N=[123].
```

These routines (36 again) provide random access to image sections of apparently one, two, or three dimensional images. The same dimensionality mismatch cautions apply as above. While outside the scope of this document, the *boundary extension* mechanism is worth a brief mention at this point. If boundary extension is disabled, a pixel reference outside the bounds of the image will generate an error. With boundary extension, on the other hand, values for out of bound pixels can be generated using any of several heuristics, *nearest neighbor* and *constant* boundary extension being the two simplest.

```
buf =    imgsN_ (im, x1,x2 [,y1,y2 [,z1,z2]])   Get a section from an image, N=[123].
         impsN_ (im, x1,x2 [,y1,y2 [,z1,z2]])   Output a section to an image, N=[123].
```

These dozen typed routines provide sequential line-by-line access to images of arbitrary dimensionality. These are very useful with tasks that implement pixel-by-pixel independent algorithms, since no prior knowledge of the dimensionality (or size) of the input images is necessary. A typical application will loop over the routines until an `EOF` is returned, processing an image line at a time using the vector operators.

```
stat =   imgnl_ (im, bufptr, v)               Generalized get next line.
         impnl_ (im, bufptr, v)               Generalized output next line.
```

Other IMIO routines are available that access generalized image sections or generalized datatypes. These will not be discussed further.

In addition to the system level header keywords that keep track of such necessary items as the pixel datatype, dimensionality, and the size of an image, there may be any number of *user area* keywords to keep track of data specific information such as filters and exposure time. The routines below allow access to this information. The usage should be fairly self-explanatory. The IRAF environment variable *min_lenuserarea* should be increased from its default value of `8000` if your image headers are being truncated.

```
stat =   imaccf (im, key)                     Does the named keyword exist?
         imaddf (im, key, type)               Add, but don't initialize, a keyword.
         imadd_ (im, key, value)              Add or modify a header keyword.
         imastr (im, key, value)              Add or modify a string keyword.
         imdelf (im, key)                     Delete a header keyword.
value =  imget_ (im, key)                     Get a keyword of the specified type.
         imgstr (im, key, string, maxch)      Get a string valued keyword.
         imput_ (im, key, value)              Modify an existing header keyword.
         impstr (im, key, value)              Modify an existing string keyword.
code =   imgftype (im, key)                   What is the keyword's datatype?
```

The following routines implement the image section template mechanism. Usage is very similar to the template routines that are described under the CLIO interface. Note, however, the availability of routines to randomly access the template and to rewind the image list. An image template differs from a file template mostly in the syntax used to specify image sections.

`imt = imtopenp (param)`	**Open an image template.**
`imtclose (imt)`	**Close an image template.**
`stat = imtgetim (imt, string, maxch)`	**Get the next image name.**
`stat = imtrgetim (imt, index, string, maxch)`	**Get a randomly indexed image name.**
`nimages = imtlen (imt)`	**Return the number of images in a list.**
`imtrew (imt)`	**Rewind an image list.**

6.4. The Memory I/O Interface (*memio*)

The MEMIO (*Memory I/O*) interface supports the dynamic memory allocation mechanism of IRAF. Allocating memory dynamically frees IRAF programs (tasks) from containing huge amounts of static storage for images and other data files. Besides issues of programming efficiency, avoiding the allocation of static arrays free tasks from compiled in limits on the sizes or numbers of images that can be handled. MEMIO supports both stack and heap oriented memory allocation.

These three routines are all that are needed to allocate memory on the *stack*. The normal way to use the stack facilities is to first *mark* the current position of the *stack pointer* using `smark`. A number of subsequent calls to `salloc` are used to obtain pointers to dynamically allocated new arrays. These pointers are then dereferenced using the `Mem_[]` constructs to actually access these arrays. At the end of the calling routine, the dynamically allocated memory is returned with a single call to `sfree`.

`smark (sp)`	**Save the current stack pointer.**
`salloc (ptr, nelem, type)`	**Allocate space on the stack.**
`sfree (sp)`	**Pop the stack.**

These routines, by comparison, are used to allocate memory on the so-called *heap*. As the name suggests, the heap is just a pile of memory that can be partitioned up into separate lumps to support various chores. If a task has a associated internal data structure, this structure is usually allocated on the heap. Unlike the stack which should be marked and freed within a single routine, space on the heap is typically allocated in one routine, but freed some time later from within an entirely different routine. Also, space on the heap need not be freed in any particular order. Only the pointer returned by `malloc` need be passed around from routine to routine to allow the data structure to be referenced. Note the options to allocate a zeroed buffer and also to change the size of a previously allocated buffer.

`malloc (ptr, nelem, type)`	**Allocate space on the heap.**
`calloc (ptr, nelem, type)`	**Allocate and zero space.**
`realloc (ptr, nelem, type)`	**Adjust the size of a buffer.**
`mfree (ptr, type)`	**Free space on the heap.**

6.5. The Graphics I/O Interface (*gio*)

The GIO (*Graphics I/O*) interface allows the creation and manipulation of device independent plots. All IRAF graphics tasks create a metacode representation of the desired plot, although a user may never have to interact with the metacode directly. In the normal situation of generating a plot to be viewed on the terminal or sent to a plotter, the metacode is routed and translated by the CL into whatever is appropriate for the given graphics device (*e.g.*, Tektronix codes).

Within the source code for your task, on the other hand, the programmer is also insulated from the details of this routing and translation. The necessary device dependent information is maintained within the *dev$graphcap* file and the IRAF system manager (perhaps yourself wearing a different hat) is responsible for configuring your IRAF installation to support the particular devices that are available at your site. These topics are outside the scope of this document. You are directed to the documents, the *Sun/IRAF Installation Guide* and the *Sun/IRAF Site Manager's Guide* as a good place to start. Other computers may also have versions of these documents available that are tailored to the particular machine.

As usual with these interfaces, GIO includes a routine to initialize the wide variety of internal parameters and data structures that are used by the individual procedures. This is the `gopen` routine, which returns a pointer to a graphics descriptor which will be used in subsequent calls to the device. When the task is finished with the graphics device it should be closed and the memory required by the graphics descriptor returned to the system using `gclose`.

Entire IRAF plots may be buffered by the system depending on the device. Plots that are sent to the terminal are obviously not buffered at this level although polyline output is also internally buffered. Hardcopy plotters may typically have an associated buffer that holds perhaps eight plots. The `gflush` procedure will cause the buffered plots to be actually delivered to the device. Note that this is automatically done when the device is closed with `gclose` in any case. `Gflush` also reinitializes certain internal data structures and is thus useful for synchronizing graphics and other I/O.

`gp = gopen (device, mode, fd)`	**Open a graphics stream.**
`gclose (gp)`	**Close a graphics stream.**
`gflush (gp)`	**Flush the graphics output.**

The underlying graphics routines are the usual variety of move and draw commands for lines, markers, polylines and polymarkers:

`gline (gp, x1, y1, x2, y2)`	**Draw a line from `(x1,y1)` to `(x2,y2)`.**
`gpline (gp, xa, ya, npts)`	**Draw a polyline.**
`gmark (gp, x, y, type, xs, ys)`	**Draw a marker of a given size.**
`gpmark (gp, xa, ya, npts, type, xs, ys)`	**Draw a polymarker.**
`gamove (gp, x, y)`	**Move the pen to the absolute position.**
`gadraw (gp, x, y)`	**Draw (absolute) from the current position.**

More exotic routines are perhaps best introduced by example, see §5 for such examples of setting various GIO options and of scaling and drawing axes for a plot.

`gseti (gp, param, value)`	**Set a GIO option.**
`gswind (gp, x1, x2, y1, y2)`	**Set the window in the world coords.**
`gsview (gp, x1, x2, y1, y2)`	**Set the viewport in NDC.**
`gascale (gp, array, npts, axis)`	**Scale the axis to fit the data.**
`glabax (gp, title, xlabel, ylabel)`	**Draw and label the axes.**

The `gpagefile` routine is useful for accessing a file of help information from within a graphics cursor loop:

`gpagefile (gp, file, prompt)`	**Page a file from graphics cursor mode.**

6.6. Vector Operators (*vops*)

The IRAF *Vector Operators* form a coherent and extensible interface to any vector hardware that a host computer may support. Virtually all IRAF tasks that manipulate images make use of the VOPS procedures. Even on an average workstation that may not include any special vector support, the fact that the image arithmetic tends to be concentrated into a well defined set of vector operators lends itself to easily optimized code design. There are too many individual VOPS routines to describe here, but a few are mentioned below. A typical vector operator has typed input and output vectors that are the same length. The more fundamental operators, such as `amov`, are designed to allow the input and output arrays to overlap. The range of available datatypes (indicated below by an underscore) varies from routine to routine.

In general, the only way to be sure of the currently available set of vector operators and of the supported datatypes for each is to examine the source code in the *vops$* directory. The vector operators are written using the IRAF **generic** preprocessor. This allows a single source code file to generate the individual datatype dependent routines. Read the **generic** help page for more information. **The VOPS routines are also callable from IMFORT programs.**

	`amov_ (a, b, npix)`	**Copy a vector.**
	`amovk_ (k, b, npix)`	**Copy a constant into a vector.**
	`aabs_ (a, b, npix)`	**Absolute value of a vector.**
	`aadd_ (a, b, c, npix)`	**Vector `c` is the sum of vectors `a` and `b`.**
	`aaddk (a, b, c, npix)`	**Vector `c` is the sum of vector `a` and scalar `b`.**
	`aavg_ (a, npix, mean, sigma)`	**Mean and sigma of a vector.**
`value =`	`amed_ (a, npix)`	**Return the median of a vector.**
	`abav_ (a, b, nblocks, blocksize)`	**Block average of a vector.**
	`abeq_ (a, b, c, npix)`	`c[i]=1` **if** `a[i] == b[i]`**, else** `c[i]=0`.
	`alim_ (a, npix, minval, maxval)`	**Compute the min and max of a vector.**

6.7. Miscellaneous (*etc*)

The IRAF *etc$* directory contains a grabbag of handy routines that don't fit into the normal suite of VOS interfaces. In addition to a variety of individual routines there are a few "mini-interfaces" of a dozen or a half-a-dozen routines each. Some of these are highly technical, such as those that implement subprocess control or the internals of the environment mechanism. These two in particular are really only meant to be used in the guts of the IRAF Command Language. Others are more general purpose such as the SPP error calls or the routines that get or set the values of environment variables.

These two routines make a sandwich with the SPP error handling mechanism in the middle. The `error` procedure is used to explicitly trigger an error handler (posted within an `iferr` block) or to simply abort a task. The `erract` procedure, on the other hand, is invoked at the end of the error handler block to provoke a particular final outcome. There are four basic such outcomes: do nothing, print a warning message (`call erract (EA_WARNING)`), immediately terminate the task (`call erract (EA_FATAL)`), or trigger another error action to be handled by the routine that called the current routine (`call erract (EA_ERROR)`). The system header file *imhdr.h* must be included to define the three error codes, `EA_WARNING`, `EA_ERROR`, and `EA_FATAL`. `Erract` should only be called from within an error handler. In order to correctly trap an error generated by a subprocedure, that subprocedure must be referenced in an `errchk` statement before the `begin` statement of the calling procedure. In general, it is a good idea to `errchk` all subprocedures in an `iferr` block that could possibly generate and error condition.

`error (code, message)`	**Generate an error condition.**
`erract (severity)`	**Take an error action.**

These routines allow your tasks to query the values of environment variables. This is not something user programs typically have to do, but may be useful for some special purposes.

```
  len =    envgets (key, value, maxch)
value =    envget_ (key)
```

`len = envgets (key, value, maxch)`	**Fetch the string value of an env. variable.**
`value = envget_ (key)`	**Return the typed value of an env. variable.**

These two routines implement the quick sort algorithm which will efficiently sort a large list of items. For shorter lists a simple bubble or insertion sort may be more appropriate since the quick sort has a bit of start up overhead. The `compare` function is an integer function supplied by the task that has two integer arguments. These two arguments will be substituted with the values of various elements from the integer `array`. The function should return a value of `0` (zero) if the two arguments are identical, should return `-1` if the first argument is less than the second, and should return `1` if the first argument is greater than the second. The second form of the routine, `gqsort`, allows passing an additional integer argument that can be used for context data or bookkeeping. The calling sequence of the function can be summarized as `-1,0,1 = compare ([arg,] x1, x2)`.

`qsort (array, nelems, compare)`	**Sort an integer array using** `compare`**.**
`gqsort (array, nelems, compare, arg)`	**Sort by a function with an argument.**

There are various routines that are handy for a wide variety of purposes. The `pagefile[s]` routines allow access to help information from inside a running task, for instance a cursor loop task. The `urand` function returns a pseudo-random `real` number in the range from `0` to `1` (zero to one). A handy identification string can be generated using the `sysid` procedure. These strings are most familiar from the tops of IRAF plots. The `btoi` function is useful for converting a boolean value (`true` or `false`) into an integer code (`YES` or `NO`) in order to store the value portably in a data structure.

`pagefiles (files)`	**Page text file(s) on the STDOUT.**
`pagefile (file, prompt)`	**Page a text file on the STDOUT.**
`real = urand (seed)`	**Uniform "random" number in (0,1).**
`sysid (string, maxch)`	**Return a system and user ID string.**
`int = btoi (bool)`	**Convert a** `bool` **to an** `int` **(**`YES` **or** `NO`**).**

6.8. The Formatted I/O Interface (*fmtio*)

The FMTIO (*Formatted I/O*) interface is used to generate formatted human-readable output and to read similarly formatted (usually whitespace delimited) input. The capabilities are similar to those of the C STDIO library routines. Note that IRAF tasks do not use Fortran I/O routines. FMTIO also includes string comparison and searching abilities.

The fundamental routines for generating output are the different varieties of the `printf` procedure. Currently in SPP a call to one of these `printf` routines is only used to initialize the process of writing the formatted output. The individual variables to be printed are supplied through individual calls to the `parg` procedures.

`printf (format)`	**Begin a print to the** `STDOUT`**.**
`eprintf (format)`	**Begin a print to the** `STDERR`**.**
`fprintf (fd, format)`	**Begin a print to a file.**
`sprintf (string, maxch, format)`	**Begin a print to a string.**
`clprintf (param, format)`	**Begin a print to a CL parameter.**
`parg_ (value)`	**Complete a typed format.**
`pargstr (string)`	**Complete a string format.**

The fundamental routines for reading "list directed" input are the different varieties of the `scan` procedure. Currently a call to one of these `scan` routines is only used to initialize the process of reading the input. The individual variables to be scanned are supplied through individual calls to the `garg` procedures.

```
stat =      scan ()                         Begin a scan from the STDIN.
stat =     fscan (fd)                       Begin a scan from file.
stat =     sscan (str)                      Begin a scan from a string.
stat =    clscan (param)                    Begin a scan from a CL parameter.
           garg_ (value)                    Get a typed value.
         gargstr (string, maxch)            Get the rest of the line.
         gargwrd (string, maxch)            Get the next "word".
```

FMTIO also includes a wide variety of routines for processing character strings, only a few of which are:

```
stat =    strdic (input, keyword, maxch, dict)   Look up a string in a dictionary.
bool =      streq (string1, string2)             Compare two strings for equality.
          strcpy (string1, string2, maxch)       Copy string1 to string2.
```

Format Specifications

An SPP format specification has the form "`%w.dC`", where *w* is the field width, *d* is the number of decimal places or the number of digits of precision, and *C* is the format code. The *w* and *d* fields are optional. The format codes *C* are as follows:

`b`	**boolean (`YES` or `NO`)**
`c`	**single character (`c`, `\c`, or `\nnn`)**
`d`	**decimal integer**
`e`	**exponential format, *d* is the precision**
`f`	**fixed format, *d* is the number of decimal places**
`g`	**general format, *d* is the precision**
`h`	***hms* format (`hh:mm:ss.ss`, *d* is the number of decimal places)**
`H`	***HMS* format, convert from degrees to hours first (divide by 15)**
`m`	***ms* or *hs* format (`mm:ss.ss`), *d* is the number of decimal places**
`M`	***MS* or *HS* format, convert from degrees to hours first**
`o`	**octal integer**
`rN`	**convert integer to or from radix *N***
`s`	**string, *d* is the maximum number of chars to print**
`t`	**advance to column given by *w***
`u`	**unsigned decimal integer**
`w`	**output the number of spaces given by *w***
`x`	**hexadecimal integer**
`z`	**complex format (`(r,r)`), *d* is the precision**
`*`	**deferred, get the field from the next `parg_` call**

Conventions for specifying the field width:

`w =  n`	**right justify and blank fill in a field of `n` characters**
`w = -n`	**left justify and blank fill in a field of `n` characters**
`w = 0n`	**right justify and zero fill in a field of `n` characters**
`w =  0`	**use as much space as needed**
`absent`	**same as `w = 0`**

Escape sequences for string literals and character constants:

`\b`	**backspace**
`\f`	**form feed**
`\n`	**newline (`<CR><LF>`)**
`\r`	**carriage return**
`\t`	**tab**
`\"`	**string delimiter character**
`\'`	**character constant delimiter character**
`\\`	**backslash character**
`\nnn`	**octal value of character**

Any combination of the fields *w*, *d*, *C*, or *N* may be specified as asterisks (`*`) in the format string, allowing the field to be passed at run time in a `parg_` call.

6.9. SPP Intrinsic Functions

A table of the supported SPP intrinsic functions is given below. Since the SPP intrinsic functions are derived from the underlying host system compilers and libraries, there may well be other intrinsic functions that aare available on a given host computer. You should resist any temptation that you may have to use them. To do so would compromise the portability of your programs. In general, the functions below form a rather complete set and using other functions would be more of a convenience than a necessity.

Type Conversions

`char =`	`char (integer)`	**Convert to character.**
`short =`	`short (z)`	**Convert to short.**
`int =`	`int (z)`	**Truncate to integer.**
`int =`	`nint (x)`	**Round to integer.**
`long =`	`long (z)`	**Convert to long integer.**
`real =`	`real (z)`	**Convert to real.**
`real =`	`aimag (complex)`	**Imaginary part.**
`double =`	`double (z)`	**Convert to double precision.**
`complex =`	`complex (z)`	**Convert to complex.**

Trigonometry (angles are in radians)

`y =`	`sin (y)`	**Sine.**
`y =`	`cos (y)`	**Cosine.**
`x =`	`tan (x)`	**Tangent.**
`x =`	`asin (x)`	**Inverse sine.**
`x =`	`acos (x)`	**Inverse cosine.**
`x =`	`atan (x)`	**Inverse tangent.**
`x =`	`atan2 (x1, x2)`	**Inverse tangent of** `x1/x2`**.**
`x =`	`sinh (x)`	**Hyperbolic sine.**
`x =`	`cosh (x)`	**Hyperbolic cosine.**
`x =`	`tanh (x)`	**Hyperbolic tangent.**

Miscellaneous

`w =`	`abs (z)`	**Absolute value.**
`complex =`	`conjug (complex)`	**Complex conjugate.**
`w =`	`min (w1, w2, ...)`	**Minimum value.**
`w =`	`max (w1, w2, ...)`	**Maximum value.**
`w =`	`mod (w1, w2)`	**Remainder after** `w1/w2`**.**
`y =`	`sqrt (y)`	**Square root.**
`y =`	`log (y)`	**Natural logarithm.**
`x =`	`log10 (x)`	**Common logarithm.**
`y =`	`exp (y)`	**Exponentiation.**

The allowed datatypes of the arguments and return values are:

`w =` `integer`, `real`, or `double`
`x =` `real` or `double`
`y =` `real`, `double`, or `complex`
`z =` `integer`, `real`, `double`, or `complex`

Do not use `short` or `long` integer arguments with any functions other than the type conversion intrinsics. The datatypes must match for functions with more than one argument. The datatype returned by the functions is the same as the arguments, except for type conversions or the absolute value of a `complex` number. SPP performs the “normal” automatic type conversions in expressions.

6.10. Other Interfaces and Library Routines

The short answer is that these are outside the scope of this document.

The slightly longer answer is that the IRAF VOS includes numerous other interfaces for performing a very wide variety of chores. Many of these interfaces are available within the normal IRAF system libraries and may be called from within your programs with no further ado. Many others are contained within specific libraries that may be simply referenced on the command line when your executables are linked. The math libraries fall under this latter category, for instance. The final category is the large body of example code that is available as source code, although not archived in any particular object library.

All of these are likely to change (grow, that is, IRAF interfaces are not obsoleted without some fundamental reason) from one release of IRAF to the next. A list of these other VOS interfaces currently includes:

`fmio`	**Binary file manager (typically internal to other interfaces)**
`gty`	**Generalized termcap style database reader**
`imfort`	**Fortran and C callable subset of IRAF data access routines**
`ki`	**IRAF networking kernel interface (internal to the VOS)**
`libc`	**Special purpose binding of certain VOS routines for the CL**
`mtio`	**Magnetic tape I/O**
`mwcs`	**World coordinate system support**
`plio`	**Pixel list I/O**
`pmio`	**Pixel lists tied to image I/O (masks)**
`qpoe`	**Position ordered event support (photon event counting)**
`symtab`	**Symbol table management**
`tty`	**Terminal control**

A list of the math libraries follows. You will have to decide whether any of the particular numerical techniques involved are appropriate to your data. The command line argument that you will need to specify to `xc` when you link your program is shown in the final column.

`bevington`	**Routines from** *Data Reduction and Error Analysis...*	*-lbev*
`curfit`	**Linear least squares curve fitting**	*-lcurfit*
`deboor`	**Routines from** *A Practical Guide to Splines*	*-ldeboor*
`gsurfit`	**Linear least squares surface fitting**	*-lgsurfit*
`iminterp`	**Image interpolation (typically internal to IMIO)**	*-liminterp*
`interp`	**Ditto**	*-linterp*
`llsq`	**Singular Value Decomposition**	*-lllsq*
`nlfit`	**Nonlinear least squares fitting (Levenberg-Marquardt)**	*-lnlfit*
`surfit`	**Fit gridded surface data**	*-lsurfit*

The *pkg$xtools* directory contains many useful routines that are scattered about its main directory and several subdirectories. These include `icfit`, for graphical linear least squares fitting, `gtools`, to provide useful graphing options to your cursor loop tasks, and `ranges`, that provides the bookkeeping for sorting through hyphenated lists of numerically indexed files or, in an alternate version, of pixels.

Rather than simply peruse this list that will be outdated very quickly, you are encouraged to examine example code from the IRAF directory tree that performs a similar chore to what is desired. Example scientific and system application tasks will be found scattered about in the subdirectories of *noao$* and *pkg$* (*iraf$noao/* and *iraf$pkg/*). Source code for the VOS system libraries is found in the subdirectories of *sys$* (*iraf$sys/*). The IRAF math libraries are defined in the subdirectories of *math$* (*iraf$math/*).

A map of the IRAF directory tree is available from the *iraf/docs* directory of the IRAF anonymous ftp archive on *iraf.noao.edu*. There are several versions available depending on the host computer and the version of IRAF. You most likely want the file *d210sun.ps.Z*.

6.11. Arguments & Return Values

VOS procedures are particular about the datatype of the arguments with which they are called. There are a few general rules of thumb for specifying these arguments, but with the huge number of VOS routines that are available it is likely that you will find some exceptions to any enumerated set of rules. The only way to be sure of the usage is to either examine some piece of code that uses a particular routine that is known to work correctly, or alternately to examine the source code for the VOS routine itself.

- In general, file descriptors (as returned by `open`) are integers while image descriptors (as returned by `immap`) are pointers. Most other descriptors (*e.g.*, graphics) are also pointers.
- Routines that write to output string arguments should always have another integer argument that specifies the maximum length of the string.
- Few VOS routines have boolean arguments. Instead they rely on integer arguments that encode the boolean value as the predefined macro values `YES` or `NO`. Always use these macro names in expressions, never use the explicit values (of 1 and 0).
- Short integer arguments must be honored if your code is to be portable. Byte swapped machines are likely to choke on long integers specified for such arguments. Luckily there aren't very many instances where integer arguments to a VOS procedure are specified as `short`, rather than `int` or `long`.
- In general, you must distinquish between `short`, `int`, and `long`, and between `real` and `double`. Often the safest way to do so, assuming that your program uses a local variable of a different integer or floating point type, is to declare an extra variable of the correct type and to assign the value of inappropriately typed variable to this variable using the type conversion intrinsic functions. This should be done as a separate step for the highest reliability across the vaguaries of host dependent compilers.
- It can be tricky to specify data constants of the correct integer or floating point data type in an argument list. This is a characteristic that SPP inherits from Fortran and is dependent on the particular host compiler. For maximum portability it is safest to only pass variables in argument lists rather than explicit constant values, especially of the more infrequently used types such as `short`. This is obviously critical for output arguments that will be modified by the procedure.

7. Making an IRAF Package

An IRAF *package* is a collection of tasks that are in some way related. For instance, an IRAF *core system* package, such as the **images**, **dataio**, or **plot** packages, typically contains tasks that serve as generic tools for handling such things as IRAF images, tapes and outside data formats, or hardcopy and terminal graphics. On the other hand, a typical package from the **noao** umbrella of packages collects together tasks that address a more specific data reduction need, *e.g.*, the **ccdred** or **apphot** packages that provide support for pipeline CCD reductions or for numerical aperture photometry, respectively.

You may need such a cohesive package, or you may just have a number of homegrown tasks that you would like to add to IRAF. In either case, assembling an IRAF package is a good way to proceed, both for easy access by your users (perhaps only yourself), and also for easy maintenence by your programmers (perhaps only yourself).

To make creating a package easier, a *tar* file of the **examples** package that contains all of the examples from this document is available by anonymous ftp from the IRAF network archive. You may find the canned **examples** package useful both to provide example programs to refer to while reading this document, and also as a straightforward platform to which to add your own tasks. The particular examples were written to provide hooks to support the more frequent sorts of things that users are likely to want to do within IRAF. Moreover, a small library of useful plotting and astronomical algorithms has been included that may be referenced from within your own tasks.

7.1. Installing the "examples" Package

The instructions for retrieving the **examples** package archive over the network and for unpacking this archive into your own work directory are of necessity host machine dependent. Rather than attempting to describe all possible host specific scenarios in adequate detail, the particular instructions given below are for SunOS v4.*N*. These should apply to most varieties of Unix, particularly Berkeley dialects. You should be able to extrapolate to your situation.

The following is a typical anonymous **ftp** session to retrieve the **examples** package archive file. The short retrieval instructions for the experienced Internet user are that the archive is contained in the compressed tar file *examples.tar.Z* in the directory *iraf.old* accessable by binary anonymous **ftp** on the host *iraf.noao.edu* (which has the Internet address *140.252.1.1*). Note that a compressed PostScript version of the document you are reading is located in the *iraf/docs* directory in the file *sppguide.ps.Z*. The first step should be to create an empty directory on your own machine in which to place the package files:

```
% cd someplace_convenient
% mkdir examples
% ftp iraf.noao.edu
Connected to tucana.noao.edu.
220 tucana FTP server (SunOS, SAG's 28Mar89 version) ready.
Name (iraf.noao.edu:seaman): anonymous
331 Guest login ok, send ident as password.
Password: <seaman@noao.edu>
230 Guest login ok, access restrictions apply.
ftp> verbose
Verbose mode off.
ftp> cd iraf.old
ftp> get readme.examples
ftp> binary
ftp> get examples.tar.Z
ftp> quit
```

The angle brackets (<>) in the reply to the password prompt are meant to imply that the identifying password will not be echoed on your screen. The file *readme.examples* contains a variation of the instructions that you are currently reading.

If your local machine does not support anonymous ftp, alternate arrangements can be made. Please contact IRAF site support for help. The email addresses are *iraf@noao.edu* on the Internet and also (via a gateway) on Bitnet, *5355::IRAF* on SPAN , and *uunet!noao.edu!iraf* via UUCP. The phone number for the IRAF hotline is *602-323-4160.*

Assuming that you now have the distribution archive of the **examples** package, *examples.tar.Z,* on your local machine, the files can be unpacked with commands such as:

```
% cd examples
% zcat ../examples.tar.Z | tar -xpf -
```

You might also have decided to download the archive file directly into the *examples* directory (or conversely, into */tmp*), but placing it in the parent directory avoids having the file clutter up the directory listings illustrated below and also makes recreating the package simpler if you want to start over. In any case, unpacking the archive should only take a few moments. Since the **examples** package is small, you can specify `-v` to the **tar** command if you want to see how the unpacking progresses.

The **examples** package was archived without any binary executable or object files. IRAF external packages from programming groups both inside and outside of NOAO are typically not distributed with binary files for any computer architecture. This is in contrast to the IRAF core system and **noao** packages which are supported directly by the IRAF group and which are distributed with pretested binaries for various host systems. In any case, if your computer is not configured for IRAF (or host) software development, it is best that you find out now while trying to build **examples**.

A single copy of IRAF can support multiple host system architectures, for instance, a Sun 4 fileserver may support a number of Sun 3 diskless nodes with an NFS mounted copy of IRAF. This applies to IRAF external packages as well as to the core system of IRAF. The **examples** package is distributed with hooks for supporting the various types of Sun computer architectures and floating point options, specifically *Sparc*, *i386*, *f68881*, and *ffpa* (the last two are floating point options for the Sun 3 *mc68020* architecture). The following instructions will assume a Sparc host system, but the steps would be similar for any of the other architectures.

At this point, a directory listing should show these files and subdirectories:

```
% ls
bin             bin.sparc       examples.hd     lib
bin.f68881      doc             examples.men    mkpkg
bin.generic     examples.cl     examples.par    src
```

If your directory does not look like this, you will have to determine whether the **ftp** download or the **zcat** unpacking are at fault before continuing.

To standardize the remainder of the commands for this section and to avoid any distracting host dependent commands, we will get into the IRAF CL before continuing. First you will need to utter a few magic incantations to get the subsequent commands to work properly:

```
% setenv iraf iraf_path/
% source $iraf/unix/hlib/irafuser.csh
```

Where `iraf_path` should be substituted with the full Unix pathname of the root directory of the IRAF directory tree. **Do not forget the trailing slash, “/”, on the definition of “iraf”.** IRAF, itself, is smart enough to figure out such site specific information as where it lives on your computer’s filesystem. Recall, however, that the IRAF compiler and other programming tools are host level programs. On a typical Unix system, the two commands above serve to supply sufficient information to the programming tools to let them do their job. The C-shell script *irafuser.csh* contains a number of site specific environment variable declarations. Conversely, on a VMS system this information was made globally available as part of the IRAF system installation by your system manager. In either case, this detailed information is only needed for IRAF software development, not for simply running IRAF. It is most convenient to put the two commands above into your *.login* file so that they will be automatically executed the next time you log onto the computer. Folks who aren't running the C-shell will need to make alternate arrangements.

One final bit of magic specific to the **examples** package is:

```
% setenv examples examples_path/
```

Where `examples_path` should be substituted with the Unix pathname of the directory into which the package was unpacked. Again, do not forget the trailing "`/`". The reasoning behind this magic will be revealed a little later. At this point you should get into IRAF in whatever way you usually do so. The rest of us will wait for you...

...ok, are you now waiting with bated breath at a `cl>` prompt? Let's continue. One side affect of the magic **setenv** command is to provide an IRAF logical directory to which you can now **cd**:

```
cl> cd examples
```

The IRAF **mkpkg** facility is used to maintain all but the simplest of SPP programs as described in §4. We will make use of some of the standard capabilities of **mkpkg**. First, we should verify for which of the available software development architectures the package happens to be configured:

```
cl> mkpkg arch
system is configured for generic
```

The `generic` architecture is correct for any external package that is not currently being worked on, and certainly for any external package fresh from an archive. We are building the package under the Sparc architecture. The command to configure the package for this is:

```
cl> mkpkg sparc
delete any dreg .e files left lying about in the source directories
archive and delete generic objects
restore archived sparc objects
no object archive found; full sysgen will be needed
```

The meaning of these messages will be discussed below, but if you see a message such as:

```
Warning: IRAFARCH is still set in your environment to f68881
```

you are running IRAF on a non-Sparc computer (in this case, a Sun 3) and will either have to log off and find a different computer, or will have to substitute `f68881` (or whatever) for `sparc` wherever necessary. A few additional hints for building the package on a non-Sun computer are included at the end of this section (§7.1).

You should now be ready to rebuild the entire package by simply typing:

```
cl> mkpkg -p examples >& SPOOL &
[1]
```

The output and error messages are redirected into the file *SPOOL* for later review. Even a small IRAF package such as **examples** may generate hundreds of lines of messages as it is being compiled, linked, and installed. We put the command into the background so that you can go about your business for the few moments that the package will take to compile and link.

A handy Unix trick that you may want to use to monitor the progress of the **mkpkg** is to **tail** the spool file as it grows:

```
cl> !tail -f SPOOL
```

This trick will grow in utility as your package grows. You can escape from the **tail** command at any point by typing `^C` (`control-C`).

When **mkpkg** is finished you should see a message similar to: `[1] done  60.2 2:20 42%`. At this point you should review the summary of the spooled **mkpkg** and compiler messages:

```
cl> mkpkg summary spool=SPOOL
warning: library 'libpkg.a' not found
ranlib libpkg.a
Updated 20 files in libpkg.a
move 'xx_examples.e' to 'examplesbin$x_examples.e'
```

This is a rather Spartan summary in which messages matching a variety of character templates are simply discarded as uninteresting. In this case, the `warning` message merely indicates that you are rebuilding the package from a version that has been stripped of binaries and so there is no previous copy of the package library file, *libpkg.a*, to search for outdated compiled object files. The `ranlib` message is a bit of host dependent magic that you should never have to think about. We are next told that 20 files have been updated (or simply added) in the package library file. These files were scrounged from the *src* directory and from its *asttools* and *airmass* subdirectories. The final message tells us that the package executable has been installed in a package *bin* directory, specifically *bin.sparc* since we earlier executed a **`mkpkg sparc`**.

If any of these steps malfunctioned for whatever reason, you would see an error message (not just a `warning`). If any error messages appear that you don't understand, you should first review the full spool file to see if that sheds any light on the problem:

```
cl> page SPOOL
```

The next step in tracking down problems in building the package would be to run through the directions in this section from the top, carefully verifying each. If that does not help then you (or your system manager) need to consider whether there is some problem with the host system compilers or libraries on the computer. For instance, does the computer even have Fortran and/or C compilers?

Assuming you made it past the **mkpkg**, you are almost done. All that remains is to grant yourself access to the newly retrieved, unpacked, and compiled package. This only requires that you define the package script as an IRAF task:

```
cl> task examples = examples$examples.cl
```

At this point, you can now load the **examples** package:

```
cl> examples
      fibonnaci   hello       impix       imreplace1  imreplace2  pairmass
```

and execute a task:

```
ex> hello
Hello, world!
```

and do anything with the package (almost) that you can do with other packages and tasks:

```
ex> lpar imreplace2
        input =                 List of input images
       output =                 List of output images
       (value = 0.)             Constant for the replacement operation
       (lower = INDEF)          Lower limit of the replacement window
       (upper = INDEF)          Upper limit of the replacement window
        (mode = "ql")
```

At this point you are all set to run the **examples** package, at least until you log out and back in again. You still need to define the package in such a way that it is automatically available when you log into IRAF. There are two ways to do this. The first way is to edit the task statement for the package into your *login.cl*, or better yet, your *loginuser.cl* file. The other requirement for this to work is to define `examples` in the IRAF environment, not in the Unix environment as you did in the bit of magic above:

```
set examples = examples_path/
task examples = examples$examples.cl
```

Note that *`examples_path`* can now be specified as an IRAF virtual pathname. The two commands should be placed before the `keep` statement in either IRAF *login* file. When you next log into IRAF, the **examples** package should be available to you.

It was indicated above that **examples** was *almost* ready to be used in all normal IRAF ways. The one missing piece is access to the package's help pages. This brings us to the second, politically correct, way to install an external package into IRAF. The connection between the main IRAF system and each of a particular site's external packages is contained in the file *hlib$extern.pkg*. All knowledge necessary to access the external packages is limited to that file. This consists of the two items that you were told to add to your *loginuser.cl* file (the `examples` environment variable and the `task` statement), but also the hook that allows the IRAF help command to find your new package's help pages. A more fundamental reason to install your package this way is that it allows other folks to use the package, too.

The template *extern.pkg* from the *hlib$* directory of a fresh IRAF installation looks like:

```
# External (non core-system) packages.  To install a new package, add the
# two statements to define the package root directory and package task,
# then add the package helpdb to the 'helpdb' list.

reset   noao            = iraf$noao/
task    noao.pkg        = noao$noao.cl

#reset  local           = iraf$local/
#task   local.pkg       = local$local.cl

reset   helpdb          = "lib$helpdb.mip\
                          ,noao$lib/helpdb.mip\
#                         ,local$lib/helpdb.mip\
                          "
keep
```

The commented out statements for the **local** package are meant to serve as an example when adding hooks for a new external package.

Your site's IRAF system manager can simply copy and edit the lines to fully install the **examples** package for all users:

```
# External (non core-system) packages.  To install a new package, add the
# two statements to define the package root directory and package task,
# then add the package helpdb to the 'helpdb' list.

reset   noao            = iraf$noao/
task    noao.pkg        = noao$noao.cl

reset   examples        = examples_path
task    examples.pkg    = examples$examples.cl

#reset  local           = iraf$local/
#task   local.pkg       = local$local.cl

reset   helpdb          = "lib$helpdb.mip\
                          ,noao$lib/helpdb.mip\
                          ,examples$lib/helpdb.mip\
#                         ,local$lib/helpdb.mip\
                          "
keep
```

This then is the explanation for the `setenv examples` magic with which we conjured above. We needed to enter the `examples` variable into the Unix environment, not just into the IRAF environment, to define it for **mkpkg**, not for running the package. However, by defining the variable in *extern.pkg*, we have solved both problems at the same time.

One last bit of business will conclude building the Sparc version of the package's binaries:

```
cl> mkpkg generic
delete any dreg .e files left lying about in the source directories
archive and delete sparc objects
compress bin.sparc/OBJS.arc [1] 12697
restore archived generic objects
no object archive found; full sysgen will be needed
Warning: IRAFARCH is still set in your environment to sparc
```

The basic result of all this is to confine all the architecture dependent files from the Sparc software development to the *bin.sparc* directory. The initial `delete any dreg .e` message tells you that if a previous attempt to link the package failed for some reason, any resulting garbage executable files will be cleaned up. Another way to leave an executable laying about is to execute **mkpkg** from other than the root directory of an external package, for instance typing `mkpkg -p examples` in the *examples$src* subdirectory will compile any out-of-date source files and relink the package, but will not install the executable in the appropriate *bin* directory, but rather leave it in the *src* directory under the name *xx_examples.e.*

The `archive and delete` and `compress` messages indicate that all of the Sparc architecture libraries and individual object files will be archived into a compressed **tar** format file names *OBJS.arc.Z* in the *bin.sparc* directory. This will be unpacked by **mkpkg** the next time you need to update the package for your Sparc machines.

The `restore archived generic objects` and `no object archive found` messages just represent the whole point of the `generic` pseudo-architecture to provide a neutral configuration for external packages. The final `Warning` message is meant to alert you that the IRAF and the host system are not currently configured to build the newly selected architecture. This does not mean anything for the `generic` case, but is a very important issue if you are building multiple versions of the package on the same computer.

Finally, you were promised some hints for installing the package on computers other than Suns. Most of the host system dependencies on software development are hidden within IRAF, which is after all one of the reasons you are likely to be interested in programming in SPP. Details such as the libraries that are required by the host compilers and the host's particular floating point format are largely hidden from you, and those that aren't are parametrized in a machine dependent header file such as *hlib$mach.h* just in case you need to know. You will, however, need to keep track of the specific architectures of the machines on your local network. If you have a fileserver that supports IRAF on client machines with varying architectures, you will need to have an appropriately named *bin* subdirectory for each architecture. You will also need to modify the file *examples$mkpkg* to include a "module" for each architecture. Entries supporting software development on an Ultrix DECstation and a Mips might be:

```
dsux:                                   # DECstation/Ultrix binaries
        $verbose off
        $set DIRS = "lib src"
        !$(hlib)/mkfloat.csh dsux -d $(DIRS)
        ;
mips:                                   # install mips binaries
        $verbose off
        $set DIRS = "lib src"
        !$(hlib)/mkfloat.csh mips -d $(DIRS)
        ;
```

The corresponding *bin* directories would be *bin.dsux* and *bin.mips*. It is your responsibility to guarantee that the host compiler knows for which architecture you are compiling the package. On a Sun 3, for instance, you may need to set `FLOAT_OPTION` in your environment to distinguish between the 68881 and FPA floating point options.

7.2. Adding a Task to the Package

Now that you have acquired your own copy of the **examples** package, we will discuss how to add your own tasks to the package. Let's say that you want to add the task **howdy** to the package. The SPP source code for the task is contained in the file, *t_howdy.x*, and consists of the following variation on the "Hello, world!" theme:

```
# HOWDY -- Variation on HELLO.

procedure t_howdy ()

pointer howdy, sp

begin
        call smark (sp)
        call salloc (howdy, SZ_LINE, TY_CHAR)

        call clgstr ("howdy", Memc[howdy], SZ_LINE)
        call printf ("%s\n")
            call pargstr (Memc[howdy])

        call sfree (sp)
end
```

Howdy will prompt the user for the string value of the `howdy` parameter (unless it is specified on the command line) and will then print that string out. The task has a corresponding one line parameter file, *howdy.par*:

```
howdy,s,a,,,,Text string to be echoed
```

The first step is to copy these two files into the *examples$src/* directory.

Next a few files must be edited to fully install the task. Let's finish with the *src/* directory before moving on. First the SPP `task` statement must be edited into the file *x_examples.x* and the mkpkg file must be modified to compile and link the object code for *t_howdy.x*. Determining the precise changes that are needed is left as an exercise for the reader. These two changes take care of actually linking the new SPP task into the package executable.

There are potentially four files that you may need to change in the *examples$* directory itself. These are *examples.cl* to add the CL task statement declaring the task for the user, *examples.par* to update the version timestamp for the package, *examples.hd* and *examples.men* to add a help file for the task. Writing a help page is always a good idea. The **mkmanpage** task in the **softools** package will provide a blank help page for you to edit. The mkhelpdb task compiles the information in the *helpdb.hd* file into the machine independent binary file, *helpdb.mip*:

```
cl> cd examples
cl> edit helpdb.hd
cl> edit helpdb.men
cl> softools
so> cd doc
so> mkmanpage softools
so> cd ../lib
so> mkhelpdb root.hd helpdb.mip
```

To actually update the package's binary executable, a sequence of commands such as the following will be needed. This assumes that the package has been installed in the file *hlib$extern.pkg*.

```
cl> cd examples
cl> mkpkg sparc
cl> mkpkg -p examples
cl> mkpkg generic
```

More complicated tasks may, of course, have many more files associated with them such *.h* header files, cursor keystroke help files, and multiple source code files. As general advice, if a task requires more than a couple of such files then you should create a subdirectory to contain the selection of files for the task. Consult the instructions in §5 and in §7.1 for more hints.

Appendix I

Topics Not Discussed

There are some topics that just would not fit in easily anywhere else:

- IRAF tasks are not restricted to the Command language, but can be called from the host operating system (shell) prompt. The SPP `task` statement in the source code (recall that this is replaced by the automatically generated code for the `sys_runtask` procedure that you see when your program is linked) supplies a simple command interpreter for situations in which an IRAF executable is run at the host level.

  Further discussion is outside the scope of this document, but some special usage situations may benefit from this option. Contact IRAF site support (`iraf@noao.edu`) for information.

- SPP debugging aids are a field of current activity. Mike Fitzpatrick's package of programming tools is available in the IRAF **ftp** archive. Installation instructions are in the file *readme.spptools*.

```
% ftp iraf.noao.edu
Connected to tucana.noao.edu.
220 tucana FTP server (SunOS, SAG's 28Mar89 version) ready.
Name (iraf.noao.edu:seaman): anonymous
331 Guest login ok, send ident as password.
Password: <seaman@noao.edu>
230 Guest login ok, access restrictions apply.
ftp> verbose
Verbose mode off.
ftp> cd iraf.old
ftp> get readme.spptools
ftp> binary
ftp> get spptools.tar.Z
ftp> quit
```

- IRAF variables are static variables. Their value is remembered from one execution of a procedure to the next. It is not the safest programming practice to rely on this behavior, but you may find the characteristic useful on occasion.

Appendix II

References

There are a number of references available that you may want (or need) to consult while reading this document. The first of these is available from STScI via anonymous ftp to *stsci.edu* in the *spp* subdirectory of */software/stsdas/v1.2/doc/programmer*:

- *SPP Programmer's Reference*, edited by Zoltan G. Levay, 1991.

The remainder of the documents are available in the IRAF "Big Blue Books", mostly in volumes 3A and 3B. These are also available from the IRAF anonymous ftp archive on *iraf.noao.edu* in the *iraf/docs* directory, along with a large number of documents of every description that may also be interesting and useful to you. Note that the normal format for the archived files is as compressed PostScript. IRAF site support (`iraf@noao.edu`) can help you if you have trouble downloading the documents (remember to set binary ftp mode) or printing them out. The particular archive filename is listed at the end of each entry.

These documents provide access to other documents or to the IRAF directory tree:

- Table of Contents of the IRAF Newsletters. (*TOC_news.txt*)
- Table of Contents of the IRAF User and System Handbooks. (*TOC*.txt*)
- IRAF/UNIX and IRAF/VMS Directory Structure. (*d*sun.ps.Z*, *d*vms.ps.Z*)

The next documents provide general or specific information about SPP and the VOS:

- *IRAF Standards and Conventions*, Elwood Downey, *et. al.*, 1983. (*std.ps.Z*)
- *A Reference Manual for the IRAF Subset Preprocessor Language,* Doug Tody, 1983. (*spp.txt.Z*)
- *Programmer's Crib Sheet for the IRAF Program Interface,* Doug Tody, 1983. (*prog_crib.txt.Z*)
- *Named External Parameter Sets in the CL and related revisions,* Doug Tody, 1986. (*pset.ps.Z*)

Next are some alternatives to writing SPP programs:

- *A User's Guide to Fortran Programming in IRAF, The IMFORT Interface,* Doug Tody, 1987. (*imfort.ps.Z*)
- *Specifying Pixel Directories with IMFORT*, Doug Tody, 1989. (*imfortmem.ps.Z*)
- *An Introductory User's Guide to IRAF Scripts*, rev. by Rob Seaman, 1989. (*script.ps.Z*)

Various documents provide background information you may find illuminating:

- *The IRAF Data Reduction and Analysis System*, Doug Tody, 1986. (*iraf.ps.Z*)
- *The Role of the Preprocessor*, Doug Tody, 1982.
- *A Reference Manual for the IRAF System Interface*, Doug Tody, 1984. (*os.ps.Z*)
- *Graphics I/O Design*, Doug Tody, 1987. (*gio.txt.Z*)
- *IRAF IMIO Overview*, Doug Tody, 1986. (*imio_1.txt.Z*)
- *New Release of Image I/O, etc.*, Doug Tody, 1985. (*imio_2.ps.Z*)
- *The IRAF Image I/O Interface, Design Strategies, Status and Plans*, Doug Tody, 1983. (*imio_3.ps.Z*)
- *The Image Header*, Doug Tody. (*imio_4.txt.Z*)

For completeness, the document that you are reading is also available from the IRAF anonymous ftp archive on *iraf.noao.edu* in the file *sppguide.ps.Z* in the *iraf/docs* directory. The quick reference card is in the same directory in the file *quickref.ps.Z*. The *examples* package is archived in the *iraf.old* directory as the file *examples.tar.Z* with retrieval and installation directions in the file *readme.examples.*

NAME

hello -- Hello, world!

USAGE

hello

DESCRIPTION

HELLO is the familiar "Hello, world!" programming example coded in SPP. The source code for this task (i.e., an IRAF program) is:

```
# HELLO -- Simple program to introduce SPP programming basics.

task hello = t_hello_world

procedure t_hello_world ()

begin
        call printf ("Hello, world!\n")
end
```

EXAMPLES

To compile the source code, first save it into the file "hello.x" and then type:

```
cl> xc hello.x
```

This will produce the executable file "hello.e".

To declare the task to the CL, if "hello.e" is in the current directory:

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

In this case the HELLO task will only function in this particular directory. The initial dollar sign ($) indicates that this particular task does not have an associated parameter file.

To declare the task to the CL, if "hello.e" is in the directory "home$spp/":

```
cl> task $hello = home$spp/hello.e
```

HELLO should now work from all directories.

To run the task:

```
cl> hello
Hello, world!
```

Or load the examples package first to run the precompiled version:

```
cl> examples
ex> hello
Hello, world!
```

To run the task outside of the CL (Unix version):

```
% alias hello /u2/seaman/spp/hello.e hello
% hello
Hello, world!
```

To run the task outside of the CL (VMS version):

```
$ hello :== $usr$0:[seaman.spp]hello.exe;1 hello
$ hello
Hello, world!
```

Note that the latest version of "hello.exe", in this case "hello.exe;1" is known within IRAF as "hello.e" thanks to IRAF virtual filename mapping.

NAME

fibonnaci -- print the first N terms from the Fibonnaci sequence

USAGE

fibonnaci nterms

PARAMETERS

nterms

The number of terms to print from the Fibonnaci sequence.

DESCRIPTION

FIBONNACI will print the first N terms from the Fibonnaci sequence. It uses two different algorithms, one algebraic and one iterative, which may be compared for accuracy. The task outputs the actual Fibonnaci numbers, that is integers, with the result that it reaches the limit of integer precision on typical 32 bit workstations at about nterms=46. There is a hard maximum of 50 for this parameter.

EXAMPLES

To print the first dozen Fibonnaci numbers:

```
fibonnaci 12
```

NAME
imreplace1 -- replace pixels within lower and upper thresholds

USAGE
imreplace1 image value

PARAMETERS
image
The image to be modified.

value
The constant for the replacement operation.

lower
The lower limit of the replacement window.

upper
The upper limit of the replacement window.

DESCRIPTION
IMREPLACE1 is a simple version of the IRAF IMREPLACE task in the PROTO package. It will replace pixel values in a single image, which is modified in place. The lower and upper thresholds must be provided or the user will be prompted.

EXAMPLES
To replace all pixels in the image 'test', between 100 and 200 with the value 0:

```
imreplace1 test 0 lower=100 upper=200
```

SEE ALSO
imreplace2, impix

NAME
imreplace2 -- replace pixels within thresholds for a list of images

USAGE
imreplace input output

PARAMETERS
input
The list of input images.

output
The list of output images.

value=0.
The constant for the replacement operation.

lower=INDEF
The lower limit of the replacement window. The default value of INDEF indicates that the data minimum should be used as the limit. If **lower** is greater than **upper**, then values outside the window will be replaced.

upper=INDEF
The upper limit of the replacement window. The default value of INDEF indicates that the data maximum should be used as the limit. If **upper** is less than **lower**, then values outside the window will be replaced.

DESCRIPTION
IMREPLACE2 is an enhanced version (relative to IMREPLACE1) of the IRAF IMREPLACE task in the PROTO package. It will copy the list of **input** images to the list of **output** images, replacing pixel values that lie between **lower** and **upper**.

If **lower** or **upper** are specified as INDEF, the minimum or maximum (respectively) of the image will be used as the corresponding limit. If **lower** is greater than **upper** then the pixels outside of the specified window will be replaced

EXAMPLES
To replace all pixels between 100 and 200 in the input image 'test' with the value 0, outputing the changes to the image 'testout':

```
imreplace2 test testout lower=100 upper=200
```

To replace all pixels less than 0:

```
imreplace2 test testout upper=0
```

To replace all pixels greater than 1000 with the value 1000:

```
imreplace2 test testout value=1000 lower=1000
```

To replace all pixels either less than 0, or greater than 1000:

```
imreplace2 test testout lower=1000 upper=0
```

To replace pixels in a list of images. This requires that an *output* image template be specified:

```
imreplace2 *.imh out//*.imh
```
or
```
imreplace2 b*.imh %b%newb%*.imh
```
or
```
imreplace2 @inlist @outlist
```

SEE ALSO
imreplace1, impix

NAME

impix -- point at and edit pixels in an image

USAGE

impix image

PARAMETERS

image
The image to be edited.

peakup=yes
Peak up within the box?

localmin=no
Peak up on the local minimum, rather than maximum?

replace="median"
The replacement algorithm. The choices are "constant", "mean", and "median".

constant=0.
The value for the constant replacement algorithm.

boxsize=5
The size of the peaking and statistics box.

imcur=""
The image cursor.

frame=1
The frame number for the image display.

update=yes
Actually edit the image?

DESCRIPTION

IMPIX is a simplified version of IMEDIT that serves as an example of an IRAF cursor loop task. The task first displays the **image** and then enters the cursor loop that allows making modifications. The cursor keystrokes are summarized below. The various parameters can be adjusted and queried interactively.

KEYSTROKES

```
                    IMPIX Commands

               Cursor Keystroke Commands

 q  Exit the task.
 r  Redisplay the image.
 s  Report the statistics within the box
 z  Replace the pixel using the current algorithm
 ?  Get this help.


               Colon Commands

Issue a command with an argument to set the corresponding value,
or with no argument to print the current setting.  Commands and
arguments may be abbreviated.

 :eparam        Edit the parameters for DISPLAY
 :peakup        Peak up the raw cursor coordinates?
 :localmin      Peak up on the local minimum, rather than maximum
 :replace       Set the algorithm:  "constant", "mean", or "median"
 :constant      The value used for the constant replacement algorithm
 :boxsize       The size of the peaking and statistics box
```


EXAMPLES

To interactively edit an image, ‘test’:

```
impix test
```

To edit ‘test’ without centering the cursor:

```
impix test peak-
```

This is useful for zapping obviously bad pixels. The image display may be zoomed to make this easier.

SEE ALSO

imedit, imreplace1, imreplace2

NAME

pairmass -- plot the airmass throughout the day for a given object

USAGE

pairmass

PARAMETERS

ra
The right ascension of the object.

dec
The declination of the object.

epoch=INDEF
The epoch of the coordinates.

year
The year of the observation.

month
The month of the observation (a number from 1 to 12).

day
The day of the month of the observation.

longitude=111.6
The longitude of the observatory. The default is for Kitt Peak.

latitude=31.98
The latitude of the observatory. The default is for Kitt Peak.

resolution=4
The number of UT points per hour for which to calculate the airmass.

listout=no
List, rather than plot, the airmass versus the universal time?

PLOT PARAMETERS

wx1=0., wx2=0., wy1=0., wy2=5.
The range of window (user) coordinates to be included in the plot. If the range of values in x or y = 0, the plot is automatically scaled from the minimum to maximum data values along that axis. The maximum plotted Y value (airmass) is set to 5 by default.

pointmode = no
Plot individual points instead of a continuous line?

marker="box"
If **pointmode** = yes, the marker drawn at each point is set with this parameter. The acceptable choices are "point", "box", "plus", "cross", "circle", "hebar", "vebar", "hline", "vline", and "diamond".

szmarker = 0.005
The size of the marker drawn when **pointmode** = yes. A value of 0 (zero) indicates that the task should read the size from the input list.

logx = no, logy = no
Draw the x or y axis in log units, versus linear?

xlabel="Universal Time", ylabel="Airmass"
Labels for the X-axis and Y-axis.

title="default"
Title for plot. If not changed from "default", a title string consisting of the object's position, the date, and the observatory location is used.

vx1=0., vx2=0., vy1=0., vy2=0.
NDC coordinates (0-1) of the plotting device viewport. If not set by the user, a suitable viewport which allows sufficient room for all labels is used.

majrx=5, minrx=5, majry=5, minry=5
The number of major and minor divisions along the x or y axis.

round = no
Round axes up to nice values?

fill = yes
Fill the plotting viewport regardless of the device aspect ratio?

append = no
Append to an existing plot?

device="stdgraph"
Output device.

DESCRIPTION

The airmass is plotted through the course of a day for a given object from a given observatory. Various plotting options are supported. The plotting routine, as well as the astronomical utility routines are available for inclusion into a user's own SPP programs.

EXAMPLES

To plot the airmass for M82 from Kitt Peak for Groundhog's Day in 1992:

```
pairmass ra=9:51:42 dec=69:56 epoch=1950 year=1992 month=2 day=2
```

This is obviously a good opportunity to use the parameter editor.

SEE ALSO

airmass, setairmass, prows

An SPP/VOS Quick Reference Card

The short name for each interface is in parentheses following the title. The IRAF directory for the source code is the same as the short name. Pertinent system header files (located in *lib$* or *hlib$*) are listed at the end of the title line. The names given to the arguments are meant to reflect their usage (and thus their datatype), but you may still find that you need to consult the source code. Some routines can be located through the help database: `help error opt=so`. Others can be listed directly: `page etc$pagefiles.x`. Examples are scattered throughout IRAF and usage is described more fully in §6 of *An Introductory User's Guide to IRAF SPP Programming*.

Command Language Input/Output (*clio*)

	clgstr (param, outstr, maxch)	**Get a string parameter.**
	clpstr (param, string)	**Output a string parameter.**
value =	clget_ (param)	**Get a typed parameter.**
	clput_ (param, value)	**Output a typed parameter.**
stat =	clgcur (param, x,y,wcs, key,cmd,maxch)	**Read a cursor parameter.**
stat =	clgwrd (param, keyword, maxch, dict)	**Look up a parameter in a dictionary.**
list =	clpopni (param)	**Open a file template or the** STDIN.
list =	clpopns (param)	**Open a sorted template.**
list =	clpopnu (param)	**Open an unsorted template.**
	clpcls (list)	**Close a file template.**
nfiles =	clplen (list)	**Return the number of files in a list.**
stat =	clgfil (list, outstr, maxch)	**Get the next file name.**
	clcmdw (command)	**Send a command to the CL and wait.**

File Input/Output (*fio*) <fset.h>

fd =	open (fname, mode, type)	**Open a file for I/O.**
	close (fd)	**Close a file when finished.**
stat =	read (fd, buffer, maxch)	**Binary byte stream input.**
	write (fd, buffer, maxch)	**Binary byte stream output.**
	flush (fd)	**Flush the buffers immediately.**
stat =	access (fname, mode, type)	**Can the file be accessed?**
stat =	fstati (fd, param)	**Get the status of an open file.**
	fseti (fd, param, value)	**Set a FIO option.**
	delete (fname)	**Delete the named file.**
	rename (old_fname, new_fname)	**Rename a file.**
	mktemp (root, fname, maxch)	**Make a temporary file name .**
stat =	protect (fname, action)	**Protect or unprotect a file.**
stat =	getline (fd, linebuf)	**Get a line of text from a file.**
	putline (fd, linebuf)	**Output a line of text to a file.**

Image Input/Output (*imio*) <imhdr.h>, <imset.h>

im =	immap (image, mode, hdr_arg)	**Map ("open") an image.**
	imunmap (im)	**Unmap ("close") an image.**
	imflush (im)	**Flush the image buffers.**
buf =	imgl*N*_ (im [, line [, band]])	**Get a line from an image,** *N*=[123].
	impl*N*_ (im [,line [,band]])	**Output a line to an image,** *N*=[123].
buf =	imgs*N*_ (im, x1,x2 [,y1,y2 [,z1,z2]])	**Get a section from an image,** *N*=[123].
	imps*N*_ (im, x1,x2 [,y1,y2 [,z1,z2]])	**Output a section to an image,** *N*=[123].
stat =	imgnl_ (im, bufptr, v)	**Generalized get next line.**
	impnl_ (im, bufptr, v)	**Generalized output next line.**
stat =	imaccf (im, key)	**Does the named keyword exist?**
	imaddf (im, key, type)	**Add, but don't initialize, a keyword.**
	imadd_ (im, key, value)	**Add or modify a header keyword.**
	imastr (im, key, value)	**Add or modify a string keyword.**
	imdelf (im, key)	**Delete a header keyword.**
value =	imget_ (im, key)	**Get a keyword of the specified type.**
	imgstr (im, key, outstr, maxch)	**Get a string valued keyword.**
	imput_ (im, key, value)	**Modify an existing header keyword.**
	impstr (im, key, value)	**Modify an existing string keyword.**
code =	imgftype (im, key)	**What is the keyword's datatype?**
	imdelete (image)	**Delete an (unmapped) image.**
imt =	imtopenp (param)	**Open an image template.**
	imtclose (imt)	**Close an image template.**
stat =	imtgetim (imt, outstr, maxch)	**Get the next image name.**
stat =	imtrgetim (imt, index, outstr, maxch)	**Get a randomly indexed image name.**
nimages =	imtlen (imt)	**Return the number of images in a list.**
	imtrew (imt)	**Rewind an image list.**

Memory Management (*memio*)

	smark (sp)	**Save the current stack pointer.**
	salloc (ptr, nelem, type)	**Allocate space on the stack.**
	sfree (sp)	**Pop the stack.**
	malloc (ptr, nelem, type)	**Allocate space on the heap.**
	calloc (ptr, nelem, type)	**Allocate and zero space.**
	realloc (ptr, nelem, type)	**Adjust the size of a buffer.**
	mfree (ptr, type)	**Free space on the heap.**

Graphics Input/Output (*gio*) <gset.h>

gp =	gopen (device, mode, fd)	**Open a graphics stream.**
	gclose (gp)	**Close a graphics stream.**
	gflush (gp)	**Flush the graphics output.**
	gline (gp, x1, y1, x2, y2)	**Draw a line from** (x1,y1) **to** (x2,y2).
	gpline (gp, xa, ya, npts)	**Draw a polyline.**
	gmark (gp, x, y, type, xs, ys)	**Draw a marker of a given size.**
	gpmark (gp, xa, ya, npts, type, xs, ys)	**Draw a polymarker.**
	gamove (gp, x, y)	**Move the pen to the absolute position.**
	gadraw (gp, x, y)	**Draw (absolute) from the current position.**
	gseti (gp, param, value)	**Set a GIO option.**
	gswind (gp, x1, x2, y1, y2)	**Set the window in the world coordinates.**
	gsview (gp, x1, x2, y1, y2)	**Set the viewport in normalized device coordinates.**
	gascale (gp, array, npts, axis)	**Scale the axis to fit the data.**
	glabax (gp, title, xlabel, ylabel)	**Draw and label the axes.**
	gpagefile (gp, file, prompt)	**Page a file from graphics cursor mode.**

Vector Operators (*vops*)

	amov_ (a, b, npix)	**Copy a vector.**
	amovk_ (k, b, npix)	**Copy a constant into a vector.**
	aabs_ (a, b, npix)	**Absolute value of a vector.**
	aadd_ (a, b, c, npix)	**Vector** c **is the sum of vectors** a **and** b.
	aaddk (a, b, c, npix)	**Vector** c **is the sum of vector** a **and scalar** b.
	aavg_ (a, npix, mean, sigma)	**Mean and sigma of a vector.**
value =	amed_ (a, npix)	**Return the median of a vector.**
	abav_ (a, b, nblocks, blocksize)	**Block average of a vector.**
	abeq_ (a, b, c, npix)	c[i]=1 **if** a[i] == b[i], **else** c[i]=0.
	alim_ (a, npix, minval, maxval)	**Compute the min and max of a vector.**

Miscellaneous (*etc*)

	error (code, message)	**Generate an error action (may be trapped).**
	erract (severity)	**Take an error action.**
len =	envgets (key, value, maxch)	**Fetch the string value of an environment variable.**
value =	envget_ (key)	**Return the typed value of an environment variable.**
	qsort (array, nelems, compare)	**Sort an integer array by the function** compare ().
	gqsort (array, nelems, compare, arg)	**Sort by a function with an argument.**
	pagefiles (files)	**Page text file(s) on the STDOUT.**
	pagefile (file, prompt)	**Page a text file on the STDOUT.**
real =	urand (seed)	**Uniform "random" number in the interval (0,1).**
	sysid (outstr, maxch)	**Return a system and user identification string.**
int =	btoi (bool)	**Convert a boolean to an integer (**YES **or** NO**).**

Formatted Input/Output (*fmtio*) <chars.h>, <ctype.h>

	printf (format)	**Begin a print to the STDOUT.**
	eprintf (format)	**Begin a print to the STDERR.**
	fprintf (fd, format)	**Begin a print to a file.**
	sprintf (outstr, maxch, format)	**Begin a print to a string.**
	clprintf (param, format)	**Begin a print to a CL parameter.**
	parg_ (value)	**Complete a typed format.**
	pargstr (string)	**Complete a string format.**
stat =	scan ()	**Begin a scan from the STDIN.**
stat =	fscan (fd)	**Begin a scan from file.**
stat =	sscan (str)	**Begin a scan from a string.**
stat =	clscan (param)	**Begin a scan from a CL parameter.**
	garg_ (value)	**Get a typed value.**
	gargstr (outstr, maxch)	**Get the rest of the line.**
	gargwrd (outstr, maxch)	**Get the next "word".**
stat =	strdic (input, keyword, maxch, dict)	**Look up a string in a dictionary.**
bool =	streq (string1, string2)	**Compare two strings for equality.**
	strcpy (string1, string2, maxch)	**Copy string1 to string2.**

Format Specifications

An SPP format specification has the form "%*w*.*dC*", where *w* is the field width, *d* is the number of decimal places or the number of digits of precision, and *C* is the format code. The *w* and *d* fields are optional. The format codes *C* are as follows:

b	**boolean (YES or NO)**
c	**single character (c, \c, or \nnn)**
d	**decimal integer**
e	**exponential format, *d* is the precision**
f	**fixed format, *d* is the number of decimal places**
g	**general format, *d* is the precision**
h	***hms* format (hh:mm:ss.ss, *d* is the number of decimal places)**
H	***HMS* format, convert from degrees to hours first (divide by 15)**
m	***ms* or *hs* format (mm:ss.ss), *d* is the number of decimal places**
M	***MS* or *HS* format, convert from degrees to hours first**
o	**octal integer**
r*N*	**convert integer to or from radix *N***
s	**string, *d* is the maximum number of chars to print**
t	**advance to column given by *w***
u	**unsigned decimal integer**
w	**output the number of spaces given by *w***
x	**hexadecimal integer**
z	**complex format ((r,r)), *d* is the precision**
*	**deferred, get the field from the next parg_ call**

Conventions for specifying the field width:

w = n	**right justify and blank fill in a field of n characters**
w = -n	**left justify and blank fill in a field of n characters**
w = 0n	**right justify and zero fill in a field of n characters**
w = 0	**use as much space as needed**
absent	**same as *w* = 0**

Escape sequences for string literals and character constants:

\b	**backspace**
\f	**form feed**
\n	**newline (<CR><LF>)**
\r	**carriage return**
\t	**tab**
\"	**string delimiter character**
\'	**character constant delimiter character**
\\	**backslash character**
\nnn	**octal value of character**

Any combination of the fields *w*, *d*, *C*, or *N* may be specified as asterisks (*) in the format string, allowing the field to be passed at run time in a parg_ call.

SPP Intrinsic Functions

Type Conversions

char =	char (integer)	**Convert to character.**
short =	short (z)	**Convert to short.**
int =	int (z)	**Truncate to integer.**
int =	nint (x)	**Round to integer.**
long =	long (z)	**Convert to long integer.**
real =	real (z)	**Convert to real.**
real =	aimag (complex)	**Imaginary part.**
double =	double (z)	**Convert to double precision.**
complex =	complex (z)	**Convert to complex.**

Trigonometry (angles are in radians)

y =	sin (y)	**Sine.**
y =	cos (y)	**Cosine.**
x =	tan (x)	**Tangent.**
x =	asin (x)	**Inverse sine.**
x =	acos (x)	**Inverse cosine.**
x =	atan (x)	**Inverse tangent.**
x =	atan2 (x1, x2)	**Inverse tangent of x1/x2.**
x =	sinh (x)	**Hyperbolic sine.**
x =	cosh (x)	**Hyperbolic cosine.**
x =	tanh (x)	**Hyperbolic tangent.**

Miscellaneous

w =	abs (z)	**Absolute value.**
complex =	conjug (complex)	**Complex conjugate.**
w =	min (w1, w2, ...)	**Minimum value.**
w =	max (w1, w2, ...)	**Maximum value.**
w =	mod (w1, w2)	**Remainder after w1/w2.**
y =	sqrt (y)	**Square root.**
y =	log (y)	**Natural logarithm.**
x =	log10 (x)	**Common logarithm.**
y =	exp (y)	**Exponentiation.**

The allowed datatypes of the arguments and returned values are:

```
w = integer, real, or double
x =          real  or double
y =          real,    double, or complex
z = integer, real,    double, or complex
```

Do not use short or long integer arguments with any functions other than the type conversion intrinsics. The datatypes must match for functions with more than one argument. The datatype returned by the functions is the same as the arguments, except for type conversions or the absolute value of a complex number. SPP performs the "normal" automatic type conversions in expressions.

National Optical Astronomy Observatories
P.O. Box 26732, Tucson, AZ 85726-6732