DRAFT

A User's Introduction to the IRAF Command Language
Version 2.3

PETER MB SHAMES
Space Telescope Science Institute

DOUG TODY
National Optical Astronomy Observatories

Revised – August 11, 1986

ABSTRACT

This tutorial introduction to the IRAF Command Language presents an overview of the use and features of the language. The discussion is aimed toward the first-time user and describes the execution of tasks from the Command Language. The focus is the Command Language itself; the many packages and tasks that compose the IRAF system and the SDAS packages from STScI are described elsewhere. The emphasis is on using the Command Language to run existing programs, although sections are included that describe the addition of new tasks of one's own making. A quick guide to language features and facilities and to the suite of reduction and analysis packages currently available is provided in the Appendices.

About the Authors

Peter Shames is Chief of the Systems Branch at STScI, and along with Jim Rose and Ethan Schreier, was one of the key persons responsible for the selection of IRAF as the command language and operating environment for the STScI Science Data Analysis System (SDAS) in December of 1983. Since that time, Peter has supervised the VMS/IRAF development effort at STScI, overseeing the implementation of the VMS/IRAF kernel, the initial port of IRAF to VMS, and the development of version 2.0 of the IRAF command language. Peter wrote the original CL User's Guide (version 2.0).

Doug Tody is the originator and designer of the IRAF system (including the CL) and has been Chief Programmer of the IRAF project since the inception of the project at KPNO (now NOAO) in the fall of 1981. As Chief Programmer, Doug has written virtually all of the IRAF system software with the exception of the VMS/IRAF kernel and the original CL 1.0 (which was written by Elwood Downey). Since 1983 Doug has been head of the IRAF group at NOAO, overseeing the development of the NOAO science applications software while continuing work on the IRAF systems software, and coordinating the effort with STScI.

Acknowledgements

The authors wish to acknowledge the efforts of the many people who have contributed so much time, energy, thought and support to the development of the IRAF system. Foremost among these are the members of the IRAF development group at NOAO (Lindsey Davis, Suzanne Hammond, George Jacoby, Dyer Lytle, Steve Rooke, Frank Valdes, and Elwood Downey, with help from Ed Anderson, Jeannette Barnes, and Richard Wolff) and members of the VMS/IRAF group at STScI (Tom McGlynn, Jim Rose, Fred Romelfanger, Cliff Stoll, and Jay Travisano). The sharp editorial eye and sharper pencil of Chris Biemesderfer have made major contributions to the clarity and style of this document.

The continuing patience and understanding of members of the scientific staff at both institutions has been essential to the progress that has so far been achieved. A major software project such as IRAF cannot be attempted without the cooperation of many individuals, since the resources required must inevitably place a drain on other activities. In particular, the support and encouragement of Harvey Butcher, Garth Illingworth, Buddy Powell, Steve Ridgway and Ethan Schreier has been invaluable. Mention should also be made of Don Wells, who started in motion in the latter part of the last decade the process which eventually led to the creation of the IRAF system.

Peter Shames
Doug Tody

Contents

A User's Introduction to the IRAF Command Language
Version 2.3

Peter MB Shames
Space Telescope Science Institute

Douglas Tody
National Optical Astronomy Observatories

How to use this book

This document is an introduction to the IRAF Command Language (CL), and is designed to be a tutorial for the first-time user. The examples presented in the text can (and should) be tried at a terminal. Although this text is large enough to be a bit daunting at first, it can be tackled in easy stages, and need not all be read before trying the system. A basic knowledge of computer systems is assumed.

The first three chapters form an introductory section which covers the most basic elements of IRAF. Reading through these, preferably while seated near a terminal where the examples may be tried out, is the recommended entry into the IRAF world. The fourth and fifth chapters deal with the interface between IRAF and the host system, and with some of the more advanced uses of IRAF for normal data analysis activities. These chapters will be of use once you are familiar with the basic environment and the examples here are also designed to be tried out on a live system. The rest of this document is for the more adventurous user, who is not happy until he can say **doit** and get **it** done to a turn. Try some of these last examples when you are ready to customize IRAF for your own particular uses.

In spite of its size, this document is not intended to be a complete guide to using and programming the IRAF system, but is an introduction to many of the functions of the CL and a quick guide to other sources of more detailed information. The CL is described as the user's interactive interface to the system, and simple commands that use the terminal for starting and controlling tasks and for customizing the environment are presented. Development of simple functions in the CL are covered briefly here, but coverage of all the details of programming in the CL or in the IRAF environment is beyond the scope of this document. A reasonable amount of documentation is accessible at the terminal via the online help facilities, which are described here as well.

More extensive details of the CL may be found in the manual pages for the *language* package, in the *CL Programmer's Manual* and in *The IRAF User's Guide*. Details of programming in the IRAF system itself are described in the *Programmer's Crib Sheet for the IRAF Program Interface*, in the *Reference Manual for the IRAF Subset Preprocessor Language* and in other documents referred to in the last section of this text, however, these documents are somewhat dated and most of the documentation planned for the IRAF programming environment remains to be written. Documentation in the form of manual pages for the suites of applications packages being developed at both NOAO and STScI are available both online and in printed form.

1 Introduction

1.1 An Overview of IRAF

The Image Reduction and Analysis Facility (IRAF) has been designed to provide a convenient, efficient and yet portable system for the analysis of images and other classes of data. While the system has been designed for image data, and for astronomical image data in particular, it has general facilities that can be applied to many other classes of data. Some of the functions that are provided are quite specialized, dealing as they do with the characteristics of specific instruments, but others are generalized functions for plotting data, computing statistics, processing lists, and performing other functions that are common to data processing tasks in many other fields.

The runtime IRAF system consists of four basic pieces:

- Command Language - which provides the user interface to the system.
- Applications Packages - that are the real data analysis algorithms.
- Virtual Operating System (VOS) - which is the heart of the portable system and provides the foundation for all the higher level functions.
- Host System Interface (HSI) - the interface between the portable IRAF system and a particular host system. At the heart of the HSI is the IRAF **kernel**, a library of host dependent primitive subroutines that connects the system independent VOS routines to the host operating system. Each host system requires a different kernel, hence we speak of the UNIX/IRAF kernel, VMS/IRAF kernel, and so on.

All of these interconnected, yet separable, subsystems act together to form the IRAF data analysis environment. In most cases the user need not be further concerned with this structure, except to understand that the specific part of this structure that this tutorial addresses is the Command Language or CL.

IRAF is designed as an open system that can be extended to add new analysis capabilities, and that can support user customization of the existing facilities. There are several levels of customization available, that range from tailoring task parameters to special needs;

through "re-packaging" existing functions into application specific tasks; and extending to installation of new compiled code tasks in the environment. There are a variety of facilities provided in IRAF to assist the user in the creation and installation of such new tasks. It is not *essential* that all of these functions be performed in the IRAF way, but full integration of a user task within the IRAF environment can best be accomplished by using the facilities that are provided. However, even without the use of the IRAF interfaces, other tasks may be incorporated into the user's operating environment and used as if they were part of the distributed system.

The applications packages and the VOS routines are designed to be rather stable, and have been coded in the IRAF SPP language for portability. The kernel layer supports portability across operating system architectures, and its interface is stable, but the inner details change as a function of the requirements and capabilities of the host operating system. The CL is also rather stable, since it forms the user's interface to the system, but it is also an area where change is anticipated, as the system evolves to meet the needs of the users. Because the CL is itself a program that is supported by the VOS and isolated from the rest of the system by the VOS, it can be evolved as required without perturbing the other parts of the system.

1.2 Function of the Command Language

The basic function of the Command Language is to provide a clean, consistent interface between the user and the various packages of functions that complete the IRAF environment. The CL provides an interface between the user and all applications programs, giving the user complete control over the parameters, data, and system resources (graphics devices, etc.) used by IRAF programs. Many features have been incorporated into the CL to provide on-line support for users, whether they are old hands or new to the system.

The packages of programs that are provided offer many of the standard functions for data analysis, and they can be invoked interactively, one at a time, to perform many common operations. The execution of these functions is controlled by various parameters, and users may define their own values for the parameters as required. IRAF preserves the last value used, and presents it as the default value upon next use. Furthermore, there are facilities at several levels to allow users to assemble existing functions into new tasks that perform the specific operations on their data sets. This customization can involve new assemblages of existing functions or the inclusion of new functions written in the interactive CL language or in compiled languages.

The CL will primarily be used as a *command* language, but it is also an interpreted *programming* language. To be a good *command* language, the CL must make it as easy as possible to enter commands that perform common functions. To this end the CL provides command menus, minimum-match name abbreviations, parameter prompting and parameter defaults, tutoring on command parameters and options, and a concise syntax for simple commands. There is also a history mechanism that supports recall of previous commands and easy error correction within them.

A good interactive *programming* language must be reasonably efficient, be able to evaluate complicated expressions, to compile and run general procedures, and to offer the user an interpreted environment for investigating his data and exploring the applicable range of analysis techniques. This version of the CL (Version 2.3) includes all of the command language features of the earlier versions and makes major strides in the direction of becoming a powerful interactive programming language as well, although much remains to be done before the CL provides a reasonably complete and efficient interpreted programming environment.

This may sound complicated at this point, but examples presented throughout the body of the text will help clarify the use of the various features. We suggest a first reading of this introductory section and the next chapter, and then a session at the terminal, trying out the examples as presented.

1.3 Capabilities of the CL

Besides fulfilling the basic functions of a command language, the CL is capable of performing as a programmable desk calculator, evaluating expressions, executing CL script tasks or external programs, and doing some rather sophisticated programming functions. These features provide a means of connecting tasks to build new high level operators. The user's interaction with newly created tasks appears the same as interactions with the standard tasks and utility packages, as will become apparent in the discussions on usage and script tasks.

The CL has many features familiar to UNIX users in that I/O redirection, pipes and filters are provided. The output of any task may be routed to a file (redirection) or to another task (pipes), and many functions are provided to perform standard transformations on a variety of data types (filters). Be aware, however, that there are many differences between the CL and the UNIX command interpreters. The CL and the IRAF system present the user with a complete data analysis environment which is independent of the underlying operating system. Users running IRAF under UNIX, VMS, AOS, or some other operating system have the same analysis environment available to them and can type exactly the same commands while in the CL.

The CL supports an open environment in which packages of application specific tasks may be created and run. Some of these packages have been prepared by the developers to provide a variety of utility services, others that deal with specific instruments and analytic techniques are being made available, and still others can be created by you, to support your own view of the data analysis process. Beyond this, mechanisms exist that allow compiled external programs to be inserted in the system in such a way that they appear (and act) as an intrinsic part of IRAF. It is this open-ended nature that makes IRAF so powerful in its support of a variety of analysis activities.

2 Getting Started

2.1 Setting up the IRAF environment

A visitor wishing to use IRAF does not need to take any special action to do so. Computer support personnel will provide an account on one of the analysis computers and configure the environment as necessary to run IRAF. Staff members and long term visitors will already have established themselves with an account and will only need to perform a few simple operations before the CL and IRAF can be used. [1] After this has been done, all of the other commands referenced within this document will be available.

An interactive IRAF session begins with entry of the command **cl** to run the CL. When the CL starts up, it looks for a file called LOGIN.CL in the user's current directory. If this directory does not contain a LOGIN.CL file, the CL will function for simple things such as the evaluation of numerical expressions, but will not work properly for all functions. Therefore, you should always run the CL from a properly configured IRAF login directory. This directory needs to be initialized for IRAF before the CL is invoked; you can use the **mkiraf** command to setup the IRAF environment. The login directory, once set up, can be used for any number of sessions, and if you wish, you can set up several independent login directories and data directories for working with different types of data.

Summarizing the steps required to set up the IRAF environment:

1. Decide on a login directory.
2. Go there.
3. Type **mkiraf**.

That is all that is required. The **mkiraf** command performs several actions, the most important of which are making a LOGIN.CL file which you may wish to edit to change defaults, and the creation of a UPARM subdirectory, which is used by IRAF to store your customized parameter sets. The default login file consists mostly of environment declarations (**set** statements) that define directories, devices, and so on. The function of the environment and the significance of the standard **environment variables** are discussed in §4.2.

The **mkiraf** command can be entered at any time to reinitialize the environment, i.e., create a new LOGIN.CL from the system default and clear the UPARM directory. This is recommended periodically to pick up any recent changes in the system, and may be required when a major new release of the system is installed.

---

[1] **VMS :** The command `IRAF` must be entered to define system symbolic names. This command can be entered at the terminal or stored in your VMS LOGIN.COM file; it must be present in the LOGIN.COM file for queued IRAF background jobs to startup correctly.

2.2 Starting the CL

After configuring your IRAF directory, type the command **cl** to start the command language. After a bit the welcome message will appear on your terminal, and the first or root "menu" of IRAF will be displayed. This menu gives the names of the packages available through the CL. The `cl>` prompt will be issued indicating that the CL is ready to accept commands.

```
                        Welcome to the IRAF.

     dataio    images    lists     noao      sdas      system
     dbms      language  local     plot      softools  utilities

  cl>
```

Everything shown in the root menu of IRAF is a **package** name. A package is a set of **tasks** that are logically connected. For example, the *plot* package contains an assortment of general plotting tasks. You must *load* a package before any of the tasks therein can be run; you can load any package by simply typing its name.

`cl>` **plot**

would load the plot package and make the tasks in the package known to the CL. To unload the current package type *bye*; this frees any system resources used by the loaded package and restores the CL to state it was in before the package was loaded. Note that the system comes up with the *clpackage, system, language* and the default *user* packages already loaded (the *user* package allows the user to personalize the system, and is discussed in §5.6).

A comment should be made at this point about case sensitivity in IRAF. The CL accepts input in both upper and lower case, and distinguishes between them, i.e. a **'Y'** is different from a **'y'**. All command names are purposely specified in lower case, which is the default, and all user responses are expected to be in lower case as well. Upper case or mixed case names and commands are possible, but should be used with care.

Once the `cl>` prompt appears, many tasks will be available and ready for execution. A list of all loaded packages and the tasks in each package may be obtained by typing two question marks (**??**). This will list the tasks organized by package, starting with the current package. The packages are listed in the order in which they are searched when you type a command. Type one question mark (**?**) to list only the task names in the current package, or **?***packagename* to list the tasks in package "packagename".

2.3 Executing commands from the CL

At this point you may want to try executing a few simple commands. First try the **help** command. This will give additional information about the tasks in the current package.

`cl>` **help**

For detailed information about a particular package or task, type **help** followed by the name of the package or task for which help documentation is desired. For example,

`cl>` **help system**

will print detailed information about the *system* package, and

`cl>` **help page**

will print detailed information about the *page* task which is in the *system* package (after each page of text, the *help* program will prompt with a list of keystrokes and pause until you type one of them).

Now let's try running some tasks from the *system* package, which is already loaded. To display the file LOGIN.CL on the terminal, enter the following command:

`cl>` **page login.cl**

The *page* routine, like *help*, will pause at the end of each page of text, waiting for you to type a command keystroke, e.g., to display the next page of text, quit, return to the start of the file, go on to the next file if paging a set of files, and so on (typing `?` in response to the *page* prompt will cause a summary of the acceptable keystrokes to be printed). To get a directory listing of the files in the current directory, type:

`cl>` **dir**

Observe that all package, task, and parameter names may be abbreviated while working interactively. Any abbreviation may be given which contains sufficient characters to identify the name unambiguously; if the abbreviation is not unique, an error message is displayed. In general the first two or three characters are enough to identify most commands, but changes to the operating environment, i.e. loading additional packages, may require entering more characters or specifying the *packagename* as a prefix to unambiguously identify the required command.

To list all your files with the .CL extension, you can type:

`cl>` **dir *.cl**

As you gain familiarity with the CL you may find that you cannot remember the IRAF command to do something, but do know the correct command to use in the native operating system. There is an *escape* mechanism built into IRAF, in that any operating system specific command may be used by prefixing it with a '**!**'. There are some cautions to be observed that are described in detail (§4.1), but this knowledge may remove one possible source of frustration. Of course, the CL commands '**?**' or '**??**' may also be used to produce a display of the available package names and functions.

Packages are loaded the same way tasks are run, viz. merely by typing the name of the package as a command (a package is in fact a special kind of task). If the desired package is a subpackage of a package, the main package must be loaded first. For example, suppose we want to run the *precess* task. To find out what package *precess* is in we run *help* on the task *precess* and observe that the package path (printed at the top of the help screen) is "noao.astutil". This means that the *precess* task is in the *astutil* package which is in the *noao* package, which we recognize as a root level package.

We load first the *noao* package and then the *astutil* package by typing:

```
cl> noao
no> astutil
as>
```

The set of new tasknames now available to you will be displayed automatically. Note that the prompt will change from `cl>` to `no>` to `as>` to let you know you have entered another package.

One of the astronomical utility programs available is the *precess* program, which is used to precess lists of astronomical coordinates. The simplest way to run *precess* is to type only its name:

```
as> precess
```

The CL will then prompt you for the parameters it requires to run the program; in this case, the CL needs the name of an input file containing a list of coordinates to be precessed and the years over which the precession is to be computed. If you do not have the coordinates in a file, give the filename as STDIN (it must be upper case), and you can then enter the coordinates interactively from the terminal. Any number of coordinates (input lines from the special file STDIN) may be entered; signal the "end of file" for STDIN by typing the EOF key, e.g., `CTRL/Z`. [2] Coordinates are entered in pairs (RA and DEC, delimited by spaces) in either decimal or sexagesimal notation (e.g., 12.5 or 12:30:04.2). If you have any problems type **help precess** for additional information, including examples.

If you have a long list of coordinates to precess, try entering them into a file. The command:

[2] `CTRL/Z` is the standard EOF (end of file) sequence on VMS and most UNIX systems. Similarly, `CTRL/C` is the standard interrupt key on these systems. For simplicity we use the explicit control codes to refer to these functions in most of the IRAF documentation, but the reader should be aware that different control sequences may be used on the local system and be prepared to make the translations. For example, the key `CTRL/D` is often used to signal EOF instead of `CTRL/Z`.

`as>` **edit coord1950.txt**

will call up the default editor (Vi on UNIX systems; EDT or EMACS on VMS systems) to edit the file COORD1950.TXT. After creating your coordinate file and exiting the editor in the usual fashion, you will be back in the CL. Now try executing the *precess* program, using the file COORD1950.TXT as input:

`as>` **precess coord1950.txt**

Of course, the output will still appear on the terminal, and you may wish to **redirect** the output into a file as well:

`as>` **precess coord1950.txt > coord1984.txt**

If the coordinate list is *very* long, you may wish to process the list as a background job. To avoid interruptions from parameter prompts by the background task (it will inquire at the terminal), be sure to enter all the necessary parameters on the command line. To execute the task *precess* in the background, type:

`as>` **precess coord1950.txt 1950 1984 > coord1984.txt &**

The final '**&**' tells the CL to run the task in the background. The two parameters 1950 and 1984 will be passed to the task; you will not be prompted for them. Once the background task is started, the CL will be available for further interactive use and you will be informed when the background job is complete. The use of background tasks for batch processing is treated in more detail in §5.4.

2.4 A Comment on Input and Output

The notion of output **redirection** has already been introduced, and the topics of input redirection (accepting input from a file rather than the terminal) and **pipes** (connecting the output from one task to the input of the next) will be dealt with in §3.3. The point to be made at this time is that *all* tasks can be thought of as having three main I/O paths associated with them:

STDIN	the input path
STDOUT	the output path
STDERR	where error messages appear

By default, all of these I/O paths are connected to your terminal (referred to as TTY) and you may redirect any one or all of them using simple command line requests. The output *redirection* introduced in the previous example of *precess* is an example of just such an action. Other examples in §3.3 will cover this topic in more detail.

There are other standard output streams as well that depend on the specifics of the task. Not surprisingly, graphics tasks want to talk to a graphics terminal or other suitable device (STDGRAPH) and image tasks need access to an image display (STDIMAGE). There is a stream for the graphics plotter device as well (STDPLOT). Each of these *logical* devices is assigned to a physical device, either by commands in your LOGIN.CL file or by explicit parameters in the function calls.

2.5 The Graceful Exit

Now that you are a couple of layers deep into the CL, you may wonder how to get back out again. If you type **bye**, you will exit the current package and return one level of loaded packages. You cannot, however, type **bye** at the root CL level (`cl>` prompt). The command:

`cl>` **logout**

may be used to exit directly from the CL at any level. The **bye** command or the CTRL/Z sequence that signals EOF will exit from any task except the CL itself. This is to prevent an unintended *logout* from occuring if a series of EOF's are entered from the terminal.

For a less gentle departure from function execution, the interrupt sequence CTRL/C may be used at any level. This will usually terminate any task that appears to be hung or is operating in error, but will normally put you back in the CL in interactive mode.

3 Basic Usage

The CL can be used as both a command language and a programming language, but most first-time users (and many experienced ones) will mostly use the command features of the language. Commands to the CL may be entered at the terminal, one at a time, or they may be read in from a script file; in either case the syntax is the same and abbreviation of command names and variable names is supported. When the CL is being used for programming the rules are more restrictive, and full name specification is required, as is a more formal specification of task parameters. During the early sections of this document only the command forms will be used for simplicity. Parameters to a task may be specified on the command line for brevity, and prompting is automatically enabled for any required parameters that are not specified or that are given values that are out of range.

3.1 Command Syntax

The form of a command that calls an IRAF task is the *task name*, optionally followed by an *argument list*. The argument list consists of a list of *expressions* delimited by spaces. Simple filenames or string arguments that appear in the unparenthesized argument list need not be quoted, but any string that contains an embedded blank or other special characters should be quoted. *Positional* arguments (typically the first few arguments *required* for a function must be given first and in order. All of these may be followed by *param = value* keyword assignments, *param*± switches, and *file* I/O redirection assignments. These last three types of arguments may appear in any order. In general, the form is as follows :

`cl>` *taskname* [*expression* ...]	[*param=value*]	[<*filename*]
	[*param*±]	[>*filename*]
		[>>*filename*]
		[> &*filename*]

Any or all of these types of parameters may be present and defaults are provided for most parameters. In particular, the only parameters that *must* be set are the **required parameters** and if these are not specified on the command line, the CL will prompt for them. Other parameters and switch values are defaulted, but may be overridden if desired. The I/O streams typically default to the login terminal, but the redirection operators may be used to request: input from a file (<); output to a file(>); appending to a file (>>); or redirecting the standard output and the standard error stream to a file (>**&**).

The form of a command line need not be limited to a solitary call to a task. Several tasks may be called in sequence on a single command line, using the semicolon character ';' to delimit each call:

`cl>` **clear; dir**

If the command sequence is too long to fit on a single line, it can be enclosed in braces:

```
cl> {
>>> clear
>>> directory
>>> beep
>>> }
```

Note that the prompt changes to >>> after the first line to signal that the CL requires more input before it will execute the task. (In this particular example, the CL is waiting for a '}'.)

Such a construct is called a **compound statement** and may be used to aggregate several simple commands into a single new command. Compound statements may be used directly from the terminal (or within scripts as we shall see later) and will be treated as a single entity by the display and editing commands. An arbitrary number of commands may be entered in a compound statement and then executed as a single unit.

Commands may be strung together in another way too, by use of the pipe notation, which requests that the output of one command be used as the input to the next. Creation of the temporary files that support this, and connection of the task logical I/O paths to these files is handled automatically by IRAF.

```
cl> type coord1950.txt | precess 1950 1984
```

The pipe symbol '|' directs the CL to feed the output of one task (**type**) to the input of the next (**precess**).

If an argument list is too long to fit on one line, continuation is understood if the last item on a line is a backslash '\', the pipe symbol, or an operator (e.g., '+' or '//').

```
pl> graph "pix[*,5],pix[*,10],pix[*,15]" po+ marker=circle \
>>> xlabel=column ylabel=intensity \
>>> title = "lines 5, 10, and 15"
```

Quotes *may* be used around any string of characters, but are generally not required on commands entered at the terminal. In the previous example quotes are used around the string value of the **title** parameter because the string contains embedded spaces.

To make precise the rules for quoted strings: a string need not be quoted provided [1] it appears as an identifier (a name) in an argument list *not* enclosed in parentheses, AND [2] the string does not contain any blanks or other characters which are special to the CL, e.g., the i/o redirection symbols, the pipe symbol, semicolon, the begin comment character (#) or curly braces. If the string contains any special characters it must be quoted.

Comments may be freely embedded in a command sequence. Everything following the comment character on a line is ignored by the parser, so entire comment lines may be entered by starting the line with a comment:

`cl>` **# This is a full line comment**
`cl>` **type login.cl** **# Display the login file**

or by appending a comment to the end of a line as in the last example.

3.2 Task Parameters

Nearly all tasks have a formally defined set of parameters associated with them. The parameters for a task may be listed with the command **lparam** *taskname*. For example, to list the parameters for the task *delete*, type:

`cl>` **lparam delete**

The **lparam** command produces a display of the parameters of the named task in the order in which they must be given on the command line; it shows the current values of the parameters and the prompt strings as well.

After one types **lparam delete**, the following list will appear, giving the parameter name, its current value, and the prompt string associated with it:

```
           files =        list of files to be deleted
        go_ahead = yes    delete or not ?
         (verify = no)    verify operation before deleting each file ?
 (default_action = yes)   default delete action for verify query
    (allversions = yes)   delete all versions of a file ?
       (subfiles = yes)   delete any subfiles of a file ?
           (mode = ql)
```

Notice that there are two types of parameters, those with parentheses around the *param = value* fields and those without. The parameters not enclosed in parentheses are called **positional parameters**; they are required parameters and will be queried for if not given on the command line. Positional *arguments* are the first arguments on the command line (following the command itself), and they are associated with parameters by their position on the command line. The first positional parameter will be set by the first positional argument on the command line, the second positional parameter by the second positional argument, and so on.

The parameters enclosed in parentheses are called **hidden parameters**, and are the topic of the next section. Either type of parameter may be referred to by a **param = value** clause, although these parameter references must *follow* the positional arguments. Such name references *must* be used for the hidden parameters, but may be used for all.

Some of the parameter handling actions in the CL are rather elaborate and require discussion. As was just noted, the CL will automatically prompt for any required parameters that have not been provided in some way by the user. Beyond this, the normal action of the CL is to remember the parameters that you last used, and when that parameter name is next encountered, to offer the last value used as the new default value. This **learning** of parameters is intended to reduce user effort and is a means of customizing use of the system. The learned parameters are saved for you in the UPARM subdirectory, and will be preserved across uses of the system.

3.2.1 Hidden Parameters

The parameters of the **delete** task that appeared in parentheses are *hidden parameters* for the task. The CL does not query for hidden parameters, but automatically uses the default values. However, a query will be generated for even a hidden parameter if there is no default value or if the default value is illegal for some reason. Hidden parameters may be set on the command line, but unlike positional parameters, the value from the command line will not be learned, i.e., it will not become the new default value. The default value of a hidden parameter may be changed only by an explicit assignment, or by use of the *eparam* task (§3.2.3), and you should exercise caution in doing this, because it is easy to forget that hidden parameters have been changed.

Hidden parameters are often used to change the behavior of a task, achieving considerable flexibility without requiring many arguments on the command line, and without annoying queries for parameters. Hidden parameters make it possible to support functions like *graph* that support different display options, since users can modify the default behavior of the task to make it behave in the manner they want. Hidden parameters can also be dangerous if they are used improperly (e.g., for data dependent parameters in scientific programs).

The *delete* task is a good example of a task that is useful to personalize. The default behavior of *delete* is simply to delete the named file or files (provided they are not protected in some way). File deletion can be hazardous, of course, particularly since a pattern matching template may be used to delete many files. As many of us are unhappily aware, inadvertently typing

```
cl> delete *
```

will bring about the swift deletion of *all* of the (unprotected) files in the current default directory. As IRAF recognizes a number of special pattern matching metacharacters in addition to '*', one could easily free up a lot of disk space if one were not familiar with the use of pattern matching templates.

To reduce the possibility of such devastating side-effects, you might wish to change the default behavior of *delete* to verify each file deletion. This is done by changing the value of the hidden parameter *verify*, which defaults to *no*. Hidden parameters that are boolean flags (yes/no) may be overridden temporarily on the command line as follows:

`cl>` **delete *.dat verify=yes**

or, equivalently,

`cl>` **delete *.dat verify+**

Either of these commands would cause a prompt to be issued naming each file matching the template and asking if you want to delete it (this would happen even if the task were running in batch mode).

If you set a hidden parameter on the command line, you override the value of that parameter only for that command; the default value is not changed. As indicated before, to change the default value of a hidden parameter, an explicit assignment is required:

`cl>` **delete.verify = yes**

which will cause all subsequent file deletions to be verified, unless the **delete** command is issued with the argument **verify=no** or **verify**− on the command line. The change may be undone by another assignment, or by *unlearning* the task parameters.

3.2.2 Learning and Unlearning parameters

The CL facility called **learn mode** is designed to simplify the use of the system. By default, the CL automatically "learns" the value of all task **parameters** that are prompted for or explicitly set. In practice, this means that once a required parameter (such as the precession epoch in the *precess* example) has been set, it need not be respecified. The CL will still prompt for required parameters, but the default value displayed will be the last value you entered. Simply hitting RETURN will cause the CL to reuse the old value; but a new value may be entered and it will be preserved as the new default. If the required parameters are specified on the command line, you will not be prompted for them, and the value you specify will still be learned.

The parameter-learning mechanism has other ramifications as well. The most recently used parameter values are automatically preserved by the CL in .PAR files stored in your UPARM directory. These saved parameter sets are reloaded when you next start the CL, thus providing a *memory* of the options that you used in a previous session. Any command line arguments that you specify will override these *learned* defaults, but they will be available if you wish to use them.

An explicit command may be used to *reset* the values of parameters, i.e., to restore the defaults. The **unlearn** command restores the system default values of all of the parameters for a single task or for an entire package.

cl> **unlearn delete**

will restore the parameters of the task *delete* to their default values, and

cl> **unlearn system**

will restore the defaults for *all* of the tasks in the system package. If you want to restore the defaults for all the parameters in your IRAF environment, delete the .PAR files from the logical directory UPARM :

cl> **delete uparm$*.par**

3.2.3 Specifying Parameters to a Task

The simplest and fastest way to invoke a task is to simply type in the name of the task followed by the necessary arguments on the command line, as we have been doing in most of the examples thus far. In many cases, the arguments for a task will be obvious, either from the context and the prompts issued by the task, or from the **lparam** display. If you are unsure about how to proceed, you can simply type the task name, and answer the questions. Each prompt may include minimum and maximum acceptable values, if such apply, and the current value of the parameter if such exists. For parameters that have only a fixed set of allowable values the list of valid options will be enumerated.

Alternatively, the **eparam** command may be used to invoke the parameter *editor*. The *eparam* task presents the parameters of a task in a tabular display on the screen and supports the use of the cursor keys to navigate the options. It also has commands for changing entries, or for recalling previous entries for further editing. The command:

cl> **eparam precess**

will display the parameters for *precess* (the *noao* and *astutil* packages must first be loaded). The RETURN key will move you down the list or the cursor keys may be used to move among the parameters, and any entries that you type will replace the displayed values. You may exit from *eparam* at any time with a CTRL/Z and the parameters for the task will be updated with your newly edited values. If you wish to exit the editor *without* updating the parameters, use the interrupt request CTRL/C instead. Specifying parameters via **eparam** has the same effect as does entering them on the command line, they will be remembered by IRAF and not prompted for when the function is next invoked.

Eparam and the history editor **ehistory** both use the same simple set of editor commands, and they can mimic several editors that are common on the currently supported systems. For any of these editors the default style supports use of the cursor (arrow keys) on the terminal and the use of the DELETE key. The sections on editors (§5.2-3) describe this in more detail.

If you find that you must invariably run *eparam* before running a particular task, e.g., because the task has too many parameters to be specified on the command line, it is possible to get the CL to run *eparam* for you automatically whenever the task is run interactively. This is called **menu mode**. To set menu mode for a task we set the string value of the *mode* parameter of the task; all tasks have such a parameter. For example,

`cl>` **precess.mode = "ml"**

will set both menu and learn mode for the task *precess*. The default mode for most tasks is **ql**, i.e., query (the task will query for all parameters not set on the command line) plus learn (old parameter values are learned).

Once you are familiar with the operation of a task, you can enter the parameter values on the command line in the order in which they appear in the **lparam** listing. Parameters may also be set using the **param = value** clause on the command line, but remember that any positional arguments must be given first. Note that a command line argument may be any general expression, much like the arguments to a Fortran subroutine.

`cl>` **precess stdepoch= (1984+i*4)**

Here an expression is used to compute the value of the hidden parameter *stdepoch*. Note that the expression must be enclosed in parentheses in order to cause it to evaluated, since it will otherwise be treated like a string and just passed into the task for it to handle. The variable **i** must previously have been set to some legal value; otherwise the CL will prompt for it.

3.3 Pipes and I/O Redirection

We have already seen how tasks can take their input from either the terminal or from a file, and send the output to either the terminal or a file. By default, both the standard input and standard output for a task are written to the user terminal; the capability to change them on the command line is called *I/O redirection*. The Appendix of IRAF commands at the end of this document was created with the following simple command:

`cl>` **help** *pkg* > *pkg.txt*

where the name of each package was substituted for *pkg*.

The pipe syntax is a powerful kind of I/O redirection. A pipe is formed by connecting the output of one task to the input of another task; an arbitrary number of tasks may be connected together in this way to form a single command. UNIX users will already be familiar with the concept and uses of pipes, but be aware that CL pipes differ from UNIX pipes in that the CL tasks execute *serially* rather than concurrently (i.e., nothing comes out of the end of the pipe until *all* the input has been processed). Another difference between IRAF and the usual UNIX implementation is that IRAF pipes are implemented with temporary files which are managed by the system. Note also that queries for parameters are not affected by the use of I/O redirection or pipes, i.e., required parameters will still be prompted for when requested by a task.

A simple example of the use of a pipe is redirecting the output of a command to the line printer. This can be done with I/O redirection as follows:

```
cl> help plot > temp
cl> lprint temp
cl> delete temp
```

The pipe notation accomplishes the same thing and is more concise:

```
cl> help plot | lprint
```

For a more sophisticated example of the use of pipes, load the **lists** package and try out the following command:

```
cl> ?? | words | match : stop+ | sort | table
```

This sequence of commands takes the list of menus produced by **??**, breaks it into a list of words, filters out the lines that contain the colon character (the package names), sorts the list, and prints a menu listing the tasks in all loaded packages.

The following example shows the use of a pipe-filter to sort the output of a long form directory listing of the system library directory LIB, sorting the list in reverse numeric order by the size of the file, so that the largest files come out at the top of the list:

```
cl> dir lib l+ | sort num+ rev+ col=3
```

We can go a bit further and extend the pipe to print only the ten largest files and page the output:

```
cl> dir lib l+ | sort num+ rev+ col=3 | head nlines=10 | page
```

Any or all of the input, output or error logical I/O streams may be redirected with simple command line requests. The next example shows the use of redirected input and output streams:

`cl>` **match ^set < home$login.cl > names.env**

This command reads from your LOGIN.CL file, created by the initial **mkiraf** command, matches all the lines that contain **set** environment statements (the metacharacter ^(up-arrow) causes **set** to be matched only at the beginning of a line), and writes these out into the file NAMES.ENV.

The > redirection operators will create a new output file. To append to an existing file we use the >> operator instead:

`cl>` **set** | **match tty >> names.env**

which will scan for all the environment variables having something to do with the terminal and append them to the file NAMES.ENV.

The operators > and >> will redirect only the standard output stream STDOUT; error messages will still come out on the terminal. To redirect both STDOUT and STDERR the operators >**&** and >>**&** should be used instead.

The graphics output streams may be redirected (but not piped) much as is done for the ordinary textual output streams.[3] For example, to redirect the standard graphics output of the *surface* task (in the *plot* package) to produce a graphics metacode file SURF.MC:

`cl>` **surface dev$pix >G surf.mc**

To redirect the STDIMAGE stream, substitute the operator >**I**, and to redirect the STDPLOT stream, use the operator >**P**. The characters **GIP** must be uppercase. The > may be doubled to append to an existing file, as for the standard text streams. As a special case, a graphics stream (or indeed any stream) may be redirected to the so-called *null* file DEV$NULL to discard the output. For example,

`cl>` **prow dev$pix 100 >G dev$null**

will plot row 100 of image DEV$PIX, redirecting the graphics output into the null file. The null file can be used anywhere a normal file can be used.

[3]This holds only for standard IRAF tasks, i.e., tasks which use the IRAF graphics subsystem. This feature is not currently available for the STScI SDAS tasks since they do not use the IRAF graphics facilities.

4 Operating System Interface

Although IRAF provides a quite complete environment for data analysis activities, it must be hosted in some particular operating system whenever it is being used. The isolation from the peculiarities of any specific operating system command syntax is rather complete, but there are instances where the syntax of the underlying system must be used (host filenames) or where the user may desire to use familiar commands from the host system. IRAF does allow commands to be passed through to the host operating system, but because IRAF maintains all of its own environment descriptors, directory structures, and task and program information, the operating system commands should only be used to bring information into the IRAF environment, but not to modify it. In order to change any of the status or control information that affect IRAF execution, the commands provided by IRAF must be used.

4.1 Sending Commands to the Host Operating System

IRAF allows access to the underlying operating system, and hence to other programs that operate within the native operating system environment. There are limitations on some of the system facilities that can be used without regard to side-effects, but, in general, almost any program can be called from within IRAF. External programs can be accessed from within the user's environment and will operate with a standard interface that is compatible with the rest of the processing functions that are available.

Any command may be sent to the host operating system by prefixing the command with the escape character '**!**'. The rest of the command line will be passed on unmodified. For example, to read your mail on a UNIX or VMS system:

`cl>` **!mail**

Upon exiting the mail routine, you will be back in the CL. Almost any task that is executable in the normal host environment can be invoked from within IRAF by means of this escape mechanism. The OS escape is used to implement some of the standard IRAF commands that request operating system information, such as **spy**. The **edit** command also uses the escape mechanism, so that the host supported editors can be used, rather than require that a completely new editor be learned in order to use IRAF.

Occasional conflicts will arise if these external tasks re-assign their terminal input and output streams or perform other unnatural acts. If strange things happen when trying to use such tasks from within the CL, consult your **IRAF Guru**. The other major source of problems with host system tasks is that they may depend upon system specific data that have been defined for the OS but are unknown to IRAF. This is a particular problem under VMS, which does not pass system environment parameters to sub-tasks, as does UNIX. Variables that affect the execution of tasks within the environment are controlled by IRAF and are passed between the executing tasks, as in described next.

4.2 Environment Variables

The CL maintains a table of environment variables which affect the operation of *all* IRAF programs. The environment variables are used to define logical names for directories, to associate logical device names with a specific physical device, and to provide control over the low level functioning of the IRAF file I/O system. The default environment is created by IRAF at login time, i.e., when the CL is first run. Part of this initialization uses a standard system-wide, site dependent file named HLIB$ZZSETENV.DEF. Additional initialization of personal environment variables, or redefinition of standard environment variables, may be done with commands in your LOGIN.CL file.

One may add new environment variables, or redefine old ones, at any time during a session with the **set** command. **Set** declarations made during CL execution, however, may be lost upon exit from a package. To secure environment declarations for a full session, make them *immediately* after logging in. To make environment declarations permanent, place the relevant **set** commands in your LOGIN.CL file.

The **set** command is usually used to change the *session* defaults for output devices and such, but all IRAF programs which write to the line printer or to a graphics device also permit the device to be selected on the command line. For example,

`cl>` **set terminal = vt100**

informs IRAF that the user is using a VT100-type terminal for this session. When typed without any arguments, e.g.:

`cl>` **set | page**

set displays a list of the current values of all of the environment variables. Note that abbreviations are *not* supported for environment variable names, they must be spelled out in full. If a shorter name is used the CL will silently create a new environment variable for you, which may not be what you desired at all.

Identifying the kind of terminal you are using, the size of the display window to be used, and setting other terminal options may most conveniently be done with the **stty** command:

`cl>` **stty tek4014 baud=1200**

This command should be used early in the session (if not already present in the LOGIN.CL file) to identify the kind of terminal that you are using, since the operation of the various editors and of other functions will be affected by these values. It is only necessary to set baud rate as in the example if you are working remotely via modem. As was the case with the **set** command, typing **stty** with no arguments will display the current terminal type and settings.

The current value of *individual* environment variables may be displayed with the **show** command:

`cl>` **show printer**

A selection of the more important environment variables is shown in the following table.

Selected Environment Variables		
variable	*sample value*	*usage*
terminal	"vt100"	default terminal device
printer	"printronix"	default line printer device
stdgraph	"vt640"	name of graphics terminal device
stdplot	"versatec"	batch plotter device
stdvdm	"uparm$vdm"	name of graphics metacode file
stdimage	"iism75"	image display device
clobber	no	clobber (overwrite) output files
filewait	yes	wait for busy files to become available

Clearly, the permissible names of devices are site dependent; for a list of the devices available at a particular site the user should consult their **IRAF Guru** (or look in the TERMCAP and GRAPHCAP files in the IRAF logical directory DEV).

Among the set of environment variables that control the operation of the CL is a subset of variables that define the user environment. These variables describe the user's home and scratch directories, terminal type, and editor preference. Because these values describe a user's-eye view of IRAF, they can be thought of as *customization* variables and can be **set** in the LOGIN.CL file to your preferred values.

User Environment Variables		
variable	*sample value*	*usage*
editor	"vi"	default editor mode
home	"/*user*/iraf/" [4]	user home directory
uparm	"home$uparm/"	user scratch directory
imdir	*system-dependent*	directory where bulk data is stored
imtype	"imh"	default image type (header file extension)
userid	*user*	user identification name (for output)

The HOME directory specification, and possibly an IMDIR declaration should be the *only* places in your LOGIN.CL file where any system specific names appear at all. All of the IRAF name references (except a single root reference) are processed by the virtual name mapping algorithms. If this same mechanism is used for all user files as well, then only IRAF virtual filenames need to be referenced once the root directory has been properly specified.

[4] **VMS :** an equivalent VMS example might be "DISK\$1:[USER.IRAF]". Note that any dollar sign characters appearing in host filenames must be escaped in IRAF since the dollar sign is a reserved character in IRAF filenames.

The default *uparm* declaration assumes that a UPARM subdirectory has been set up in your login directory; the **mkiraf** command described earlier (§2.1) sets this up for you. If a UPARM subdirectory does *not* exist, the CL will refuse to update user parameters and will issue a warning message.

4.3 File and Directory Names

The IRAF system employs **virtual file** names so that all file references will look the same on any computer, and IRAF primitives convert virtual filenames into their host operating system equivalents. In general, either the IRAF virtual filename or the operating-system-dependent filename may be used in a command entered by the user, but users should avoid the use of OS-specific names wherever possible. Internally IRAF itself uses only virtual filenames for reasons of transportability.

Note that filename mapping does not operate automatically for virtual file names that are passed as parameters to foreign (host system) tasks, but a CL intrinsic function *osfn* will perform the mapping if called explicitly on the command line. The host task must be declared as an IRAF *foreign task* (§5.6) for this to work. There is no provision for filename mapping when the regular OS escape mechanism (§4.1) is used.

The environment variables described in the preceding section play a fundamental role in the mapping of virtual filenames. Environment variables define the logical directories that are used to create host operating system specific names from logical names. An example of a virtual filename is the default logfile, HOME$LOGFILE.CL. The HOME field, delimited by the '$' character, is the logical directory; the file name within that directory is LOGFILE.CL. Successive translations of '$'-delimited logical names are performed until the operating system dependent name has been generated. Names such as HOME$UPARM/ are *directory* references; the trailing '/' indicates that a filename or sub-directory name may be appended to produce a legal file or directory pathname.

4.3.1 File Name Templates and Metacharacters

Although filenames cannot be abbreviated the way commands can, pattern matching templates can be constructed that refer to many files. You need only type a short string (the pattern) that serves as a *template*, and all files whose names match the template are selected. All of the IRAF functions that process filenames (and this is most of them) use the same function to expand filename templates into a list of files. The pattern matching **metacharacters** are a super-set of those used in the UNIX and VMS operating systems. To print all files having the extension .CL, type:

`cl>` **lprint *.cl**

To page through all files in the logical directory FIO with the .X extension, type:

`cl>` **page fio$*.x**

The filenames matched by the file template are passed to the **page** task which pages through the set of files. As each file is accessed, the VOS filename translation facilities are used internally to generate the host system filename, which is passed to the kernel to physically open the file.

Pattern Matching Metacharacters		
Meta-char	*Meaning*	*Example*
*	Match zero or more characters	*.cl
[...]	Any character in class	[a-z]
[^...]	Any character not in class	[^A-Z]
?	Match any single character	a?c
{...}	Ignore case for the enclosed string	{Lroff}
@*file*	Read filenames from a list file	@listfile

To delete a named list of files, type:

`cl>` **delete file1,file2,file3**

Note that the list of filenames is separated by commas **','** *with no intervening blanks*. This causes the individual filenames to be treated as one list-form parameter rather than to be processed as three separate parameters. A blank is treated as a delimiter by the parser, and thus may not appear in a list-form parameter unless the list is enclosed in quotes.

The following is equivalent to the previous example, except that no warning will be issued if any of the three files does not exist, since we are asking the system to find all files that match the template, rather than naming the files explicitly:

`cl>` **delete file[123]**

Consider the following simple command:

`cl>` **delete filex**

The name "filex" given here is actually ambiguous; it could be either the name of a file (a string constant) or the name of a string parameter set to the name of the file to delete. In this simple and common case, the CL will quietly assume that "filex" is the *name* of the file. If the identifier **filex** is really the *name* of a variable, it will have to be parenthesized to force it to be evaluated. Either of the following forms are equivalent to this command and both are unambiguous requests to delete the file named FILEX:

`cl>` **delete 'filex'**
`cl>` **delete ('filex')**

Note that within parentheses the *string* **'filex'** must be typed as shown, with quotes, or the CL will attempt to process it as a variable name, causing a runtime error if there is no such variable currently defined within the scope of the *delete* task.

The following command is also unambiguous, and it specifies that the CL is to take the name of the file from the *parameter* "filename":

`cl>` **delete (filename)**

Note that in many of these examples, a *single* string type argument, viz. the file matching template with metacharacters, is used to refer to a list of files. This convention is employed by all IRAF tasks which operate on lists of files. Be careful not to confuse a file list template with the argument list itself. Thus:

`cl>` **delete file,file2,prog.∗**

is perfectly acceptable, and does what the next example does:

`cl>` **delete 'file1, file2, prog.∗'**

as long as there are no blanks between elements of the first name list. If blanks were inadvertently included in the unquoted template string the CL would interpret the template as several string arguments, probably causing an error something like "`too many positional arguments`".

The list file approach is useful when it is difficult to specify a template for the desired set of files, when the same set of files will be operated upon several times, when a very large number of files are to be operated upon, or when a list is already available. The file list may be generated by the editor, or by a task such as *files*, e.g.:

`cl>` **files *.im,run[1-4].* > listfile**

The textfile LISTFILE may then be referenced in a filename template as **@listfile** to operate upon the listed files. A variation on the listfile approach is **@STDIN** (must be upper case), which allows the filenames to be typed in when the task begins running.

Some tasks use the filename template mechanism to generate the names of a *new* set of *output* files. The filename template expansion code provides two operators for generating new filenames from old ones. The file template *operators*, which are used to construct new filenames, should not be confused with the pattern matching *metacharacters*, which are used to match a subset of an existing set of files.

The first and simplest operator is the string concatenation operator `//`. This may be used to concatenate a string suffix to the *root* field of a filename, to concatenate a filename to a string prefix, to concatenate two filenames, or some combination of the above. For example,

```
cl> files lib$*.com//_o
```

will produce a new list of files by appending the string `"_o"` to the root of each filename matched by the template at the left.

The second and last operator is the string substitution operator `%`. If a sequence of the form `%a%b%` is inserted somewhere in a file template, the string **a** will participate in the pattern matching operation, but will be replaced by **b** in the generated filename. Either **a** or **b** may be omitted to insert or delete fields from a filename. For example,

```
cl> files lib$*%%_o%.com
```

is equivalent to the concatenation operation illustrated in the preceding example. The command

```
cl> files lib$*.%com%dat%
```

would find all the .COM files in the logical directory LIB, generating a new list of files with the extension .DAT substituted for .COM.

All IRAF tasks that use pattern matching or template expansion use the same syntax and metacharacters as in the examples given here for filename templates. This includes, for example, the use of templates in the *help* task to locate manual pages, and the use of pattern matching in the *match* task to search text files for lines that match a pattern.

4.3.2 Directories and Path Names

It is often useful to employ several different directories as an aid to organizing your data. For instance, you may have one directory for M87 data, and one for M8 data, or, as was set up for you by the **mkiraf** command, a login directory HOME and a scratch directory UPARM. New directories may be created with **mkdir**; use **chdir** or **cd** to change the default directory, and **back** to return to the most recent default directory.

For example, to display the pathway through the system to your current default directory, type:

cl> **path**

To change to a new default directory, type:

```
cl> chdir newdir
```

where *newdir* may be an IRAF logical directory name defined with a **set** command, an IRAF pathname to the directory, or a host system directory name (provided any dollar sign characters therein are escaped).

The **mkdir** command can be used to create a new sub-directory of the current directory:

```
cl> mkdir m87
```

To define a logical directory name ("m87") for this subdirectory of your home directory, use the following set command (note the trailing '/'):

```
cl> set m87 = 'home$m87/'
```
[5]

Once this logical name mapping has been established, you may type either of the following commands to change the default directory to the "m87" directory (note **chdir** may be abbreviated **cd**):

```
cl> chdir m87
cl> cd home$m87
```
[6]

If you type **chdir** or **cd** without any arguments, the default directory will be set to your "home" directory.

Once a logical directory has been defined, the IRAF pathname notation may be used to reference any file or directory in the vicinity of the new logical directory. For example, the following command would page the file CURSOR.KEY in the subdirectory SCR of the subdirectory LIB of the IRAF root directory IRAF, a predefined logical directory:

```
cl> page iraf$lib/scr/cursor.key
```

The current directory and the directory one level up from the current directory may be referenced in pathnames via the synonyms "**.**" and "**..**". For example, if the current default directory is PKG, a subdirectory of LIB like SCR in the preceding example, the path to the CURSOR.KEY file could be entered as follows:

---

[5] **VMS :** IRAF supports logical names for files and directories that may contain mixed cases and special characters. However, to avoid unpleasant surprises, we recommend that for root directories you use only names valid for the underlying operating system.

[6] **VMS :** The characters `$` and `[`, commonly used in VMS device and directory names, will cause a conflict if VMS file or device names using them are passed to IRAF tasks since these characters have a special meaning in IRAF filenames and filename templates. If either of these characters is used in a VMS filename passed to an IRAF program, the character must be escaped to avoid interpretation as a VOS metacharacter, e.g., `page usr\$0:\[iraf.local]login.cl`.

```
cl> page ../scr/cursor.key
```

It is not necessary to change the default directory to reference files located in another directory. Your login directory, for example, has the logical name HOME$ assigned to it. The following command would page through the LOGIN.CL file in your home directory, regardless of the current default directory:

```
cl> page home$login.cl
```

The logical directory names (UPARM and IMDIR are examples of directories that are normally appended to the HOME directory, and you may set up other logical directories as required. The names of all of the standard IRAF system directories are defined automatically when the CL starts up, and may be listed with the *set* command.

4.3.3 Virtual Filename Processing

Virtual filenames are used throughout IRAF and the CL in preference to operating system specific names. The obvious reason for this is to isolate OS specific interfaces to a small set of locations, as a way of ensuring commonality across operating systems and as an aid to portability. There is an obvious benefit to the user as well, in that filename references will look the same within IRAF regardless of the host environment. Operating system specific names must eventually be generated, but the details of these operations are best buried in dedicated interface routines.

The only place where OS specific names need appear at the user level is in file system directory names and in references to system physical devices. Even here, the use of OS specific names should be isolated to only one or two root directory names. The other place where OS names must appear is calls to operating system routines or to external programs that are accessed from within IRAF via the OS escape mechanism (§4.1). The *pathnames* task and the *osfn* intrinsic function are used to translate IRAF virtual filenames into host system filenames.

Either of the following commands will print the fully qualified OS name for the file HOME$LOGIN.CL.

```
cl> path home$login.cl
cl> = osfn ('home$login.cl')
```

The *pathnames* task writes the translated filename on its standard output, while *osfn* returns the translated filename as the function value. The **pathnames** task will also expand filename templates, and thus can be used to generate the OS names for a list of files:

```
cl> path home$ss433.* > ss433files.list
```

will generate a list of all of the files in directory HOME that match the template, and will write the fully qualified OS names of these files into SS433FILES.LIST. This ASCII file can be edited as necessary, and used as list-structured input to other IRAF functions (§2.3, §6.9, §7.3).

The most common use of the *pathnames* task is probably to print the current default directory, which is its function when called with no arguments on the command line.

4.4 Image Data

An IRAF *image* is an N-dimensional data array with an associated *image header* describing the physical and derived attributes of the image. The content of the header tends to be very data or application specific. The datatype selected to store the *pixels* (data values) is also application dependent, and a variety of choices are provided. Images of up to seven dimensions are currently supported, although in practice most images are either one or two dimensional, and most programs are written to operate upon one or two dimensional images. Any IRAF program can be used to operate upon a *section* of lesser dimension (or extent) than the full image, using the *image section* notation discussed in §4.4.3, hence the dimensionality of the algorithm implemented by a program need not prevent use of the program on images of higher dimension.

4.4.1 Image Names and Storage Formats

The notation used to refer to images is similar to that used to refer to files, except that images are more complex objects than files and hence a somewhat more complex notation is required. Most of the file, directory, and pathname notation discussed in §4.3 carries over to images. Sets of images are referred to by an *image template* notation which is an extension of the file template notation discussed in §4.3.1.

In most, but not all, cases, an IRAF image is stored on disk in two separate files, one containing the image header and the other containing the pixels. The basic image name is the filename of the header file. The filename of an image header file always has an extension specifying the format in which the image is physically stored on disk. [7] Two storage formats are currently supported, the old iraf format (OIF) and the SDAS group data format (STF). The old IRAF format images have the extension IMH. The STF images may have any three character extension ending in H, e.g., HHH (the extension IMH is reserved for OIF images, of course). Both types of images may be accessed at any time, with the extension being used to identify the physical storage format to the IRAF software.

[7] In versions of IRAF prior to V2.3, only one physical image storage format was supported, hence image header files did not have extensions.

For example, the IRAF system is distributed with a standard OIF format test image PIX stored in the system directory DEV. The full filename of the header file is DEV$PIX.IMH. To make a copy of this image in the current directory we could load the *images* package and enter the following command:

`cl>` **imcopy dev$pix pix**

or since we don't want to change the image name,

`cl>` **imcopy dev$pix .**

Note that we did not have to specify the image type extension in the copy operation. The extension is optional whenever a single image is referenced; in image templates, the template must match the full filename of each image as it appears in a directory listing, hence the extension is required in image templates.

Sometimes it is necessary to specify the image type extension to force an image of a certain type to be created. For example,

`cl>` **imcopy dev$pix pix.bah**

would create an STF format copy of the standard test image in the current directory.

When making a copy of an existing image, the new image will have the same format as the old image unless an extension is specified in the output image name. When creating a new image from scratch, e.g., when reading a data tape to disk, the default image type is determined by the value of the CL environment variable IMTYPE, the value of which is the three character default image type extension. If IMTYPE is not defined, the default value is **imh**, i.e., an OIF format image will be created. To change the default to be to create an STF format image, add a command such as

`cl>` **set imtype = hhh**

to your LOGIN.CL file.

4.4.2 Image Templates

Image templates are equivalent to filename templates except that the character `'['`, a pattern matching character in filename templates, has a different meaning in image templates, as we shall see in the next section. [8]

For example, given a directory containing the files

[8] If you really want to perform file template style character class expansion in an image template, use the operator `![` instead of `[`. The conventional escape mechanism, i.e., `\[`, is used to include the `[` in the *filename*, as in a filename template.

```
irs.log  irs.0030.imh  irs.0031.imh  irs.0032.imh
```

the template **irs.*.imh** would match the three image files, whereas **irs.*** would match the LOG file as well, causing *imheader* to complain about an illegal format image in its input list.

4.4.3 Image Sections

All IRAF programs which operate upon images may be used to operate on the entire image (the default) or any section of the image. A special notation is used to specify **image sections**. The section notation is appended to the name of the image, much like an array subscript is appended to an array name in a conventional programming language. Note that array or image section *index references* are integer only in pixel coordinates, but that the data may be of any valid type.

section	*refers to*
pix	whole image
pix[]	whole image
pix[i,j]	the pixel value (scalar) at [i,j]
pix[*,*]	whole image, two dimensions
pix[*,-*]	flip y-axis
pix[*,*,b]	band B of three dimensional image
pix[*,*:s]	subsample in y by S
pix[*,l]	line L of image
pix[c,*]	column C of image
pix[i1:i2,j1:j2]	subraster of image
pix[i1:i2:sx,j1:j2:sy]	subraster with subsampling

A limited set of coordinate transformations may be specified using image sections, but please observe that transpose is *not* one of them. The "match all" (asterisk), flip, subsample, index, and range notations shown in the table may be combined in just about any way that makes sense. As a simple example:

`cl>` **graph pix[*,10]**

will graph line 10 of the image PIX. To generate a contour plot of an 800-pixel square image subsampled by a factor of 16 in both dimensions:

`cl>` **contour pix[*:16,*:16]**

To display the fifth $x - z$ plane of a three dimensional image named **cube** on frame 1 of the image display device:

```
cl> display cube[*,5,*] 1
```

The image section string is part of the image name and is processed by the IRAF system software (rather than by each applications program), hence image sections can be used with all IRAF programs. A section can be used to write into a portion of an existing output image, as well as to read from an input image.

4.4.4 The OIF Image Format

The old IRAF image format (OIF) is the original IRAF image format, unchanged since it was first used in the beginning of the project. It is called the "old" format in anticipation of its eventual replacement by a new format to be layered upon the planned IRAF database facilities. The OIF format is the current standard IRAF image format and is the format used to test the IRAF image processing software at NOAO.

In the OIF format, each image is stored in a distinct pair of files, the header file (extension IMH) and the pixel file (same root name as the header file, extension PIX). The pixel file need not reside in the same directory as the header file; by default all pixel files are created in a user directory on a scratch disk device to permit a different file quota, file expiration, and backup policy to be employed than is used for the smaller, more permanent ordinary user files.

The CL environment variable IMDIR determines where OIF pixel files will be created. IMDIR is a required parameter and is normally defined in the user's LOGIN.CL file. The value of IMDIR is only used when the pixel file is created; if the value of IMDIR is later changed, new pixel files will be created in a different directory, but the system will still be able to find the pixel files of the older images.

By default, the *mkiraf* script will create an image storage directory for the user on a public scratch device and place the host pathname of the new directory in the user's LOGIN.CL file. For example, on a UNIX system, a typical set environment statement might be:

```
set imdir = /tmp2/iraf/user/
```

which will cause the pixel files to be created in the named host directory, regardless of the directory in which the image header file resides. As an option, we can request that the pixel file be placed in the *same* directory as the header file:

```
set imdir = HDR$
```

or in a subdirectory of the header file directory, e.g., subdirectory PIXELS:

```
set imdir = HDR$pixels/
```

Note that the reserved logical directory name HDR must be upper case, and that a trailing slash is required if the subdirectory option is used. The subdirectory will be created automatically by the system when the first pixel file is created, if the directory does not already exist. The HDR option should *only* be used if the header file itself is created in a directory on a scratch device; it should always be used if the image is created on a remote node in the local network.

4.4.5 The STF Image Format

The STF image format is the format used by STScI to store Space Telescope image data. IRAF provides a dedicated image kernel to read and write this format so that sites reducing binary ST data do not have to carry out expensive format conversions to be able to access the data from within IRAF. SDAS users should note that the SDAS software can *only* access STF format images, hence the STF format must be used if you plan to make extensive use of SDAS. Reductions involving only IRAF programs should not use the STF format, since the OIF format is simpler and more efficient, and is the format used to test the IRAF software.

In the STF format, an image or a *group* of similar images may be stored in a pair of files, the image header file (extension **??H**), and the associated pixel storage file (extension **??D**). If multiple images are stored in a group format image, all member images share the same group header. The group header file is a special VMS format text file which can be examined by *page* and *type*, as well as with *imheader*. Each member image in a group format image also has its own private binary format header, called the group parameter block. The STF image format supports only single precision real pixels, since that is what SDAS programs require.

IRAF programs consider images to be independent entities, with any associations between images being left up to the user. When a member image of an STF group format image is accessed from an IRAF program, IRAF constructs the image header of the member image by concatenating the group header to the group parameter block for the member image; no distinction is made between the two classes of header parameters once the image has been opened.

To refer to a specific member image of a group format image, the group subscript must be specified in the image name. If there is an image section as well, it comes after the group subscript. For example, if WFPC is an STF group format image,

```
cl> implot wfpc[3]
```

would call up the interactive image plotting task *implot* on group 3 of the group format image. If no subscript is specified, the default is group 1. To plot the same image with the lines flipped end for end, we add an image section:

```
cl> implot wfpc[3][-*,*]
```

To create a new group format image, we must preallocate space for all the member images, all of which must be the same dimensionality, size, and datatype. For example,

```
cl> imcopy wfpc wfpc2[1/10]
```

would create a new group format image WFPC2 with the same dimensionality, size, and group parameter block as the existing STF image WFPC, then copy the pixels from WFPC to WFPC2[1]. The new image would inherit the header of the old image as well. Once a new group format image has been created, the remaining member images may be written into by specifying the group subscript in the output image name passed to an IRAF program. The group count (**/10**) should be omitted, else IRAF will try to create a new group format image, rather than write into one of the member images of an existing group. Note that member images cannot be added or deleted once a group format image has been created.

5 Advanced Topics in the CL

In addition to the basic facilities already described, the CL permits user control over many aspects of the environment. This includes direct control over the CL itself, control over user tasks and background processes, the job logfile and the command history mechanism. These features and others will be of use to the more advanced user and enable user customization of interactions with the system.

5.1 CL Control Parameters

The CL is itself a task which has a set of parameters that are used to direct its execution. For example, if you wish to keep a permanent record of all the commands you enter, the CL will do this if you set its boolean parameter *keeplog* to yes. (Boolean parameters can assume only the values *yes* or *no*.) Simply type:

`cl>` **keeplog = yes**

and all subsequent commands will be written to the log file. The name of this file is defined by the string parameter *logfile* which defaults to the filename HOME$LOGFILE.CL. The name of the logfile may be changed by assigning a new value to the parameter, e.g.:

`cl>` **logfile = "commands.log"**

The important CL parameters which you may wish to alter or otherwise access are described in the table below.

CL Parameters		
parameter	*typical value*	*function*
echo	no	echo CL command input on stderr?
ehinit	(see manpage)	ehistory options string
epinit	(see manpage)	eparam options string
keeplog	no	record all interactive commands in logfile?
logfile	"home$logfile.cl"	name of the logfile
logmode	(see manpage)	logging control
menus	yes	display menu when changing packages?
mode	"ql"	default mode for servicing parameter queries
notify	yes	send done message when bkgrnd task finishes?
szprcache	3–4	size of the process cache

A full list of CL parameters can be obtained with the *lparam* command, or by typing the command **help language.cl**. The latter provides a brief description of each CL control parameter including references to *language* package manual pages containing more detailed information.

Changes that you make to any of the CL task parameters by assignment during a session will be lost when you log out of the CL. This is in contrast to the parameters of a normal task, which are learned by the CL. If you want the CL to "remember" values of CL parameters, you should initialize them to your personal default values in your LOGIN.CL file and they will be reestablished for you each time you log in.

5.2 Setting the Editor Language and Options

The parameter editor (**eparam**) command and the history editor (**ehistory**) both use the same simple set of edit commands and a choice of editor languages is available. The command:

`cl>` **set editor = emacs**

will set the edit mode for both editors to use the Emacs set of keystrokes. This also changes the editor that is invoked when you issue the **edit** command, so that all of the editor interfaces that are available will appear to operate in the same way.

Editor choices, with their associated key bindings, are:

- EDT (the default for VMS devotees)
- Vi (ditto for their UNIX counterparts)
- Emacs (which runs on either system)

For convenience, all of these editor choices support use of the cursor keypad keys and the DELETE key; the ambitious user may define his own personal set of command key bindings. The bindings that are available by default in IRAF are shown in an Appendix (§A.4). The default editor language that IRAF will start with is as shown above, chosen for compatibility with the host operating system. You may, of course, include a **set** command in your LOGIN.CL file to establish your own preferred editor.

The edit facilities provided within IRAF are limited in scope, since they are only intended to facilitate manipulation of user accessible internal structures, task parameter blocks and history file. IRAF has not implemented a full scale text editor, so the **edit** command invokes the standard system editor which you choose by setting the **editor** parameter. A host system editor must be used for all major text manipulations, but since it is invoked from within the IRAF environment the continuity of your session is not lost.

In addition to selecting the editor language to be used, there are a few user settable options available to control the operation of the *eparam* and *ehistory* tasks. These options are set by setting the string values of the CL parameters *epinit* and *ehinit*. For example, setting the *verify* option for *ehinit* will cause the history mechanism to pause waiting for a command to be edited or inspected, before executing the command. Read the manual pages for the *eparam* and *ehistory* tasks for a full description of these control options.

5.3 The History Mechanism

The CL history mechanism keeps a record of the commands you enter and provides a way of reusing commands to invoke new operations with a minimum of typing. The history mechanism should not be confused with the logfile; the history mechanism does not make a permanent record of commands, and the logfile cannot be used to save typing (except by using the editor on it after the end of the session). With the history editor, previous commands can easily be edited to correct errors, without the need to retype the entire command.

The **history** command is used to *display* command lines. By default, the last 15 commands entered are printed, each preceded by the command number. To show the last *n* commands, add the argument **n** to the **history** command line:

```
cl> history 3
101 urand 200 2 | graph po+ marker=circle szmarker=.03
102 help graph | lprint
103 history
cl>
```

and note that this number (*n*) will become the new default. If you ask for a negative number of commands (-*n*), the default will not change.

The **history** command allows previous command sequences to be displayed, but a related mechanism must be used to re-execute, or to edit and execute, commands. You can use the history file *editor* by issuing the command **ehistory**. Once you are in the history editor, the cursor (arrow) keys can be used to move about in the history file. You may select any command and edit it using the simple edit commands described previously (§3.2.3) for the *eparam* task. Such functions as deletions and insertions of words or characters, delete to end of line, and a simple string search and replace capabilities are provided. The Appendix lists the full range of commands that are supported. The edited command is executed by hitting RETURN. Note that it is a *new* command and, as such, it is appended to the history file. The current contents of the history file are not changed.

It is possible to recall *individual* commands and edit them; the special character '^' or the **ehistory** command may be used for this. Given the history record sequence shown above, any of the following commands could be used to fetch command 101:

```
cl> ^101               # fetch command 101
cl> ehist -3           # fetch third command previous
cl> ^ur                # fetch command starting with "ur"
cl> ehist ?mark?       # fetch command containing "mark"
```

The history command **^ur** finds the last command *beginning* with the string "ur", while the command **ehist ?mark?** finds the last command *containing* the string "mark" (the trailing '**?**' is optional if it is the last character on the line). A single '^' fetches the last command entered. Successive '^' commands will fetch the next preceding command lines from the history file.

The selected command is echoed on the screen, with the cursor pointing at it. At that point, the command can be executed just by typing RETURN, or it may be edited. The standard set of editor operations also apply when you edit a command in single line mode. Note that compound statements (those enclosed in pairs of braces "{ ... }") are treated as a *single* statement by the editor. Only one command line (which may be a compound statement) can be edited at a time with the history editor.

Sometimes you will want to reuse the *arguments* of a previous command. The notation '^^' refers to the *first* argument of the last command entered, '^$' refers to the *last* argument of the command, '^*' refers to the whole argument list, '^0' refers to the taskname of the last command, and '^N' refers to argument N of the last command entered. Thus,

```
cl> dir lib$*.h,home$login.cl
cl> lprint ^^
```

displays a table of the files specified by the template, and then prints the same files on the line printer.

One of the most useful features of the history mechanism is the ability to repeat a command with additional arguments appended. Any recalled command may be followed by some extra parameters, which are appended to the command. For example:

```
ut> urand 200 2 | graph po+
ut> ^^title = '200 random numbers'
urand 200 2 | graph po+ title = '200 random numbers'
```

in this case, the notation '^^' refers to the last *command* entered. The notation is unambiguous because the '^^' appears at the start of the command line. Do not confuse it with the use of '^^' to reference the first argument.

5.4 Foreign Tasks

The foreign task mechanism provides an alternative to the OS escape mechanism for sending commands to the host operating system. The advantage of the foreign task mechanism is that it allows foreign commands to be made available within the IRAF environment just as if they were normal IRAF tasks. Such commands may be abbreviated, their output may be redirected or piped, the commands may be run in batch mode, and the argument list is parsed and evaluated by the CL, hence may contain any valid CL expression. Users should beware, however, that IRAF virtual filenames appearing in the argument list of a foreign task are not normally translated to their host equivalents, since IRAF knows nothing about the argument list of a foreign task (the *osfn* intrinsic function may be referenced in the argument list to explicitly perform the translation, if desired).

To declare several foreign tasks with the same names in IRAF as in the host environment, use the following form of the *task* statement:

`cl>` **task $mail $grep = $foreign**

This declares the new tasks *mail* and *grep* in the current package, whatever that may be. If the current package is subsequently exited, the task declarations will be discarded.

To declare a foreign task with a more complex calling sequence, use the following form of the foreign task declaration:

`cl>` **task $who = "$show users"**

This example would be used on a VMS host to map the IRAF foreign task *who* to the VMS command **show users**. If there are any arguments on the command line when the task is called, they will be converted to strings and appended to the command prefix given.

The LOGIN.CL file contains a default USER package containing examples of several foreign task statements which may prove useful on the local host. Users should feel free to modify or extend the USER package, since it is provided with that in mind and provides a convenient structure for personalizing the CL environment.

5.5 Cursor Mode

Whenever an IRAF program reads the graphics or image display cursor, the cursor lights up or starts blinking, indicating that the user should position the cursor and type a key on the terminal to return the cursor position, keystroke typed, and possibly a character string entered by the user, to the calling program. The user may also read the cursor directly, just as a program would. For example, the command

```
cl> =gcur
345.21 883.13 1 r
```

would read the graphics cursor, printing a cursor value string such as that shown noting the world coordinates of the cursor, the world coordinate system (WCS) of reference, the keystroke typed to terminate the cursor read, and the string entered by the user if the key typed was **:** (colon).

The CL is said to be in *cursor mode* whenever the CL is waiting for the user to type a key to read a cursor. Cursor mode reserves the upper case keystrokes for itself, providing all sorts of useful functions to the user via the reserved keystrokes. For example, the graphics display can be zoomed or panned, a hardcopy of the current screen can be made on a hardcopy device, or the screen can be saved in or restored from a graphics metafile. For more information on cursor mode, type **help cursors** while in the CL.

5.6 Background Jobs

The CL provides facilities for manipulating and displaying data and allows interactive development and use of data analysis functions. However, many fully developed image analysis scenarios are very time consuming and need not be run interactively. IRAF allows such functions to be developed interactively and then processed in a batch mode as a background task, thus freeing the terminal for other interactions once the background tasks have been started. Several background tasks can be running at once, and these may be identical tasks that are just operating on different data sets.

Any command, including compound commands that may involve calls to several tasks, may be executed in the background by appending the ampersand character '&' to the end of the command block. The CL will create a new control process for the background job, start it, display the job number of the background job, and return control to the terminal. Background job numbers are always small integers in the range 1 to n, where n is the maximum permissible number of background jobs (typically 3-6).

```
pl> contour m92 dev=stdplot &
[1]
pl>
```

If a task runs to completion, and if the CL **notify** parameter is enabled (the default), the message "`[n] done`" will be printed on your terminal when the task completes.

Jobs running in the background may use all of the commands and perform any of the operations that interactive tasks can, but extensive user interaction with background jobs is necessarily somewhat limited (and not too appropriate). Another difference is that background jobs *do not* update parameter .PAR files. This is done to minimize the confusion that could occur if a background job asynchronously updated the parameter set for a task that was running interactively, or vice versa. The implication of this is that parameter values that are to be output by a task running in the background must be explicitly written into a file if they are to be available outside that job. Parameters passed between tasks in the same job are still processed correctly.

If the background job writes to the standard output, and the standard output has not been redirected, the output of the background job will come out on your terminal mixed in with the output from whatever else you are doing. Since this is generally not desirable, the STDOUT (and STDERR) for the background job should probably be redirected to a file and perused at a later time. The following example computes image statistics and directs these, and any error messages, to the file STATS.TXT:

```
im> imstatistics m87 >& stats.txt &
[2]
im>
```

If during the processing of a background job, the job finds it necessary to query for a parameter, the message

```
[1] stopped waiting for parameter input
```

will appear on your terminal. It is not necessary to respond to such a request immediately; when a convenient point is reached, respond with:

```
cl> service 1
```

The prompt string from the background job will be printed, just as if you were running the job interactively. Respond to the query and the background job will continue executing. If you do not respond to the request for service from a background job, it will eventually time out and abort.

More control over the disposition of a batch job is possible by appending optional arguments to the **&** at the end of the command line, when the job is submitted. The default action if no arguments are appended is to run the job as a subprocess of the CL, at a priority level one less than the CL, with output coming to the terminal unless redirected. To run the job as a subprocess at a specific priority, a numeric string specifying the *host dependent* priority level may be added after the **&**. For example,

```
cl> bigjob &4
```

will submit the job at host priority level 4. The priority level may also be specified relative to the CL priority in a machine independent way, e.g., **&-1** will submit the job at a priority level one notch down from the current CL priority (this is the default).

On systems which support batch queues (e.g., VMS) jobs may also be submitted to a batch queue. To submit a job to a batch queue, simply add the name of the queue after the **&**, e.g.:

```
cl> bigjob &fast
```

will submit the job to the "fast" queue. IRAF supports three logical batch queues, the **fast** queue, for short jobs to be run at a high priority, the **batch** queue, for medium size jobs, and the **slow** queue, for big jobs that may run a long time. The host system name of the desired queue may also be given. If a big job is submitted to a high priority queue it will be killed by the system when it exceeds the maximum quota permitted for that queue; see your system manager for more information on the batch queues supported by your system.

Sometimes it is desirable to wait for a background job to complete before resuming interactive work. For example, you might reach a point where you cannot proceed until the background job has finished writing a file. The **wait** command is used to wait for currently running background tasks to complete.

`cl>` **wait 1; beep**

will halt the interactive session until background job 1 completes. Issuing a **wait** command without a job number will cause the interactive session to wait for *all* background jobs to complete.

In order to discover the status of all background jobs that you have running, the command:

`cl>` **jobs**

may be used. The job number will be displayed along with information about the command that was used to start the job. The command **spy v** may also be used. It will request the host operating system to display the processor status (in an OS-dependent form), including information on the status of all processes running on the system.

There are important differences in the behavior of background jobs on different IRAF host systems. Under UNIX, the background tasks are independent of activities that may (or may *not*) be going on interactively. UNIX users may terminate their IRAF session and even logoff the UNIX system altogether, and the background jobs will continue merrily along. In the VMS implementation of IRAF, background jobs may run either as sub-processes or as regular VMS batch jobs in one of the system wide batch queues. The default is to run background jobs as sub-processes, in which case the jobs will be killed if you log out of VMS (even if you have DETACH priviledge). Under *both* systems, once the interactive CL session is terminated, communication with still-running background jobs *cannot* be re-established, even by re-entering the CL.

5.7 Aborting Tasks

Any interactive task may be aborted by typing the interrupt sequence CTRL/C. Control will return to the point at which the last interactive command was entered. When an IRAF program run from the CL is interrupted, it will usually perform some cleanup functions, deleting partially written files and so on. If an error (or another interrupt) should occur during error recovery, IRAF will issue the following message:

```
PANIC: Error recursion during error recovery
```

A panic abort is usually harmless, but may result in some half-written dregs of files being left behind. A more serious problem occurs when a subprocess becomes hung (uninterruptable). Repeatedly interrupting the CL when this occurs will eventually cause the CL to give up and shut down, necessitating a restart. A quicker solution might be to use the host system facilities to forcibly kill the subprocess.

The **kill** command may be used to abort a background job. The argument is the logical job number printed by the CL when the background job was spawned. (It may also be a list of jobs to be killed.)

```
cl> kill 1
cl> kill 1 3
```

In systems that support batch queues as well as sub-processes, the **kill** command may be used to control these as well.

NOTE

The remainder of this document is from the original draft and has not yet been brought up to date and may contain minor inaccuracies or omissions.

6 The CL as a Programming Language

All of the examples that have been presented thus far treat the use of the CL as a *command language* for running existing tasks. The CL can also be used as a high-powered desk calculator, one that can operate on and display arrays of data as well as scalars; and that can be fully programmed. The following sections introduce the programming oriented functions that are provided in the CL as background for understanding the creation of new user tasks.

Extensive use of the CL as a programming language has not been heavily emphasized, because there are substantial performance penalties associated with the use of interpreted languages, especially when dealing with large amounts of data. At the same time, the availability of an interactive environment that allows easy exploration of alternative analysis scenarios is very attractive, since it largely does away with the typical development cycles of edit; compile; link; test; edit; compile; .... Interactive development provides immediate, possibly visual, feedback about the effect of the various analytical tools upon your data.

Before delving into the details of the language however, a comment is in order regarding the distinction made in the CL between **command mode** and **program mode**. *Modes* in user interfaces are not in vogue because of the potential source of confusion, the "How does that command work now?" problem. IRAF is a little schizoid in this regard because of the desire for convenient user commands on the one hand: (to minimize the need for specific command parameter delimiters, quotes around character strings and special handling of file names and meta-characters); and the desire for a familiar language syntax for programming type activities on the other.

To resolve this dilemma, the CL has two modes: **command mode** - which is the default and is used for most terminal interactions; and **program mode** - which is:

- Entered within the body of a procedure.
- Entered within parenthesized expressions.
- Entered on the right-hand side of an equal sign ('=').

The syntax of the CL2 programming language was chosen to be as compatible as possible with SPP, the portable language in which most of IRAF is written.

Aspects of the command/program dichotomy have already crept into the discussions of identifiers, which are treated as character strings in command mode (§4.3.1), but which will be evaluated as a parameter name if thay are enclosed in parentheses. In the same vein, when inserted in a parenthesized expression, an identifier that is handled as a character string in command mode will be treated as a variable name unless it is quoted.

In program mode there is a simple disambiguating rule that can be safely used: always quote character strings and always parenthesize expressions. While this resolves the ambiguity it is not likely to be too popular with most users, who typically choose a minimum entropy approach. In the following sections, where programming issues come under discussion, programming mode will be assumed as the default.

6.1 Expressions in the CL

The CL has a conventional modern expression syntax (borrowed heavily from C and Ratfor) which should feel familiar to most users. The following operators are provided, presented in order of precedence:

Operator	*Action*
**	exponentiation
*, /	the usual arithmetic operators
+, −	and the rest of them in precedence order
//	string concatenation
&, ||	and, or
!	not
<, <=	less than, less than or equals
>, >=	greater than, greater than or equals
!=, ==	not equal, equal (2 equal signs)

Parentheses may be used to alter the default order of evaluation of an expression. Quotes are not optional in expressions or anywhere inside parentheses; identifiers are assumed to be the names of parameters and strings *must* expressly be quoted using either single or double quotes.

The data types supported by the CL are *boolean, integer, real, char*, and several exotic types (*imcur, gcur,* and *file*) that are touched upon later in this section. Observe that although the CL has *no* complex datatype, operations on complex data is supported in the rest of the IRAF system, including the **SPP** language and interface libraries. Arrays of the regular data types of up to seven dimensions are supported. Explicit type conversion is implemented with the intrinsic functions *int, real*, and *str*, the last converting an argument of any data type into a string. Mixed-mode expressions involving integers and reals are permitted, the data type of the result is promoted to the type of the target of the assignment operator.

The CL provides a special statement, called the *immediate* statement, for evaluating expressions and printing the value at the terminal. The form of the statement is an expression preceded by an equals sign:

= *expression*

or, if you prefer, the more conventional and more general **print** command can be used with the same results:

`cl>` **print** (*expression* [, *expression*, ...])

6.2 CL Statements and Simple Scripts

This is not a language reference manual; nonetheless, you will find it helpful to understand a few of the more useful types of statements provided in the CL. We will not attempt to present a complete definition of the syntax of the command language, a compendium of basic statement types is listed in the Appendix. The preceding section introduced two statements, the *immediate* statement and the *print* statement. The *assignment* statement should also be familiar from previous examples.

Often we do not want simply to assign a value to a parameter, but rather we want to increment, decrement, or scale one. These operations can all be performed with assignment statements in the CL, using the assignment operators $+=$, $-=$, $*=$, $/=$, and $//=$. For example, to increment the value of a parameter, we could use the $+=$ assignment statement:

`cl>` **y += (x ** 2)**

This statement increments the CL parameter **y** by the value of the expression **(x**2)**. The same operation could also be done with the next statement, but with some small increase in typing effort.

`cl>` **y = y + (x ** 2)**

The advantage of having a shorthand notation becomes obvious when you contemplate doing arithmetic on a fully specified parameter name as in the next examples.

6.2.1 Assigning Values to Task Parameters

The assignment statement may be used to set the value of a parameter or a variable. Most parameters are local to some task, and a "dot" notation may be used to unambiguously name both the task and the parameter. Thus, the statement:

`cl>` **delete.verify = yes**

may be used to set the value of the *verify* parameter belonging to the task *delete*. Since verify is a hidden parameter, direct assignment is the only way to permanently change this option setting.

The task *delete* belongs to the *system* package. Since IRAF permits several packages to be loaded at the same time, if there happened to be another task named *delete* in the search-path, we would have to specify the package name as well to make the assignment unambiguous:

`cl>` **system.delete.verify = yes**

In the unfortunate situation of two tasks with the same name in different packages, we would also have to specify the package name explicitly just to be able to *run* the task:

`cl>` **system.delete** *files*

In most cases such name collisions will not occur.

The ability to have the same task names in more than one package has some very positive benefits however, in that a new package of tasks that has the same calling conventions as a standard one may be readily inserted in the search path. This allows new algorithms to be tested without impacting the standard system, and also provides the hooks whereby alternate implementations of existing functions (using an array processor for instance) can be dynamically linked into the system.

6.2.2 Control Statements in a Script Task

The CL provides *if, if else, while, for, next*, and *break* statements for controlling the flow of execution in a command sequence. These statements are quite useful for writing control loops at the command level. Other control statements (*case, switch*, and *default*), which may be familiar from C or RATFOR, are also provided to ease the programming effort. By way of example, to print the values of the first ten powers of two the following statements can be used:

```
 cl> i=1; j=2
cl> while (i <= 10) {
>>> print (j)
>>> j *= 2
>>> i += 1
>>> }
```

The second of these two statements is a compound statement; note that the prompt has changed to `>>>` to indicate this.

Consider the parenthesized argument list in the **print** command in the above loop. If the parameter (**j** in this example) were not enclosed in parentheses, the CL would interpret it as a string rather than a parameter, and would erroneously print "`j`" each time through the loop. Remember that the CL will interpret identifiers as a string if found outside parentheses, but as the name of a valid parameter or variable inside parentheses. If the CL cannot find the identifier in its dictionary an error message will be issued. To avoid nasty surprises like this, one should *always* parenthesize argument lists in loops and within script tasks, and make a habit of explicitly quoting items that are to be treated as strings.

The example uses the built-in CL variables **i** and **j**. A number of variables are provided in the CL for interactive use; the integer variables provided with the CL are **i,j,k**; the real variables are **x,y,z**; the string variables are **s1,s2,s3**; the booleans are **b1,b2,b3**; and a list-structure pointer called **list** is also provided. The CL also has the ability to define new variables interactively; which is discussed in section §6.4.

6.3 Intrinsic and Builtin Functions

The usual Fortran intrinsic functions (with the exception of the hyperbolic and complex functions) are provided in the CL, along with some others specific to IRAF. The other intrinsic and builtin functions are those like **set**, **if**, **while**, **case**; the data declaration and initialization statements; and the task and I/O control statements that are described throughout the body of this document, and are listed in Appendix A. The intrinsic functions *must* be used in a statement where the returned value is explicitly assigned or output in some way. To compute (and display) the value of **sin(x)**:

`cl>` = **sin(x)**

must be entered, just typing **sin(x)** by itself is an error. The names of intrinsic functions may be used in other contexts, e.g. as parameter names, but care is needed to avoid confusion.

Function	*Action*	*Example*
abs	absolute value	z = abs(x)
atan2	arctangent	r = atan2(y, x)
cos	cosine	x = cos(r**2)
exp	exponentiation	z = exp(3)
frac	fractional part of a number	i = frac(y)
int	convert input to integer	j = int(z*3)
log	natural logarithm of a number	x = log(z)
log10	base 10 logarithm of a number	y = log10(x)
max	maximum value from input	x = min(1,17.4,43)
min	minimum value from input	y = max(47,11,92.3)
mod	modulus	z = mod(x, *base*)
radix	radix to any base	y = radix(x, *base*)
real	convert input to real	x = real(i)
sin	sine	y = sin(3*r)
sqrt	square root	z = sqrt(x**2 + y**2)
str	convert input to a string	s1 = str(num)
stridx	index of character in string	i = stridx(s1, 'abc')
substr	select substring from string	s1 = substr(s2, 3, 7)
tan	tangent	x = tan(2*theta)

As examples of the use of these functions, try entering the following expressions and see if you can predict the results (the next sections have the clues):

```
cl> = (sin(.5)**2 + cos(.5)**2)
cl> = 2 / 3.
cl> = (mod (int(4.9), 2) == 0)
cl> = 'map' // radix (512, 8)
cl> = delete.verify
```

You may have been surprised that the result of the last example was `no`. This is because **verify** is a boolean parameter which can only take on the values *yes* and *no*.

CL++

All of the intrinsic functions in IRAF return a value, the builtin tasks currently do not. With the completion of changes to the CL that are now in progress, intrinsics and builtin tasks, and any user defined functions or tasks may return an optional value. Tasks can thus be called either in the FORTRAN 'subroutine' style or in the 'function' style, where a value is expected. If a task returns a value that is not assigned to a variable it will be silently ignored.

6.4 Defining New Variables and Parameters

The CL provides a default set of variables that can be used for scratch computations, and you may declare other variables as needed. Each task that is invoked, whether it is a CL script task or an external executable task, may have input or output parameters associated with it, and these may be defined within the package for the task or as part of the task itself. Data declarations for variables contain the item name and type information, and may also contain an optional initialization clause. Declarations for function parameters are identical to those for variables, but they may contain optional fields to specify prompt strings, to define a valid data range, or to enumerate a list of valid values.

The simplest data declarations define local or global variables. The statement:

```
cl> int int_var
```

defines a simple integer variable. If this command is entered at the root (`cl>` prompt) level, it will define a variable that is globally accessible to any other task that is executed. When the same statement is incorporated into the body of a script task (after the **begin** statement), it will define a local variable visible only to that script task or to any other task that is called by that script task. Variables declared within the body of a package definition are globally available to all tasks within that package, but will be removed from the set of addressable variables when that package is unloaded.

User-defined variables may be used just like any standard IRAF variables: in expressions, passed as parameters to other tasks, or displayed in a variety of ways. In addition, these variables may be initialized when they are declared:

```
cl> real e = 2.71828183
```

establishes a real variable and assigns it an initial value. This variable may be treated like a constant (as in this case) or may be re-assigned another value during the course of computations.

The formal *parameters* for a task must be defined if the CL is to provide any of the range checking or input prompting activities. In the absence of a declaration, or if one is provided that does not define these optional fields, only the name of the parameter will be used for prompting and no range checking can be performed. The simplest case of a *parameter* declaration looks just like the simple variable declaration shown above; but it must occur within a task body *before* the **begin** statement, or define a parameter that is named in the calling list for a task.

The syntax of a data declaration statement is:

```
cl> type   [= initializer [,initializer] ... ]
           {
           [initializer [,initializer] ... ]
           [opt_field=value [,opt_field=value] ... ]
                                                  }
```

where the valid data types for these declaration statements are shown in the following table:

Data Type	*Explanation*
int	integer (scalar and array)
real	double precision floating point (scalar and array)
char	character strings (scalar and array)
bool	boolean, yes/no (scalar and array)
file	file name
struct	special form of mutli-token string
gcur	graphics cursor struct
imcur	image cursor struct

Most of these data types should be familiar to you, but the IRAF **struct** is a special class of data element that is used to hold multi-token strings, mostly for input. It will be referred to again in the section on I/O facilities in the CL (§6.8). **Gcur** and **imcur** are both structs that return a multi-token result, namely a string with RA, DEC and a data value. List structured parameters, which are described in section §6.9 are typically declared as structs.

The optional fields in a data declaration are used to define the data ranges for checking, prompt strings, special file type information, or a list of enumerated items. Syntactically, the specification of these optional fields is treated like a special form of initialization; the valid field names are described in the following table.

Field Name	*Explanation*
mode	auto, query, hidden processing modes
min	minimum value for range checking
max	maximum value for range checking
enum	enumerated list of valid responses (mutex with min/max)
prompt	prompt string for undefined values
filetype	r, rw, w, x file type specification

All of these fields are optional, the **mode** defaults to **auto**; **min/max** range checking defaults to NULL; and **filetype**, which is valid only for files, defaults to read only (**'r'**). The enumerated type (**enum**), which names the specific responses that are acceptable, is mutually exclusive with range checking, which defines a continuum of values that will be accepted.

To declare an integer parameter, enable range checking for positive values and provide a prompt string, use:

`cl>` **int new_parm { min=0, prompt='Positive integer' }**

and as an example of an enumerated list of valid input values, consider:

`cl>` **char color {enum = 'red | green | blue'}**

which defines a default length character string that is initially undefined and that has an enumerated list of three valid input values. If you attempt to use the variable **color** before a value has been assigned to it, you will be prompted for a value. If you try to assign it a value other than one of those that are enumerated, an error is reported.

6.5 Declaring Array and Image Data

Variables and task parameters may be defined as arrays of any of the data types: int, real, char or bool. Arrays may have up to seven dimensions. Array and image data will be referenced identically in the future, but for now there are some differences that are worth noting. Images are treated as large arrays of data that are stored on disk, and it is the amount of data to be processed that determines the choice of storage mechanism. Images will typically be quite bulky; it is not unusual for a single image scene to involve five to ten megabytes of data. For this reason, image data are most efficiently stored in disk files, and operations upon the data are performed by buffering it into memory as needed. The *main difference* between an array of data and an image is that the image will be buffered on disk.

IRAF provides a default IMDIR directory that may be used for bulk image file storage by all users, but it also has facilities that manage the storing, copying, and accessing of such data sets for users who wish to store this sort of data in their own directories. The logical directory IMDIR is where the IRAF system will store your image data *by default*.

IRAF images will appear to be created in your local user directory, but in fact it is only the **image header file** which goes there. The bulk pixel data are put in a second file that is part of a temporary files system, configured and managed with large datasets in mind. Such **pixel storage files** are transparent to the user, but if you have a great deal of data, you may find it more efficient to set up your own directory on a temporary files system, and to redefine IMDIR accordingly. If one has a personal IMDIR, it is also convenient to save data on tape and later restore it to disk; the header files are usually small enough so that they need not be archived if the data is going to be restored within a week or two.

To declare an integer array of length 100, type:

```
cl> int iarray[100] = 100(0)
```

which also initializes the array **iarray** to zero. A two dimensional array with data range checking can be specified by:

```
cl> real rarray[50, 50] { min=0, max=100 }
```

This array could be defined as an image by using the following declaration to indicate the different *storage class* of the data:

```
cl> real image rarray[50, 50] { min=0, max=100 }
```

where the data in this example would be stored on disk.

The choice of whether data is to be stored in an array which is stored entirely in memory, or as an image on disk is up to the user. The choice should be predicated upon the amount of data that is to be manipulated, since speed and efficiency of operation will be better using the image mode for data arrays much larger than a few hundred items. At present, the statements that manipulate these two data forms are somewhat different, as is explained in the following section.

During the development of IRAF, the handling of images has been a prime concern, representing as it does the major computational and I/O load that must be accomodated. Image files currently may be created on disk, and there are image processing functions that know how to process this class of data. The IRAF packages *images, imred*, and *plot* currently handle image data. The specification of this processing is similar, but not identical to, the operations performed on array data. The next section discusses use of image data and arrays within the CL.

CL++

At the present time the **IMIO and DBIO** subroutine libraries are still undergoing design and enhancement. As a result of this effort, the processing of image type data is not yet in final form. Existing IRAF packages do support image processing and display functions and these will appear to be functionally the same after the development has been completed.

The SDAS packages also support image processing and display functions, but at this point in time the disk format of these two types of data is vastly different, and on this interim system, data from these two packages cannot easily be mixed. This incompatibility is to be rectified when the completed **IMIO and DBIO** libraries are available.

6.6 Processing of Image Sections

All IRAF programs which operate upon images may be used to operate on the entire image (the default) or any section of the image. A special notation is used to specify **image sections**. The section notation is appended to the name of the image, much like an array subscript is appended to an array name in a conventional programming language. Note that array or image section *index references* are integer only, but that the data may be of any valid type.

section	*refers to*
pix	whole image
pix[]	whole image
pix[i,j]	the pixel value (scalar) at [i,j]
pix[*,*]	whole image, two dimensions
pix[*,-*]	flip y-axis
pix[*,*,b]	band B of three dimensional image
pix[*,*:s]	subsample in y by S
pix[*,l]	line L of image
pix[c,*]	column C of image
pix[i1:i2,j1:j2]	subraster of image
pix[i1:i2:sx,j1:j2:sy]	subraster with subsampling

In the following examples, please note that the references to image sections are all enclosed in quotes. These are *required* by the present language syntax. As the note at the end of this section indicates, this rule is to be relaxed in future.

A limited set of coordinate transformations may be specified using image sections, but please observe that transpose is *not* one of them. The "match all" (asterisk), flip, subsample, index, and range notations shown in the table may be combined in just about any way that makes sense. As a simple example:

`pl>` **graph 'pix[*,10]'**

will graph line 10 of the image **pix**. To generate a contour plot of an 800-pixel square image subsampled by a factor of 16 in both dimensions:

`pl>` **contour 'pix[*:16,*:16]'**

To display the fifth $x - z$ plane of a three dimensional image named **cube**:

```
im> display 'cube[*,5,*]', 1
```

on frame 1 of the image display device.

CL++

The image processing sections of IRAF are undergoing further development, as was noted in the previous section. Currently only image data can be processed as shown in the previous examples. Future developments will remove the need for quoting the section specifications that identify the image portion to be operated upon. The functional specifications will not change substantially, nor will the syntax itself be changed in any important way, but the need to remember to quote image section references will be removed. Image data will continue to be stored on disk and passed among tasks by passing the *name* of the file rather than passing the data itself, as a matter of efficiency.

6.7 Array Processing in the CL

The processing of *array* data is handled directly in the CL, and data arrays may also be passed to script tasks and external tasks. The entire array will be passed, since the CL cannot yet handle passing of sub-arrays to other tasks. The operations on arrays are handled via implicit looping over the array expressions, and only some of the operations described in the previous section on image data are valid for array data. References to array data sections need not be quoted however, and the syntax is otherwise identical to that supported for images.

Given the declaration:

```
cl> real a[10], b[20], c[10,20], d[10,20,30], e[10,10]
```

the following expressions are legal:

```
cl> c = 10                      # sets all elements of c to 10
cl> a = c[*,1]                  # copies row 1 of c into a
cl> = b                         # prints all values of b
cl> a = b[1:10]                 # copies subrange of b into a
cl> c = d[*,*,1]
```

and the following expressions are illegal:

```
cl> a = c                       # different dimensionalities
cl> a = c[1,*]                  # different limits on assign
cl> a = b[11:20]                # different limits
```

In general, for an expression to be a legal array expression in the CL, all array references must either be completely specified (i.e. **d[1,2,3]**), or they must loop over the same set of indices (i.e. **a** = **b[1:10]**). Indices may be specified as just the identifier (**a** or with an asterisk as the index (**b[*]**), indicating the entire array; or as a subrange specified by *integer constants* separated by a colon (**c[3:5]**).

CL++
Future developments in the CL will eliminate most of the restrictions on array operation in the CL and will bring the syntax for operations on images and arrays into complete alignment.

6.8 Input and Output within the CL

The CL provides I/O processing for parameters, cursor input and graphics output, and controls communications to external tasks. The communications that appear at the terminal in the form of prompts for parameters, error messages, data range checking queries, and much of the other I/O is performed in ASCII for portability considerations. The data items that a user can input in response to a prompt may be: integer values; floating point numbers, with or without exponent; yes/no responses for boolean data; or character strings as appropriate.

Image data and other bulk data forms that are processed by the various functions are typically *not* passed through the CL itself, instead the names of the files are passed about, and temporary files are dynamically created as needed to hold intermediate results of computations. Cursor input from graphics and image devices are passed directly through the CL however, in order that they may be re-directed to a file like any other I/O stream. The **imcur** and **gcur** structs are used to handle this type of data.

As was mentioned in §2.4 and in the section on I/O and pipes (§3.3) the CL communicates via its standard input and output, which are ASCII streams normally connected to the terminal. The CL functions that manipulate these streams can also be used on data in a file, just as the CL itself can be driven with commands from a file. The control files are all ASCII data streams, with no implied structure, and thus are easy to construct and to edit, should that be necessary.

The **scan** and **fscan** functions are provided in the CL to process ASCII input streams; the only distinction between them is that **scan** operates on the terminal input stream (or its re-directed source) and **fscan** specifically operates on an file. **Scan** reads from the terminal (there is no prompt) and returns the first token it finds in its input stream; where a token is understood to be any string of alphanumeric characters delimited by a blank. Quotes are ignored and no other punctuation has any special meaning. If a $\boxed{\texttt{CTRL/Z}}$ is entered EOF is signalled and the following print statement will not be executed.

```
cl> if (scan (s1) != EOF)
>>>     print (s1)
```

```
>>> else
>>>     return
```

The **print** command takes whatever is passed it as a parameter and displays it on the terminal, so the previous example does a simple one-line *echo* function of input to output. The argument to **scan** in this example is the character variable **s1**, but it could as easily be an integer:

```
cl> while (scan (i) != EOF)
>>>     print ("Input was a ", i)
```

This in-line script will continue looping, reading from the input and echoing to the output, until EOF is signalled. All valid numeric inputs will be accepted; real input values will be truncated to integers; character constants (single characters) will be processed as though **int** had been called; and any invalid input values will be silently ignored. **Print** shows another of its features, that string constants may be directly inserted into the output stream and that *no format* specifications need be made.

The I/O functions can be used to process more than one data element at a time, with no need for explicit formatting. If more data is presented than there are identifiers in the list, the extra data is silently ignored; and if there are more data elements in the parameter list than there is data, the remaining data elements retain their old values.

The output or input for these functions can be explicitly redirected, via the usual mechanisms, or the **fscan** and **fprint** functions can be used instead. The commands:

```
cl> list = 'infile'
cl> while (fscan (list, s1) != EOF)
>>>     fprint ('junque', 'The next input line = ', s1)
```

perform exactly the same function as:

```
cl> list = 'infile'
cl> while (scan (s1, $<$list) != EOF)
>>>     print ('The next input line = ', s1, $>>$ 'junque')
```

where the list structured variable **list** has been set to the name of the file to be used for input (INFILE) and the output is being directed to file JUNQUE. These examples are hardly exhaustive, but will serve as background information for the discussion of list structured parameters that follows.

6.9 List Structured Parameters

For certain data analysis tasks, the ability to define a *list* of data files for batch processing can be especially useful. IRAF supports **list structured parameters** for specifying a list of items to be input to a function. Many, but not all, functions will accept this form of input. List structured parameters are associated with a file and have their own peculiar, but useful, semantics.

Suppose we want to make a series of contour plots on the standard plotter device of a set of data files. This can be done interactively by entering a command to produce each plot, but this is tedious. A better approach would be to prepare a list of image sections (see §6.6) to be plotted, naming one section per line in a text file (which we choose to call SECTIONS). The following command could then be used to generate the plots in the background:

```
pl> list = 'sections'
pl> while (fscan (list, s1) != EOF)
>>>     contour (s1, device = 'stdplot') \&
```

In this example, the assignment of **'sections'** to the parameter **list** has two actions, it associates the name of the file SECTIONS with the list structured parameter, and it causes a *logical* open of the file. The actual file open takes place the first time that **fscan** is called. Successive calls to **fscan** return successive lines from the file into the string **s1**. When the end of file (EOF) is encountered the **while** loop terminates. If additional calls are made to **fscan** EOF will continue to be returned. A logical reset to the top of the file can be performed by reassignment of the same file name to the parameter **list**, or another file name can be associated with this parameter.

A user can declare other list structured data elements besides the ones that are provided. The statement:

```
cl> struct *input = uparm\$inlist
```

declares a list structured variable named input that is bound to the list named UPARM$INLIST. This file may contain several records that define image section specifications or the names of other files to be processed. Once the declaration is made, lines may be scanned from the file, as in previous examples, or the records may be fetched by use of a simple assignment operator:

```
cl> = input          # displays the next record from file
cl> s1 = input       # sets s1 to the next record
```

Successive references to the identifier **input** will result in successive records being read from the file it is bound to. If an EOF is detected it is silently ignored, and the last value read will continue to be returned. List-structured identifiers may be integer, real, or character as well as struct type data. The binding to a filename is the same regardless of type, and the only difference is that data conversion is performed on the input record to match the type of the identifier.

If you expect to write commands much more complicated than these examples, it is time to learn about **script tasks**. This topic is covered in §7 of this document and more detailed information is given in the *CL Programmer's Guide*.

7 Rolling Your Own

The true power of the CL and the whole IRAF analysis environment lies in its ability to tailor analysis functions to the users' data. This power comes from the ability to define new functions in the CL, and from the capability to extend the basic IRAF system by the addition of new analysis packages. These new routines can be developed incrementally, a line at a time, and then defined as a new task once the function operates as desired. User defined functions and analysis packages look just like any of the standard functions that are provided with the system, and are called in the same way. All of the same parameter passing, range checking, and prompting operations of IRAF are available for user defined functions.

Beyond the ability of the CL to *learn* the value of parameters that you have entered, or the creation of "one-liners", little user customization has been presented so far. However, all of the programming and extensibility features of the CL that have been used in the creation of the standard packages are also available to the intrepid user, enabling the creation of a personalized data analysis environment, one's own set of *doit* functions. This section introduces the creation of new user script tasks, user executable tasks, and packages of such tasks.

Examination of one of the .CL script tasks that defines a standard package will reveal that it contains several **set** commands to establish logical directory names; and then defines the set of tasks that compose the package. Examine the script task for the **system** package of functions:

`cl>` **page system$system.cl**

to reveal the mysteries of these basic functions. The task names that appear in this package description contain a **logical directory** portion (as in **system$**) and a filename portion (**system.cl**). The logical directory name is separated from the rest of the file name by a dollar sign **'$'**, as was discussed in the section on virtual file names (§4.3). Use of a logical directory specification in public packages, and even in those for your own private use, is highly recommended, since it provides an unambiguous specification of where to find the package and tasks.

Note that tasks need not be defined as part of a package; individual tasks can be created and defined at any time, but a package is a convenient way of grouping related tasks together. Many package have already been provided in IRAF, and should be browsed by anyone who is searching for clues about how the CL language can be used for programming.

7.1 Creating Script Tasks

All of the IRAF commands can be used within a script task and will operate the same way in that environment as they do when entered interactively. A script task need be nothing more than a text file that contains normal CL statements or commands. Commands may be entered in a script just as they would from a terminal in **command mode**, or **program mode** may be used, in which case slightly different rules apply (c.f. §6.0). For simple process control scripts command mode is likely to be satisfactory, but program mode is the obvious choice if more complicated tasks are undertaken. The main distinction is that task names must be entered in full and the standard rules should be followed for variable names and character string references. Program mode will be used in the following examples, since it is most likely to be used in any complicated script tasks that you might wish to develop.

In order to create a script task one has merely to invoke the editor

`cl>` **edit** *taskname*.**cl**

and enter the CL statements that describe the actions you wish to have performed. When you have created the new script task (or modified an existing one), exit the editor in the normal way, so that the file is written in your current directory.

A script task for demo purposes might look like:

```
{
print(' Hello, world !! ')
}
```

In order to make this new task available to the CL, you will have to identify it and indicate where the script task is to be found:

`cl>` **task $my_new_task =** *taskname*.**cl**

Note that the name by which you refer to the new task need not be the same as the name of the file, although the use of the same name is conventional. The '**$**' in the **task** statement is optional and tells IRAF not to search for a parameter (.PAR) file for the task.

Once the task has been created and declared it may be directly invoked:

`cl>` **my_new_task**

will cause the script task file to be parsed and executed by the CL. You may change the body of a script task without redefining it with another **task** statement.

While tesing new script tasks, such as this one, you may find it useful to turn on echoing:

`cl>` **echo = yes**

which will cause the CL to echo the commands on the terminal as they are read from the script.

Since all of the commands that you have entered at the terminal are logged a the history file, it is possible to edit all or part of this command log to create a new script task. You will first have to output part of the history log to a file and then edit it:

```
cl> history 30, > temp
cl> edit temp
```

which lets you change history (not a bad trick). Once you have edited the file, the commands needed to turn it into a CL script task are the same as those described above.

7.2 Passing Parameters to Script Tasks

Parameters are used to control the operation of tasks by defining input and output files, indicating execution options, etc. Script tasks that a user defines may have parameters that operate in exactly the same fashion as standard tasks. In fact, the same prompting, learning, and limit checking mechanisms that operate for standard tasks are available by default for user script tasks, as well as for external user tasks (about which more in §7.5).

CL parameters and other variables that are defined external to a new script task may be referenced from within the task with no special action being taken on your part. Global variables and variables passed from higher level tasks are also accessible to a task. However, named parameters for the task, or variables that are local to the task (and thus protected from external actions), must be declared within the script task itself. Parameters are identified by being defined in the formal parameter list of the procedure or before the **begin** statement, while local variables are declared only *after* the **begin** statement. N.B. the **begin** and **end** statements must appear all by themselves on the line, and anything that appears *after* the **end** will be ignored.

The following simple script task description will serve to illustrate many of the salient points:

```
procedure doit (inparm, groups, region, outparm)

file    inparm {prompt = 'Input file name:'}
int     groups {prompt = 'Groups to process (0 for all):'}
int     region[] {0, mode=hidden}
file    outparm {prompt = 'Output file name:'}

begin
        file    cal_file = 'calib$wfpc'         # Wide Field Camera
```

```
        int     n_group, ngp

        n_group = groups                        # get the users group request
        if (n_group == 0)
            n_group = 9999

        for (ngp=1;  ngp <= n_groups;  ngp=ngp+1) {
            calib (inparm, ngp, cal_file) |     # note use of pipe
                clean() |
                clip (region= region, >> outparm)
        }
end
```

The identifiers **inparm, group, region,** and **outparm** are parameters of the function and are used to pass data into and out of the procedure proper. There is one *required* parameter, **inparm**, which is the input file name that contains the name of the file to be operated upon. The other parameters are **groups** the number of data groups to be processed; a *hidden* parameter, **region**, which will not be prompted for; and the parameter, **outparm**, which is the name of the file that is to be written by the function. The variable **cal_file** is local to the procedure and is only available within the procedure body (or to any lower level routines to which variables may be passed as parameters).

There are some subtleties here that bear mentioning. Hidden parameters, such as **region**, may be defined for script tasks and must appear *before* the **begin** statement, but need *not* appear in the formal parameter list. The **groups** parameter will be prompted for if not specified on the command line, but is then assigned to a local variable **n_group**. This is not required, but is desirable because of a side-effect of the automatic prompting built into IRAF. Any query mode parameter will be prompted for automatically, *each time it is referenced*. This can be very useful in a script where a new value is to be input on each execution of a loop, but can be surprising if one only intends to enter the value once. The obvious fix is to assign the value to a local variable (at which time the prompt will occur) and then operate only on the local variable.

As with the simple task described before, this procedure must be declared with a **task** statement in order that the CL be able to locate it.

cl> **task doit = home$doit.cl**

Remember to specify the logical directory in the declaration so that the task can unambiguously be located no matter which directory you use it from. When you run this new task, you will be expected to enter a value for the input parameters (and you will be prompted if you don't).

Remember that once you have used the **task** statement to inform the CL how to find your new function, you need not do so again during that session, since the original declaration of the task will allow it to be located even if you edit the body of the task to change its internal operation. If you need to change the number or type of parameters to a task, or to change the name of a task, move it to another directory, or if you wish to redefine the meaning of one of the standard tasks in the system, you will have to use the **redefine** command.

The following commands:

```
cl> rename home$doit.cl funcs$doit.cl
cl> redef doit=funcs$doit.cl
```

rename the file HOME$DOIT.CL to FUNCS$DOIT.CL and then redefines the *doit* task to point to the script. This redefinition causes the CL to reprocess the script task in order to re-establish pointers to the file and to redefine the data declarations.

Once you have tested your new task and have it debugged, you may wish to enter a **task** definition for it into your LOGIN.CL file, or to install it in your own package of private functions.

7.3 Using List Structured Parameters in a Script Task

It is possible to define a CL script task such that a sequence of files whose names are defined in an input list may be processed in one call to the task. This is convenient when a number of files need to be processed in an identical way. The following example shows how the task **doit_toit** has been setup to accept a list structured input parameter, and then to call another task, passing the files from the list one at a time to the other task. The obvious advantage is that the development of the task that really does the work, viz. **doit**, can be done in isolation from the mechanics of batch processing the list of data files.

The **doit_toit** function is set up for production use and, as such, it logs all activity in the standard logfile. It may be run as a background task, like any other IRAF function.

```
# DOIT_TOIT -- A driver task for batch operation of DOIT.

procedure doit_toit (images, section)

file    *images { prompt = 'List of images to be changed' }
char    section { prompt = 'Section of input images to be output' }

begin
        struct  imgs            # working storage for image file name
        bool    klog            # default size is 32 chars
        file    logf

        klog = keeplog          # get global keeplog flag
```

```
        logf = logfile          # ditto the log file name

        while (fscan (images, imgs) != EOF) {
            if (klog) {
                # Output a startup message if keeplog is true.
                print (' DOIT_TOIT: Process images ', >>logf)
                time (>> logfile)
                print ('    Images  : ', imgs, >>logf)
                print ('    Section : ', section, >>logf)
            }

            # Do the actual task by calling the DOIT function, passing
            # in the file name with the section concatenated.

            doit (imgs // section, imgs)

            if (klog) {     # output the trailing message
                time (>>logf)
                print (' DOIT_TOIT: Completed. ', >>logf)
            }
        }
end
```

The declaration for the variable **images** needs some explanation. The asterisk **'*'** indicates that **images** is a **list structured parameter**, i.e. that it is to be processed as a pointer to a parameter list, rather than as the parameter itself. In this instance it will contain the name of a file that itself contains a list of the names of files to be processed.

As with the other script tasks that have been described, this one must be declared to the CL via a **task** statement before it can be executed. Once this has been done the task can be called as follows:

`cl>` **task doit_toit = home$doit_toit.cl**	
`cl>` **doit_toit ('images.txt', '[]')**	# no sub-sections
`cl>` **tail logfile**	# check the logfile messages

The name of the file containing the list of images "IMAGES.TXT" is passed directly into the task as a quoted string.

7.4 Establishing Your Own Function Package

Once you have defined several functions that do a useful set of operations, you may wish to set them up so that they are always available to you. This can be done by defining them as a package, which is the mechanism that IRAF uses to organize the other groups of tasks that are made available as part of the system proper. (Or, more easily, by putting the **task** declarations into your LOGIN.CL file.)

```
package my_package

set  funcs = "home$func/"          # define the logical directory

task    fib   = funcs$fibonacci.cl
        glib  = funcs$wordy.cl
        doit  = funcs$doit.cl

clbye()      # invoke the cl again for interactive use
```

If you now place the declaration for this package task in your LOGIN.CL file, these tasks will be available to you whenever you login to IRAF. It is a good practice to always use logical directory names in task declarations and in other file names, since changing the search path with a **chdir** may otherwise render tasks not locatable.

Packages may, of course, be more complex than the above, since they can refer to several logical tasks that may be CL script tasks or executable tasks. User packages may also reference logical tasks that are in other packages directly, if only one or two such tasks from another package are needed. Additionally, a package task may declare variables that are to be treated as global to all of the tasks *within* the package, but are to remain local to the package itself. Other IRAF commands can be included in a package task as well, such as a load request for a package of utility routines. As you will recall, the load operation for a package is implicitly executed whenever a package is named; thus, a utility package such as *plot* or *imred* can be loaded and made available just by naming it in the package description.

7.5 Creating Fortran, SPP and other External Tasks

While the IRAF and SDAS applications packages, along with the CL language, offer substantial facilities for interaction with and analysis of data, it would not be unusual for users to wish to make their existing algorithms available for use in IRAF. Neither the IRAF packages nor the SDAS packages can provide all of the functions that everyone might desire. IRAF has been developed as an *open system*, and is therefore extendible by the user so that externally compiled programs are accessible with the CL. These programs may be coded in any language that conforms to the standard calling conventions, but Fortran, C, and the Subset PreProcessor (SPP, the IRAF portable language) have predefined sets of interface routines that establish access to the IRAF **kernel** functions.

It is suggested that the Fortran programmer use the SDAS/ST interface routines to access the IRAF environment. These routines provide access to parameters, perform I/O on image files as well as other data file formats, provide facilities for header data manipulation, etc. The routines are described in the *SDAS Applications Programmer's Guide* (the *Green*

Book) and in the *Software Interface Definition, ST-ECF O-11*, which describes the set of interfaces that have been agreed upon between the STScI and the European Coordinating Facility (ECF).

The SDAS interface routines (and the ST-ECF interfaces) are all built upon the existing IRAF kernel interfaces as described in the *IRAF Programmer's Crib Sheet*. These interfaces are rather more complete than the SDAS/ST ones, providing full access to the facilities of the IRAF virtual operating system. These routines can be called directly from SPP, but typically *cannot* be called directly from Fortran programs.

A selection of software development tools are available to the user who wishes to integrate personal programs into the IRAF environment. These utilities are provided as parts of the *softools* package, and include such functions as IRAF-specialized compilation and linkage, help file creation, etc.

A simple SPP language routine is included here as an example that can be tried from the terminal. It is not a particularly useful function (it just feeps the terminal bell), but does show the ease with which an external task can be linked into the environment.

```
#-------------------------------------------------------------
# Simple task to show use of SPP external procs
# Compile and link using SPP command in Softools

task feep = t_feep
include <chars.h>           # include the standard char defs

# FEEP -- Feep the terminal.

procedure t_feep()
begin
        call putc (STDOUT, BEL)
end
```

After you have used the editor to create the program source file, you should load the **softools** package that contains the **xc** compile/link tool. This will compile the program, link it, and create an executable named FEEP.E.

> `cl>` **softool**
> `so>` **xc feep.x**

Once the **feep** task has been compiled and linked it may be made available from within the IRAF environment in a fashion analogous to that used for CL script tasks. A **task** statement that names the executable file and binds it to a task name must be used to declare the task to the CL. Once this has been done, the executable task can be invoked just like any other task.

```
cl> task feep = feep.e
cl> feep                                    # is used to call it
```

N.B. IRAF tasks written in SPP (and other languages) may contain more than one logical task, and the CL **task** statement must be used to declare each of them. The logical task names used on the CL **task** statement *must* correspond with the task names from the `task` statement used in the body of the SPP file.

The parameter handling facilities at the user level behave identically for executable external tasks and for CL script tasks. If the complete facilities of parameter range checking, prompt messages, and default values are desired, a data declaration statement that defines these values will have to be specified. If a declaration statement is not provided, the standard prompt processing (naming the parameter and soliciting input) will be used when parameters are referenced.

The details of parameter and file I/O in SPP, and in the Fortran and C interfaces, are sufficiently different from those presented for the CL that the entire subject is best deferred to a separate document. It is important to note however, that the facilities that are available in the CL and in the IRAF kernel routines that support it are equally available to all executable modules. The implication of this is that any program that has been linked into IRAF will have the same apparent interface as all of the existing programs, and thus can easily be made to appear as a unified part of the larger system. This can, of course, be subverted, and programs may maintain their own identities in so far as that is desirable. However, the advantages of a unified user interface and of a well defined and completely implemented set of flexible interface routines cannot be stressed enough.

8 Relevant Documentation (the Yellow Pages)

This document should serve to get the first-time user started (and if it doesn't, we would like to know about it), but there are many topics that have been covered quickly or not at all. Other documents and sources of information about IRAF, SDAS, standard packages, use of SPP, IRAF internals, etc. exist and this section will provide pointers into that realm for those who would like more information on these topics.

8.1 IRAF Command Language

Some of the documents describing the CL are now somewhat out of date since there have been at least two revisions to IRAF since they were written. However, these documents are readable and are still useful, since, by design, most of the changes made to the CL are compatible with what had gone before.

CL Programmer's Guide, in preparation

This is to be the most complete introduction to programming in the CL and to the facilities of the CL. As such, it provides details of language use (especially helpful when developing external and/or script tasks in Fortran, SPP, and C). Also included are descriptions of the workings of the CL, the IRAF kernel and inter-process communications as they affect the use of IRAF for program development.

Detailed Specifications of the IRAF Command Language, rev Jun82

This paper discusses in technical detail the major functions of the CL and some details about how it works. Since this is a specifications document, it is not as readable as a user document, but it does cover many of the same areas as the *CL Programmer's Guide* in somewhat more detail.

8.2 IRAF Applications Packages

Much of the richness of the IRAF environment comes from the packages of applications programs that are being made available within the environment. The IRAF group at NOAO and the SDAS group at STScI have been developing analysis programs that are available as packages within the IRAF environment, and the end-user-oriented documentation of these systems is described below.

IRAF Applications Packages, Structure and Requirements, Aug83

This document is an attempt to define fully the decomposition of the IRAF system and applications software into packages. The functions performed by each package are summarized in the form of requirements.

Descriptions of specific IRAF applications packages developed at Kitt Peak are available in a set of user-handbooks :

- *APPHOT - Digital Aperture Photometry, Aug83*
- *GRAPH - Simple Graphics Routine, Mar84*
- *SURFACE - 3-D Surface Display, Mar84*
- *CONTOUR - Contour Map Display, Mar84*
- *HELP - On-Line HELP for IRAF, Mar84*
- *DATAIO - Data Readers and Writers, Mar84*
- *LISTS - Basic List Processing Functions (ASCII files), Mar84*
- *UTILITIES - Miscellaneous Utility Functions, Mar84*
- *SOFTOOLS - Software Utilities, make, yacc, xc, mklib, Mar84*

Science Data Analysis Software Requirements, Final, Aug82

These are the contract requirements of the SDAS system.

SDAS User's Manual, in preparation

A descriptive guide to the use of SDAS.

8.3 Standard Program Interfaces

There are three sets of interface routines available to the programmer: those for SPP, those for C, and those for Fortran. The language that you use depends on the nature of the project being undertaken, your own level of expertise, and on the need for creating portable code. SPP is the language of choice for packages that are to become part of IRAF or for tasks that require access to the entire virtual system interface. Fortran and the SDAS/ST interfaces will remain the choice for existing Fortran codes and for many small scientific applications programs. Users are encouraged to choose the SDAS/ST interface for their Fortran programs. C has been used for the innards of the CL itself, and a set of interfaces (the LIBC library) is provided that emulates the UNIX standard I/O facilities and gives access to the IRAF kernel facilities.

A Reference Manual for the IRAF System Interface, rev May84

This document is the most complete (and recent) treatment of the linkages between the portable IRAF kernel, the CL, the external procedures and the system dependent Z-routine layer. It describes these interfaces in detail and has the complete specifications of the Z-routine interfaces. It is of particular use to both the individual who is trying to port IRAF to another system (it is a *must read* for such persons) and to the system or applications programmer who wants a more detailed understanding of IRAF.

8.3.1 SPP Interfaces

Programmer's Crib Sheet for the IRAF Program Interface, rev Sep83

This document describes the complete set of interface functions for the IRAF virtual operating system as it is available to the SPP programmer. Several sets of library functions are described for accessing images and various kinds of data files, terminal capabilities, graphics and image I/O, etc. Programs written using only these interfaces will be fully portable, along with the main body of IRAF code.

Reference Manual for the IRAF SPP, rev Sep83

This is the definitive document about the IRAF Subset Preprocessor language. The language is somewhat like Ratfor (from which it derives), but it has extensions for dealing with the memory management and referencing issues that arise when operating on large scale image data.

The Role of the Preprocessor, rev Dec81

This document is an early discussion of the philosophy behind use of the SPP language. As such, it is valuable background reading for anyone who wishes to understand fully the benefits of using a preprocessed language for implementing a large body of portable code like IRAF.

8.3.2 Fortran Interfaces

SDAS Applications Programmer's Guide

This is the complete description of the interface set that is now being used to develop the SDAS program suite at STScI. These interfaces are expected to be replaced eventually by the set of SDAS/ST interfaces mentioned in the following document.

Software Interface Definition, ST-ECF O-11, rev Aug84

This describes the set of standard interfaces agreed upon between the STScI and the ST-ECF. It incorporates most of the features of the SDAS standard interfaces and is to be used for program development at STScI and for future MIDAS programs developed at ESO. It is a rather complete interface for image data, process parameters, and table data structures, but does leave out many of the other facilities provided in the complete IRAF SPP interface.

9 And into the Future

This version of IRAF and its CL represents the first external release of a system that has been under development for more than three years, and which will continue to evolve for several more. IRAF was born out of the need for a system that could survive changes in operating systems and hardware, since such changes are a normal part of computer system evolution. To accomodate such changes, and to provide a level of stability and portability across operating systems, IRAF is a *layered* system: the CL is the user layer; the kernel is the operating system independent layer; and the z-routines are the system dependent layer.

Most of the discussion in this document describes the appearance and function of the current command language (CL2), the user layer of IRAF. As has been noted at various points in the text, the CL is still undergoing development and is expected to change over time. (The paragraphs marked **CL++** in the text hold the clues.) The intent in these changes is two-fold:

- Provide evolutionary enhancements to the user interface to improve utility, functionality, usability and coherence.
- Bring the programming elements of the CL language into line with the SPP language which is used for the bulk of the IRAF portable code.

These requirements are somewhat at odds, as was noted in §6, but further attention is being given these issues to try and resolve the dilemma. Although not all elements of the CL language can be mapped easily into the SPP programming language (consider the I/O control commands), the notion that one can do development in an interactive environment and then realize compiled code speeds for execution is an attractive one. The CL2 implementation has taken steps in this direction and we expect to see how far this idea can be taken over the next few years.

9.1 Near-Term Software Projects

While the evolution of the CL is an important part of IRAF, there are other elements of the system that are even more important which are still under development at this time. The most important single area for development is the database and data catalogue. Design and prototyping of these important functions, and the necessary query language, is in progress right now. These functions are doubly important, since they both represent a user facility for creating and operating on private catalogues of data; and also are at the heart of the effort to merge the SDAS functions cleanly into the IRAF control structure. Only after the DBIO and IMIO packages have been fully implemented and integrated into the rest of the system will functions from these two large applications groups be able to share data and file structures. A tables system is also envisioned that will offer similar capabilities to the MIDAS tables, but will be built upon the DBIO routines.

The areas of graphics and image display will also receive attention during the next year. The current GIO package supports a fast kernel and both a GKS and an NCAR emulation. However, image data are not merged into this structure and no common meta-file formats exist that allow image, annotation, and overlay graphics to be represented in one consistent form. Some efforts have already been made toward understanding what needs to be done and how (or if) the requirements can be satisfied within existing standards. Further efforts will be made, both at NOAO and STScI, and at other facilities like RAL who have expertise in this area, to develop an approach that satisfies the requirements.

Developments in the area of networks, both LAN and wide-area; workstations; and the related topics of access to distributed data bases and archives are also receiving attention. We believe that IRAF has a sufficiently robust and flexible structure that it can operate successfully within a distributed operating environment. Small scale efforts are now underway to explore some of the issues related to network file systems.

At the same time, a prototype project is underway with a single platter laser disk, to evaluate the suitability of this media for large scale, long term archival storage of images and other related data. As a part of this activity, IRAF is currently being ported to a SUN workstation and high resolution image display at the STScI. IRAF and its database package will be important to this effort, since they will provide some of the basic facilities for processing the data and managing the catalogues for this archive.

These are all components of a distributed data system, a long range goal for the Space Telescope community, and probably of interest to the entire astronomy and astrophysics community. To the extent that IRAF proves useful to astronomers, it will play a key role in such a data system.

9.2 Where Is the Future?

The usual rapid pace of developments in hardware and software systems will also prompt consideration of changes to the external appearance and some of the internal implementation details of IRAF. Developments in graphics and image displays, and in various data input and storage devices promise to make revolutionary changes in the way that people and computers interact. High resolution graphics, touch screens, mice, WYSIWYG (What You See Is What You Get), DWIM (Do What I Mean), and the 'cluttered desk' paradigm pioneered by Xerox PARC and emulated by everyone else, will all appear in one form or another in a variety of different systems in the next few years. These presentation techniques and differing interactive views of data, when well thought out and integrated into a system, can serve to make computers more accessible to people. These ideas, however, have been much slower to arrive in the large-scale systems that have been used for data analysis than in the super-micro and PC class machines.

IRAF, in its current version, incorporates only some of these elements, but many others will be experimented with as it is ported to different hardware environments. Because the CL is a separable piece of the system, it can be changed or replaced, without necessarily making major changes to the underlying system structure. Efforts now underway to move IRAF out of the super-mini systems where it has been developed, and into super-micro workstations will afford the opportunity to explore some of these user interface issues in a prototyping mode. Depending on the results of such experiments, other CL interfaces that take advantage of those elements that prove successful are likely to evolve.

These statements should not be construed to mean that constant change will be the norm. IRAF was designed to protect the substantial software investment that any large data analysis system represents, and this it will do, both for the developers and for the users. The IRAF kernel and the layered interfaces for applications programs are quite stable, and are not expected to change, except to incorporate additional functionality. Furthermore, any changes proposed for the user interface will be carefully evaluated in terms of their impact on the existing user community. But, just as we expect that faster, more efficient FFT or filtering algorithms would receive a welcome reception we expect that a stable, but slowly evolving system that continues to serve users needs will meet with approval. Feedback and commentary from the users of the system will be vitally important in this development process, and we encourage that dialogue.

A Appendices

A.1 CL Commands and the System Package

A.1.1 CL Intrinsic and Builtin Functions

access	-	Test to see if a file exists
bye	-	Exit a task or package
cache	-	Cache parameter files, OR
		Print the current cache list (no arguments)
cd	-	Change directory
chdir	-	Change directory
cl	-	Execute commands from the standard input
clbye	-	Exit a task or package to save file descriptors
defpac	-	Test to see if a package is defined
defpar	-	Test to see if a parameter is defined
deftask	-	Test to see if a task is defined
ehistory	-	Edit commands from the history log file
envget	-	Get the string value of an environment variable
eparam	-	Edit the parameters for a function
error	-	Print error code and message and abort
flprcache	-	Flush the process cache
fprint	-	Format and print a line into a parameter
fscan	-	Scan and format an input list
hidetask	-	Define a new hidden task
history	-	Print the last few commands entered
jobs	-	Show status of background jobs
keep	-	Make recent set, task, etc. declarations permanent
kill	-	Kill a background job or detached task
logout	-	Log out of the CL
lparam	-	List the parameters of a task
mktemp	-	Make a temporary (unique) file name
mkdir	-	Make a new file sub-directory
package	-	Define a new package, OR
		Print the current package names (no arguments)
print	-	Format and print a line on the standard output
radix	-	Print a number in the given radix
redefine	-	Redefine a task
scan	-	Scan and format the standard input
service	-	Service a query from a background job
set	-	Set an environment variable, OR
		Print environment (no arguments)
show	-	Show the values of one or more environment variables
sleep	-	Pause execution for specified period
substr	-	Extract a substring from a string
task	-	Define a new task
unlearn	-	Restore the default parameters for a task or package
update	-	Update a task's parameters (flush to disk)
version	-	Print the revision date of the CL
wait	-	Wait for all background jobs to complete

A.1.2 System Package Functions

allocate	-	Allocate a device
beep	-	Beep the terminal
clear	-	Clear the terminal screen
concatenate	-	Concatenate a list of files to the standard output
copy	-	Copy a file, or copy a list of files to a directory
count	-	Count the number of lines, words, and characters in a file
deallocate	-	Deallocate a previously allocated device
delete	-	Delete a file
devstatus	-	Print the status of a device
directory	-	List files in a directory
diskspace	-	Show how much diskspace is available
edit	-	Edit a file
files	-	Expand a file template into a list of files
gripes	-	Post bug reports, complaints, suggestions
head	-	Print the first few lines of a file
help	-	Print online documentation
lprint	-	Print a file on the line printer device
match	-	Print all lines in a file that match a pattern
news	-	Page through the system news file
page	-	Page through a file
pathnames	-	Expand a file template into a list of OS pathnames
protect	-	Protect a file from deletion
rename	-	Rename a file
revisions	-	Print/post a revision notice for a package
rewind	-	Rewind a device
sort	-	Sort a text file
spy	-	Show processor status
stty	-	Show/set terminal characteristics
tail	-	Print the last few lines of a file
tee	-	Tee the standard output into a file
time	-	Print the current time and date
type	-	Type a file on the standard output
unprotect	-	Remove delete protection from a file

A.2 SDAS Analysis Packages

- General Data Analysis

areavolum	-	Integrate to find the area/volume under a curve/surface
arrayops	-	Perform arithmetic, logical, and matrix operations on SDAS data arrays
convert	-	Convert data from one type to another (real, integer, logical, byte)
curfit	-	Fit curve to one-dimensional data
dimage	-	General image display package
extract	-	Extract a subset of a data array
fitsrd	-	Read a standard FITS tape and create an SDAS disk data file
fitswr	-	Write a standard FITS tape from an SDAS disk data file
four1d	-	Perform one-dimensional Fourier analysis
four2d	-	Perform two-dimensional Fourier analysis
hstats	-	Compute standard statistics, including histograms
locate	-	Locate features in a spectrum, time series, or image
makemask	-	Make a data mask
plot1d	-	Plot one-dimensional (equally-spaced) data
plot2d	-	Plot two-dimensional (equally-spaced) data as contour map or ruled-surface map
probdist	-	Compute standard probability distributions
register	-	Compute registration parameters for two displaced data arrays (use in conjunction with resample)
repmod	-	Replace/modify/input data in an existing data array
resample	-	Resample data from one grid to another (shift, rescale, rotate)
smooth	-	Smooth data by convolution filtering, median window

- Spectral Analysis

cntana	-	Continuum analysis (determine reddening, correct for reddening, fit continuum models)
ewlnst	-	Measure equivalent width/line strength
gspect	-	Generate a spectrum for testing
rvdet	-	Measure radial velocities
specph	-	Spectrophotometry

- Image Analysis

gimage	-	Generate an image for testing

- Time Series Analysis

glcurv	-	Generate a light curve for testing
hldelc*	-	Correct HSP times for light delay time
hspcir*	-	Correct HSP data for instrumental response
hspolar*	-	Polarimetry package for HSP data
hspphot*	-	Photometry package for HSP data
hsubbkg*	-	Subtract background from HSP data

- Astrometric Analysis

bdresid*	-	Make histogram of FGS beam deflector residuals
centroid*	-	Centroid raw FGS encoder data
errsig*	-	Make histogram of FGS error signals

∗ — these programs may not be available

A.3 IRAF Application Packages

- CRYOMAP Package

extract	-	Extract Cryomap spectra
findspectra	-	Find Cryomap spectra
iids	-	Convert integrated spectra extractions to IIDS format
maplist	-	List information about the multi-aperture plate
specplot	-	Plot extracted integrated spectra

- DATAIO Package

bintxt	-	Convert a binary file to an IRAF text file
ldumpf	-	List the permanent files on a Cyber DUMPF tape
mtexamine	-	Examine the structure of a magnetic tape
rcamera	-	Convert a Forth/Camera image into an IRAF image
rcardimage	-	Convert a cardimage file into a text file
rdumpf	-	Convert IPPS rasters from a DUMPF tape to IRAF images
reblock	-	Copy a binary file, optionally reblocking
rfits	-	Convert a FITS image into an IRAF image
ridsfile	-	Convert IDSFILES from a DUMPF tape to IRAF images
ridsmtn	-	Convert mountain format IDS/IRS data to IRAF images
ridsout	-	Convert a text file in IDSOUT format to IRAF images
rpds	-	Convert a PDS image into an IRAF image
rrcopy	-	Convert IPPS rasters from an RCOPY tape to IRAF images
txtbin	-	Convert an IRAF text file to a binary file
wcardimage	-	Convert text files to cardimage files
wfits	-	Convert an IRAF image into a FITS image
widsout	-	Convert an IRAF image to IDSOUT text format

- ECHELLE Package

background	-	Subtract a scattered light background
extract	-	Extract Echelle orders
findorders	-	Find Echelle orders
iids	-	Convert integrated spectra extractions to IIDS format
orderplot	-	Plot extracted integrated spectra

- GENERIC Package

biassub	-	Subtract a bias image
chimages	-	Change images: trim, flip, transpose, rotate
colbckgrnd	-	Fit and subtract a column by column background
colflat	-	Create a flat field by fitting a function to the image columns
darksub	-	Scale and subtract a dark count image
dcbias	-	Subtract a constant bias and trim images
flatten	-	Flatten images using a flat field
linebckgrnd	-	Fit and subtract a line by line background
lineflat	-	Create a flat field by fitting a function to the image lines
normalize	-	Normalize images
normflat	-	Create a flat field by normalizing and replacing low values

- IMAGES Package

imarith	-	Simple image arithmetic
imaverage	-	Average images together
imcopy	-	Copy an image
imdelete	-	Delete an image
imlinefit	-	Fit a function to each image line
imheader	-	Print an image header
imhistogram	-	Compute image histogram
imstatistics	-	Compute and print image statistics
imtranspose	-	Transpose a two dimensional image
listpixels	-	Convert an image section into a list of pixels
sections	-	Expand an image template on the standard output
shiftlines	-	Shift image lines
tv	-	Image display (see TV-IMAGE Package)

- LISTS Package

average	-	Compute the mean and standard deviation of a list
gcursor	-	Read the graphics cursor
imcursor	-	Read the image display cursor
table	-	Format a list of words into a table
tokens	-	Break a file up into a stream of tokens
unique	-	Delete redundant elements from a list
words	-	Break a file up into a stream of words

- LOCAL Package

binpairs	-	Bin pairs of (x,y) points in log separation
epix	-	Edit pixels in an image
fields	-	Extract specified fields from a list
imreplace	-	Replace pixels in a range by a constant
imscale	-	Scale an image to a specified (windowed) mean
imstack	-	Stack images into an image of higher dimension
imsurfit	-	Fit a surface to an image
imtitle	-	Change the title of an image
notes	-	Record notes
polyfit	-	Fit polynomial to lists of X,Y pairs

- MULTISPEC Package

findpeaks	-	Find the peaks
fitfunction	-	Fit a function to the spectra parameter values
fitgauss5	-	Fit spectra profiles with five parameter Gaussian model
modellist	-	List data and model pixel values
msextract	-	Extract spectra
mslist	-	List entries in a MULTISPEC database
msplot	-	Plot a line of image and model data
msset	-	Set entries in a MULTISPEC database
newextraction	-	Create a new MULTISPEC extraction database
newimage	-	Create a new multi-spectra image

- PLOT Package

contour	-	Make a contour plot of an image
graph	-	Graph one or more image sections or lists
pcol	-	Plot a column of an image
pcols	-	Plot the average of a range of image columns
prow	-	Plot a line (row) of an image
prows	-	Plot the average of a range of image lines
surface	-	Make a surface plot of an image

- SOFTOOLS Package

hdbexamine	-	Examine a help database
lroff	-	Lroff (line-roff) text formatter
make	-	Table driven utility for maintaining programs
mkhelpdb	-	Make (compile) a help database
mklib	-	Make or update an object library
mkmanpage	-	Make a manual page
xcompile	-	Compile and/or link an SPP, C or Fortran program
yacc	-	Build an SPP language parser

- TV-IMAGE Package

blink	-	Blink the TV display
display	-	Manipulate the TV display
erase	-	Erase the TV display
frame	-	Define the frames to be manipulated
lumatch	-	Match color look up tables
monochrome	-	Set display into monochrome mode
pseudocolor	-	Set pseudocolor mode on display
rgb	-	Set true RGB mode on display
window	-	Define a display window area
zoom	-	Zoom the display

- UTILITIES Package

airmass	-	Compute the airmass at a given elevation above the horizon
ccdtime	-	Compute time required to a observe star of given magnitude
detab	-	Replace tabs with tabs and blanks
entab	-	Replace blanks with tabs and blanks
lcase	-	Convert a file to lower case
precess	-	Precess a list of astronomical coordinates
translit	-	Replace or delete specified characters in a file
ucase	-	Convert a file to upper case
urand	-	Uniform random number generator

A.4 IRAF Editor Functions

Command	Emacs	EDT†	Vi‡
move-up	↑ or CTRL/P	↑	j or CTRL/P
move-down	↓ or CTRL/N	↓	k or CTRL/N
move-right	→ or CTRL/F	→	l or →
move-left	← or CTRL/B	←	h or ←
ins-chr/word	*text*	*text*	i/a-*text* ESC
del-left	CTRL/H or DEL	DEL	DEL
del-char	CTRL/D	,	x
del-word	ESC d	-	d w
del-line	CTRL/K	PF4	d d
undel-char	ESC CTRL/D	GOLD ,	u
undel-word	ESC CTRL/W	GOLD -	u
undel-line	ESC CTRL/K	GOLD PF4	u
set-fwd		4	
set-rev		5	
next-word	ESC f	1	w
prev-word	ESC b	5 1	b
move-eol	CTRL/E	2	$
move-bol	CTRL/A	BS or CTRL/H	.
next-page	CTRL/V	7	CTRL/D or CTRL/F
prev-page	ESC V	5 7	CTRL/U or CTRL/B
move-start	ESC <	GOLD 5	1 G
move-end	ESC >	GOLD 4	G
get-help	ESC ?	PF2	PF2 or ESC ?
repaint	CTRL/L	CTRL/R	CTRL/L
exit-update	CTRL/Z	CTRL/Z	: w q
exit-no update	CTRL/C	CTRL/C	: q !

† — EDT employs the notion of "direction" (forward and backward cursor motion). Several command sequences are preceded by 5 to indicate explicitly that they only function after setting "backward" mode. All EDT keystrokes, with the exception of CTRL keys, use the keypad.

‡ — Vi has *insert/replace/change modes*, which are entered by command and terminated by the ESC key. Vi-type keystrokes for **eparam** and **ehist** are not yet implemented.

B Glossary

AAS — American Astronomical Society.

band — A two dimensional array. The Nth band of a three dimensional array or **image** is denoted by the subscript [∗,∗,N], where ∗ refers to all the pixels in that dimension.

binary file — An array or sequence of data words. Data is transferred between a binary file and a buffer in the calling program by a simple copy operation, without any form of conversion.

binary operator — An operator which combines two operands to produce a single result (e.g., the addition operator in $x + y$).

brace — The left and right braces are the characters '{' and '}'. Braces are used in the CL and in the SPP language to group statements to form a compound statement.

bracket — The left and right brackets are the characters '[' and ']'. Brackets are used in the CL and in the SPP language to delimit array subscripts.

byte — The smallest unit of storage on the host machine. The IRAF system assumes that there are an integral number of bytes in a **char** and in an address increment (and therefore that the byte is not larger than either). On most modern computers, a byte is 8 bits, and a **char** is 16 bits (I∗2). If the address increment is one byte, the machine is said to be *byte addressable*. Other machines are *word addressable*, where one word of memory contains two or more bytes. In the SPP language, SZB_{CHAR} gives the number of bytes per char, and SZB_{ADDR} gives the number of bytes per address increment.

C — A powerful modern language for both systems and general programming. C provides data structuring, recursion, automatic storage, a fairly standard set of control constructs, a rich set of operators, and considerable conciseness of expression.

char — The smallest signed integer that can be directly addressed by programs written in the SPP language. The char is also the unit of storage in IRAF programs; the sizes of objects are given in units of chars, and binary files and memory are addressed in units of chars. Since the SPP language interfaces to the machine via the local Fortran compiler, the Fortran compiler determines the size of a char. On most systems, the IRAF data type *char* is equivalent to the (nonstandard) Fortran datatype I∗2.

CL — The IRAF Command Language. The CL is an interpreted language designed to execute external **tasks**, and to manage their **parameters**. The CL organizes tasks into a hierarchical structure of independent **packages**. Tasks may be either **script tasks**, written in the CL, or compiled **programs**, written in the SPP language, and linked together to form **processes**. A single process may contain an arbitrary number of tasks.

The CL provides **redirection** of all I/O streams, including graphics output, and cursor readback. Other facilities include command logging, an on-line help facility, a "programmable desk calculator" capability, and a **learn mode**. New packages and tasks are easily added by the user, and the CL environment is maintained in the user's own directories, providing continuity from session to session.

column — a one-dimensional array. The Nth column vector of a two dimensional array or image is denoted by the subscript [N,$*$], where $*$ refers to all the pixels in that dimension. The Nth column of the Mth band of a three dimensional array or image is denoted by [N,$*$,M].

coupling — A measure of the strength of interdependence among modules. The independence of modules is maximized when coupling is minimized.

CTIO — Cerro Tollolo Image Observatory, one of the NOAO facilities located in Chile.

data structure — An aggregate of two or more data elements, where the elements are not necessarily of the same type. Examples include arrays, files, records, linked lists, trees, graphs, and so on.

data file — a data storage file. Data files are used to store program generated **records** or descriptors, that contain the results of the analysis performed by a program. Datafile records may be the final output of a program, or may be used as input to a program. Data file may contain ASCII or binary data, and may have implicit or explicit data structures.

environment variables — Parameters that affect the operation of *all* IRAF programs. Environment variables define logical names for directories, associate physical devices with logical device names, and provide control over the low level functioning of the IRAF file I/O system.

ECF — European Coordination Facility. The center that is to coordinate use of Space Telescope data and programs for the European scientific community.

ESO — European Southern Observatory, headquartered at Garching, FDR.

field — An element of a **data structure** or **record**. Each field has a name, a datatype, and a value.

FITS — Flexible Image Transport System. FITS is a standard tape format used to transport images (pictures) between computers and institutions. Developed in the late 1970s by Donald Wells (KPNO), Eric Greisen (NRAO), and Ron Harten (Westerbork), the FITS standard is now widely used for the interchange of image data between astronomical centers, and is officially sanctioned by both the AAS and the IAU.

Fortran — As the most widely used language for scientific computing for the past twenty years, Fortran needs little introduction. Fortran is used in the IRAF system as a sort of "super assembler" language. Programs and procedures written in the IRAF **SPP** language are mechanically translated into a highly portable subset of Fortran, and the Fortran modules are in turn translated into object modules by the host Fortran compiler. Existing numerical and other modules, already coded in the Fortran language, are easily linked with modules written in the SPP language to produce executable programs. The IRAF system and applications software does not use any Fortran I/O; all I/O facilities are provided by the IRAF **program interface** and **virtual operating system**.

function — A procedure which returns a value. Functions must be declared before they can be used, and functions must only be used in expressions. It is illegal to *call* a function.

HSI — The IRAF Host System Interface, i.e., the interface between the portable IRAF software and the host system. The HSI include the **kernel**, the **bootstrap utilities**, and any host dependent graphics device interfaces.

hidden parameters — Parameters that are not displayed by the CL. The CL does not query for hidden parameters, but automatically uses the default values. Hidden parameters may be set on the command line, but the value from the command line will not be **learned**.

IAU — The International Astronomical Union

IKI — The Image Kernel Interface. The IKI gives IRAF the capability of dealing with multiple physical image storage formats. The high level image i/o software calls the IKI, which in turn calls one of the format specific image kernels, e.g., the OIF kernel or the STF kernel.

identifier — A sequence of characters used to name a procedure, variable, etc. in a compiled language. In the CL and SPP languages, an identifier is an upper or lower case letter, followed by any number of upper or lower case letters, digits, or underscore characters.

image — An array of arbitrary dimension and datatype, used for bulk data storage. An image is an array of **pixels**.

imagefile — The form in which images are stored in the IRAF system. IRAF currently supports images of up to seven dimensions, in any of eight different data types. Only *line storage* mode is currently available, but support for VMS mapped image sections is planned. The imagefile structure is actually implemented as two separate files, an **image header file** and a **pixel storage file**.

image header file — a file describing the contents of an image. It is a small file that is normally placed in the user's own directory system.

interface — The visible portion of a system, program or collection of programs. The only portion of such a entity that other entities need to have knowledge of or access to. The connection between hardware or software entities.

IRAF — The Image Reduction and Analysis Facility. IRAF comprises a **virtual operating system**, a command language (CL), a general purpose programming language (SPP, which was developed along with IRAF), a large I/O library, and numerous support utilities and scientific applications programs. The system is designed to be transportable to any modern superminicomputer. The system provides extensive facilities for general image processing, astronomical data reduction and analysis, scientific programming, and general software development.

IRAF Guru — Any individual whose knowledge of IRAF is greater than yours. Gurus' wisdom embraces all of the essential mysteries of IRAF, and usually includes the locations of good Chinese restaurants.

kernel — A host dependent library of SPP (or Fortran) callable subroutines implementing the primitive system services required by the portable IRAF virtual operating system (VOS). Most of the machine dependence of IRAF is concentrated into the kernel.

learn mode — A facility designed to simplify the use of the CL. By default, the CL "learns" the value of all function **parameters** that are prompted for or explicitly set.

line — A one-dimensional array. The Nth line of a two dimensional array or image is denoted by the subscript [$*$,N], where $*$ refers to all the pixels in that dimension. The Nth line of the Mth band of a three dimensional array or image is denoted by [$*$,N,M].

list structured parameter — A text file, each line of which is a record that contains one or more fields, separated by blanks or commas, that can be interpreted by the CL. Not all fields need be present, omitted fields are indicated by insertion of an extra comma (fields can only be omitted from right to left).

Lroff — The text formatter that is part of the portable IRAF system and used to process **help** file text. Lroff is patterned after the UNIX **Troff** text formatter.

macro — (1) A **script task**. (2) An inline function with zero or more arguments that is expanded by text substitution during the preprocessing phase of compilation.

metacharacter — Characters that have special meaning to the CL. For example, the asterisk '$*$' is used as a "wild card" placeholder; any alphanumeric character is considered a match.

MIDAS — Munich Image Data Analysis System. An analysis package under development by the ESO.

NOAO — National Optical Astronomy Observatories.

NRAO — National Radio Astronomy Observatory.

operand — A data object that is operated upon by an operator, **procedure**, or **task**. Operands may be used for either input or output, or both.

OIF — The old IRAF image format. Refers to the physical format in which images are stored on disk, as well as to the **IKI** kernel used to access images stored externally in the OIF format.

OS — Operating System.

package — (1) A collection of modules that are logically related (e.g., the set of **system** utilities). (2) A set of modules that operates on a specific *abstract datatype*. The modules in a package may be either **procedures** or **tasks**. Examples of abstract datatypes include the CL, the file, the **imagefile**, and so on.

parameter — An externally supplied argument to a module which directly controls the functioning of the module.

pathname — An absolute OS-dependent filename specification.

pipe — An abstraction that connects the output of one **task** to the input of another. The implementation of pipes is OS-dependent.

pixel — Picture element. The fundamental unit of storage in an **image**.

pixel storage file — a file that contains image pixel data. Typically, it is a bulky file and for this reason it is usually placed in a file system designated for such files.

portable — A program is said to be portable from computer A to computer B if it can be moved from A to B without change to the source code. A program is said to be *transportable* from computer A to computer B if the effort required to move the program from A to B is much less than the effort required to write an equivalent program on machine B from scratch.

positional parameters — Parameters that are required for the execution of a given **func-**

tion, and will be queried for by the CL if not given on the command line. Positional *arguments* are the first arguments on the command line (following the command), and they are associated with parameters by their position on the command line. The first positional parameter will be set by the first positional argument on the command line, the second positional parameter by the second positional argument, and so on.

preprocessor — A program which transforms the text of a source file prior to compilation. A preprocessor, unlike a compiler, does not fully define a language. A preprocessor transforms only those constructs which it understands; all other text is passed on to the compiler without change. The SPP language is implemented as a pre-processor.

procedure — A separately compiled program unit. The procedure is the primary construct provided by programming languages for the *abstraction of function*. The external characteristics of a procedure are its name, argument list, and optional return value.

process — An executable partition of memory in the host computer. The host OS initiates a process by copying or mapping an executable file into main memory. In a multitasking and multiuser system, a number of processes will generally be resident simultaneously in main memory, and the processor will execute each in turn.

program — A compiled procedure called by the CL. The procedure must be referenced in a **task** statement before it can be accessed by the CL. An arbitrary number of programs may be linked to form a single **process**.

program interface — The interface between an applications program and everything else. In IRAF, the program interface defines access to all I/O devices, system services, files, and several other non-computational facilities that programs require.

record — A **data structure** consisting of an arbitrary set of fields, used to pass information between program modules or to permanently record the results of an analysis program in a **data file**.

redirection — The allocation of an input or output stream to something other than the standard device. For example, **tasks** can be made to write output to files instead of terminals and the output of one task may be redirected to the input of another.

script task — An interpreted program written in the CL. A script task, like a compiled program, may have formal parameters and local variables. A script task may call another task, including another script task, but may not call itself. To the caller, script tasks and compiled programs are equivalent.

SDAS — Science Data Analysis System. A set of applications routines that are under development at STScI.

SPP — The IRAF Subset Preprocessor Language. A general purpose language patterned after Ratfor and C, the SPP provides advanced capabilities, modern control constructs, enhanced portability, and support for the IRAF runtime library (CL interface, etc.).

STF — The STScI SDAS group data image format. Refers to the physical format in which images are stored on disk, as well as to the **IKI** kernel used to access images stored externally in the STF format.

STScI — Space Telescope Science Institute.

system interface — The interface between the portable IRAF software and the host operating system. The system interface is a **virtual operating system**. The system interface routines, described in *A Reference Manual for the IRAF System Interface* are in principle the only parts of a system that need to be changed when porting the system to a new computer.

task — A CL callable program unit. CL tasks may be script tasks, external programs, or compiled procedures which are built in to the CL.

task statement — (1) The CL statement that enters the name of a task in the IRAF task dictionary, defines the type of task, and in the case of a compiled task, the name of the process in which it resides. (2) The statement in the SPP language that defines a list of programs to be linked together to form a single process.

template — A string consisting of one or more names, which may or may not contain patterns (with **metacharacters**).

text file — A file which contains only text (ASCII character data), and which is maintained in the form expected by the text processing tools of the host OS.

Troff — The UNIX text formatter.

unary operator — An operator which operates on a single operand, e.g., the minus sign in the expression "$-x$".

UNIX — An operating system developed at Bell Labs in the early 1970s by Ken Thompson and Dennis Ritchie. Originally developed for the PDP11, UNIX is now available on a wide range of machines, ranging from micros to superminis, mainframes, and supercomputers.
UNIX is the software development system for the IRAF project at NOAO.

virtual file — A file that uses a machine independent filename within IRAF. The virtual filename is mapped to its host OS counterpart by the CL.

virtual memory — A form of addressing that enables a process to address locations that are not in physical memory. The amount of physical memory available to a process is known as the *working set* of a process; the virtual address space is organized into a series of fixed size *pages*. Pages which are not *memory resident*, i.e., not in the working set, reside on some form of backing store, usually a disk file. When a page is referenced which is not in the working set, a *page fault* occurs, causing the page to be read into the working set.

virtual operating system — A set of system calls that define primitive functions comparable to those provided by an actual operating system. IRAF provides routines (the so-called **program interface**) for file access, process initiation and control, exception handling, logical names, etc.

VMS — The native operating system for the Digital VAX series of supermini computers.

VOS — The IRAF Virtual Operating System. The VOS implements all of the basic functionality provided by IRAF, and defines the environment in which applications programs are written. For example, the VOS provides facilities for file access, image access, access to graphics and image display devices, access to the command language to fetch parameters etc., process control and exception handling facilities, and so on. The VOS is written in portable SPP using the facilities provided by the IRAF **kernel**.

Z-routines — Machine dependent routines used to interface to the host operating system. The IRAF Z-routines are maintained in the package *OS*.