The IRAF Data Reduction and Analysis System

Doug Tody

National Optical Astronomy Observatories
P.O. Box 26732, Tucson, Arizona, 85726

ABSTRACT

The Image Reduction and Analysis Facility (IRAF) is a general purpose software system for the reduction and analysis of scientific data. The IRAF system provides a good selection of programs for general image processing and graphics applications, plus a large selection of programs for the reduction and analysis of optical astronomy data. The system also provides a complete modern scientific programming environment, making it straightforward for institutions using IRAF to add their own software to the system. Every effort has been made to make the system as portable and device independent as possible, so that the system may be used on a wide variety of host computers and operating systems with a wide variety of graphics and image display devices.

1. Introduction

The IRAF project began in earnest in the fall of 1981 at Kitt Peak National Observatory (NOAO did not yet exist at that time). The preliminary design of the system was completed early in 1982, and the first versions of the command language (CL) and the applications programming environment were completed during 1982. The NOAO IRAF programming group was formed in 1983. The first internal release of the system occurred at NOAO in 1984, and a beta release of the system to a few outside sites occurred in 1985.

The Space Telescope Science Institute (STScI) selected IRAF to host their Science Data Analysis System (SDAS) in December of 1983, and carried out the initial port of IRAF to VMS, as well as some CL extensions, during 1984 and 1985. In June of 1985, UNIX/IRAF became the primary reduction and analysis facility at NOAO/Tucson. By October the VMS version of the system was fully functional at NOAO on the newly installed VAX 8600, which soon became our primary data processing system. By late 1985 the system had been ported to such disparate systems as a Sun workstation running UNIX and to a Data General MV/10000 running AOS/VS (the latter port was still in progress when this paper was written and was being undertaken by Steward Observatory, Univ. of Arizona). In February of 1986 a limited public release of the system occurred, with UNIX and VMS versions of the system being distributed to about 40 astronomical sites. The system is expected to be made generally available sometime in 1987.

This paper describes the system as it existed in March of 1986, shortly after the first public release. The focus of the paper is primarily on the IRAF system software as seen by the user and by the software developer, although the NOAO science applications software is briefly introduced. The distinction is made because the IRAF system software is expected to be used by numerous institutions to host the science software developed independently by each institution. The NOAO and STScI science software packages are the first examples of this; similar undertakings are already in progress, and more are expected in the future as the system becomes more mature and more widely used. These science software systems are major projects in their own right and are best described elsewhere.

The purpose of this document is to present an overview of the IRAF system from the system designer's point of view. After a brief discussion of the global system architecture, we take a tour through the system, starting at the user level and working down through the programming environments and the virtual operating system, and ending with the host system interface. The emphasis is on the system design, on the functionality provided by the various subsystems, and on the reasoning which went into the design. The reader is assumed to be familiar with the technology and problems associated with large software systems and large software development projects.

2. System Architecture

2.1 Major System Components

The major components of the IRAF system are outlined in Figure 1. The **command language**, or CL, is the user's interface to IRAF. The CL is used to run the **applications programs**, which are grouped into two classes, the system utilities and the scientific applications programs. Both the CL and all standard IRAF applications programs depend upon the facilities of the IRAF **virtual operating system** (VOS) for their functioning. The VOS in turn depends upon the **kernel**, the runtime component of the host system interface (HSI), for all communications with the host system. All software above the host system interface is completely portable to any IRAF host, i.e., to any

system which implements the HSI. The system is ported by implementing the HSI for the new host; note that the effort required to port the system is independent of the amount of code above the HSI, and once the system is in place no additional effort is required to port new applications software.

Command Language (CL)	user interface, command interpreter
Applications Programs	system utilities, scientific applications programs
Virtual Operating System (VOS)	the system libraries, all i/o interfaces
Host System Interface (HSI)	bootstrap utilities, **kernel** primitives

Figure 1. Major System Components

From the point of view of the system structure, the CL is itself an applications program in that it uses only the facilities provided by the IRAF VOS. In principle an applications program can do anything the CL can do, and multiple command languages can coexist within the same system. In practice only the CL is allowed to interact directly with the user and spawn subprocesses, in order to provide a uniform user interface, and to minimize the kernel facilities required to run an applications program. All standard IRAF applications programs can be run directly at the host system level as well as from the CL, making it possible to run the science software on a batch oriented system which is incapable of supporting the CL.

2.2 Process Structure

In normal interactive use IRAF is a multiprocess system. The standard process structure is depicted in Figure 2. The CL process handles all communications to the user terminal, which is usually a graphics terminal of some sort. All applications programs run as concurrent subprocesses of the CL process. A single applications process will usually contain many executable programs or compiled **tasks**; the CL maintains a **process cache** of connected but idle (hibernating) subprocesses to minimize the process spawn overhead. If graphics i/o to a device other than the graphics terminal is necessary, a graphics kernel process is connected to the CL as a subprocess.

The process cache always contains the process "x_system.e", which contains all the tasks in the **system** package. The system subprocess is locked in the cache by default. The remaining process cache slots (typically 2 or 3 slots) are dynamically assigned as tasks are run by the user. Up to 3 graphics kernels may be simultaneously connected. The entire process structure is duplicated when a background job is submitted by the user from an interactive CL.

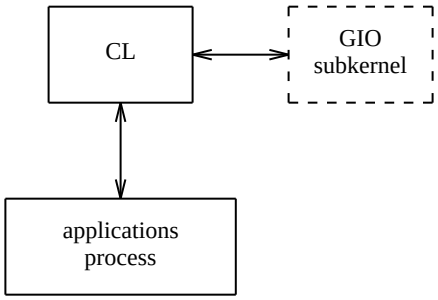

Figure 2. IRAF Process Structure

The multiprocess architecture has significant **flexibility advantages** over the alternative single process and chained process architectures. The system is highly modular and easily extended, allowing new versions of the CL or a new graphics kernel to be tested or installed even while the system is in use, without having to relink all the applications modules. New applications modules can be debugged outside the normal CL environment using host system facilities, and then installed in the system while the system is in use without any affect on the rest of the system. There is no limit to the number of applications packages which the system can support, nor is there any limit to the number of graphics devices which the system can be interfaced to. Support for a new graphics device can be added to a running system without any affect on the existing applications programs.

The multiprocess architecture also has significant **efficiency advantages** over less modular architectures. Since most of the system software resides in independent processes, the amount of code which has to be linked into an applications program is minimized, reducing the link time as well as the disk and memory requirements for an executable. Since all users on a multiuser system use the same CL executable, graphics kernels, and other system executables, significant savings in physical memory are possible by employing shared memory access to the executables. The ability of IRAF to link many tasks into a single executable promotes code sharing, reducing disk and memory requirements and greatly improving interactive response by minimizing process connects. Paradoxically, the

use of multiple concurrent processes can actually improve performance by permitting pipelined execution, e.g., the applications process can be busy generating graphics metacode while the CL or graphics kernel is waiting for i/o to a graphics device.

The **chief disadvantage** of the IRAF process structure is the difficulty of error recovery in response to a user interrupt or program abort. An interrupt may occur while all processes in the group are busily computing and passing messages and data back and forth via interprocess communication (IPC), making it necessary to terminate the current task and clear and synchronize the entire IPC data path. That this problem is tractable at all is due to the master/slave nature of the IPC protocol. At any one time there will be only one master process in the system. When an interrupt occurs it is only the master process which is (logically) interrupted. If the task currently executing in the master process does not intercept the interrupt and either ignore it or take some special action, control will pass to the VOS error recovery code in the master process, which will oversee the synchronization and cleanup of the i/o system before returning control to the CL.

3. The Command Language (CL)

3.1 Basic Concepts

The IRAF Command Language (CL) is the user's interface to the IRAF system. The CL organizes the many system and applications **tasks** (programs) into a logical hierarchy of **packages**. A package is a collection of logically related tasks, and is represented to the user using a type of **menu**. Each task has its own local set of **parameters**. To keep the calling sequence concise, each task has only a few required or **query mode** parameters. For maximum flexibility, tasks may provide any number of optional **hidden mode** parameters as well, each with a reasonable default value chosen by the programmer but modifiable by the user, either permanently or via a command line override.

A package is implemented as a special kind of task, and packages often contain "tasks" which are really subpackages, hence the logical organization of packages is a tree. A package must be **loaded** by typing its name before any of the tasks therein can be executed or referenced in any other way by the CL. Loaded packages are organized as a linear list, with the list order being the order in which the packages were loaded. References to tasks in loaded packages are resolved by a circular search of this list, starting with the **current package**, which may be any package in the set of loaded packages. If a task with the same name appears in more than one package, the package name may optionally be specified to resolve the ambiguity. Note that is is not necessary to traverse the package tree to execute a task in a loaded package.

3.2 Command Language Features

The most notable features of the IRAF command language are summarized in Figure 3. The CL is designed to serve both as a **command language** and as an interpreted **programming language**. The emphasis in this initial version of the CL has been on providing good command entry facilities. Extensive CL level programming facilities are also provided in the current CL, but full development of this aspect of the CL is a major project which must wait until development of the baseline IRAF system is completed.

- provides a uniform environment on all host systems
- package structure for organization and extensibility
- menus and extensive online help facilities
- concise command syntax similar to UNIX cshell
- i/o redirection and pipes; aggregate commands
- minimum match abbreviations for task and parameter names
- both local and global parameters, hidden parameters
- direct access to host system; foreign task interface
- parameter set editor; command history editor (edt, emacs, vi)
- background job submission (including queuing)
- logfile facility for recording all task invocations
- graphics and image display cursor mode facilities
- virtual filename facility; unix style pathnames to files
- programmable: procedures, C style expressions and control constructs

Figure 3. Selected Features of the IRAF Command Language

The basic IRAF command syntax is the same as that used in the UNIX cshell. Similar **i/o redirection** and **pipe** facilities are provided, extended in the CL to provide support for the standard graphics streams. **Background**

job submission facilities are provided, including support for batch queues, control of job priority, and servicing of parameter queries from background jobs after the job has been submitted. A cshell like **history mechanism** is provided, extended in the CL to record multiline command blocks rather than single command lines, and including a builtin screen editor facility for editing old commands. Minimum match abbreviations are permitted for task and parameter names, allowing readable (long) names to be used without sacrificing conciseness.

Extensive online **help** facilities are provided, including the package menu facility already mentioned, a utility for listing the parameters of a task, as well as online manual pages for all tasks. An interactive **cursor mode** facility provides a builtin graphics or image display control capability operable whenever a cursor is read by an applications program, without need to exit the applications program. Cursor mode is discussed further in §6.6.

While the CL provides a fully defined, complete environment independent of the host system, an **escape mechanism** is provided for interactively sending commands to the host system. In addition, host system tasks, including user written Fortran or other programs, may be declared as IRAF **foreign tasks** and accessed directly from the CL much like true IRAF tasks, permitting the use of the CL i/o redirection, background job submission, etc. facilities for these tasks. A host system **editor** interface is provided so that the user may access their favorite editor from within the IRAF environment. New IRAF programs and packages may be developed and tested from within the IRAF environment, or programs (CL procedures) may be written in a C like dialect of the command language itself.

It is beyond the scope of this paper to attempt to discuss the user level features of the CL in any detail. The reader is referred to any of the following references for additional information. *A User's Introduction to the IRAF Command Language* explains the basic use of the language, and the *The IRAF User Handbook* contains many examples as well as manual pages for the CL language features. The document *Detailed Specifications for the IRAF Command Language* presents the author's original design for the CL, and although now rather dated contains information about the conceptual design and inner workings of the CL not found in any of the more recent user oriented manuals.

3.3 Principles of Operation

With very few exceptions, all user interaction in IRAF is via the CL. This ensures a consistent user interface for all applications programs, simplifies applications code, and provides maximum flexibility, since the CL (and hence the user) controls all aspects of the environment in which a program is run. Applications programs do not know if they are being used interactively or not, or even if they are being called from the CL. Indeed, any IRAF program may be run at the host system level as well as from the CL, although the user interface is much more primitive when the program is called at the host level.

The CL executes concurrently with the applications process, responding to parameter requests from the applications process, managing the standard i/o streams, processing graphics output and managing cursor input, and so on. In effect the CL and the applications task are one large program, except that binding occurs at process connect time rather than at link time. This makes it possible for programs to have a highly interactive, sophisticated user interface, without linking enormous amounts of code into each executable. A further advantage is that since a single process is used for all user interaction, the *context* in which a task executes is preserved from one task to the next, without need to resort to inefficient and awkward techniques using disk files.

The CL recognizes a number of different types of tasks, most of which have already been mentioned. The **builtin tasks** are primitive functions which are built into the CL itself. **Script tasks** are interpreted CL procedures. **Compiled tasks** are IRAF programs written in some compiled language and executing in a connected subprocess residing in the process cache. Lastly, **foreign tasks** are compiled host programs or host command scripts, which the CL executes by sending commands to the host system. A special case of a builtin task is the **cl task**, the function of which is to interpret and execute commands from a command stream, e.g., the user terminal.

All of these types of tasks are equivalent once the task begins executing, i.e., while a task is executing the function of the CL is to interpret and execute commands from the *task*, until the task informs the CL that it has completed. If a command is received which causes another task to be run, the CL pushes a new task context on its control stack and executes the new task, popping the old context and resuming execution of the old task when the called task terminates. Logout occurs when the original "cl" task exits. The key point here is that the CL functions the same whether it is taking commands from the user, from a script, or from a compiled applications program. This is known as the principle of **task equivalence**, and is fundamental to the design of the CL.

3.4 Extensibility

New tasks or entire packages may be added to the CL at any time by entering simple declarations, hence the CL environment is easily extended by the user. The mechanism used to do this is the same as that used for the packages and tasks provided with the standard system, hence the user has full access to all the facilities used for the standard IRAF tasks, including the help mechanism. No changes have to be made to the standard system to add locally

defined packages and tasks. Conversely, a new version of the standard system can be installed without affecting any local packages (provided there have been no interface changes).

4. Applications Software

4.1 System Packages

The IRAF applications packages are divided into two broad classes, the system packages and the scientific reduction and analysis packages. In this section we introduce the system packages, which are listed in Figure 4. When describing the applications packages, we list all packages which have been implemented or which we plan to implement, since the purpose of this paper is as much to present the design of IRAF as to report its current state. The status of each package is indicated in the table below, where *done* means that the package has reached its planned baseline functionality (of course, all packages continue to evolve after they reach this state), *incomplete* means that the package is in use but has not yet reached baseline functionality, *in progress* means the package is actively being worked on but is not yet in use, and *future* means that work has not yet begun on the package. It should be pointed out that each of these packages typically contains several dozen tasks, and many contain subpackages as well. It is beyond the scope of this paper to delve into the contents of these packages in any detail.

package	*status (March 86)*
dataio - Data input and output (FITS, cardimage, etc.)	done
dbms - Database management utilities	future
images - General image processing, image display	incomplete
language - The command language itself	done
lists - List processing	incomplete
plot - General graphics utilities	done
softools - Software tools, programming and system maintenance	done
system - System utilities (file operations, etc.)	done
utilities - Miscellaneous utilities	done

Figure 4. System Packages

The system packages include both those packages containing the usual operating system utilities, e.g., for listing directories or printing files, as well as those packages which are required by any scientific data processing system, e.g., for general image processing and graphics. The conventional operating system utilities are found in the **system** package. The **language** package contains those tasks which are built into the CL itself. The **softools** package contains the software development and system maintenance tools, including the HSI bootstrap utilities, i.e., the compiler, librarian, the *mkpkg* utility (similar to the UNIX *make*), the UNIX *tar* format reader/writer programs, and so on. The **dbms** package is the user interface to a planned IRAF relational database facility. The **lists** package contains an assortment of tasks for operating upon text files containing tabular data, e.g., for performing a linear transformation on one or more of the columns of a list.

The **dataio** package contains a number of tasks for reading and writing data in various formats, including FITS, cardimage, and a number of other more NOAO specific formats. These programs are typically used to read or write magtape files, but all such programs can be used to operate upon a disk file as well, a useful alternative for sites which have access to an electronic network. The **plot** package contains a number of vector graphics utilities, including CL callable versions of all the NCAR graphics utilities (using the IRAF/GIO GKS emulator). The **images** package, which is actually a tree of related packages, contains the general image processing tasks plus the image display and display control tasks.

4.2 Optical Astronomy Packages

The NOAO packages for the reduction and analysis of optical astronomy data are summarized in Figures 5 and 6. There are two categories of optical astronomy packages. The packages listed in Figure 5 are intended to be of general use for any optical astronomy data, not just for data taken at an NOAO observatory with an NOAO instrument. Since these are intended to be general purpose, instrument independent packages, naturally they are not always the most convenient packages to use for reducing data from a specific instrument. The **imred** packages, summarized in Figure 6, fulfill the need for easy to use or "canned" reduction procedures for specific instruments. In many cases the tasks in the **imred** packages are CL scripts which fetch instrument specific parameters from the image headers and call tasks in the more general, instrument independent packages. The list of **imred** packages is continually growing as new instruments are supported.

package	*status (March 86)*
artdata - Artificial data generation package	in progress
astrometry - Astrometry package	future
digiphot - Digital photometry package	in progress
focas - Faint object detection and classification package	future
imred - NOAO Instrument reduction packages	done
local - Local user added tasks (not configuration controlled)	-
onedspec - One dimensional spectral reduction and analysis package	done
twodspec - Two dimensional spectral reduction and analysis packages	done
surfphot - Galaxy surface brightness analysis package	future

Figure 5. General Optical Astronomy Reduction and Analysis Packages

The **artdata** package consists of tasks for generating various types of test data, e.g., pure test images, artificial starfields, artificial spectra, and so on. The **astrometry** package is used to obtain astrometric coordinates for objects in stellar fields. The **digiphot** package contains a collection of tasks for automatically generating starlists, for performing aperture photometry on an image (fractional pixel, multiple concentric apertures, polygonal apertures), and for performing photometry using point spread function fitting techniques. The **focas** package performs faint object detection and classification (e.g., to discriminate between faint stars and galaxies), and will be largely a port of the existing UNIX package of the same name to IRAF. The **onedspec** package provides a standard set of tools for the dispersion correction, flux calibration, and analysis of one dimensional spectra. The **twodspec** package performs the same operations for two dimensional spectra of various types, and currently consists of the subpackages **longslit**, **multispec**, and **apextract**. The **surfphot** package fits ellipses to the isophotes of galaxies.

package	*status (March 86)*
imred.bias - General bias subtraction tools	done
imred.coude - Coude spectrometer reductions	done
imred.cryomap - Cryogenic camera / multi-aperture plate reductions	done
imred.dtoi - Density to intensity calibration	in progress
imred.echelle - Echelle spectra reductions	done
imred.generic - Generic image reductions tools	done
imred.iids - KPNO IIDS spectral reductions	done
imred.irs - KPNO IRS spectral reductions	done
imred.vtel - NSO (solar) vacuum telescope image reductions	done

Figure 6. Current NOAO Instrument Reduction Packages

The **imred** packages perform general CCD image reductions, as well as the reductions for other more specialized instruments. The **cryomap**, **iids**, **irs**, and **vtel** packages deal with specific NOAO instruments and are probably only of interest to people who observe at an NOAO observatory. The remaining packages should be useful for anyone with CCD, Echelle, or photographic (density) data.

4.3 Third Party Software

In addition to the applications packages already mentioned, all of which are being developed by the IRAF group at NOAO, we anticipate that a fair amount of third party software will eventually be available for use within IRAF as well. The STScI SDAS software is the first example of this. Third party software appears within IRAF as a new branch on the package tree. There is no limit on the size of such an addition, and in the case of SDAS we find a suite of packages comparable to the IRAF system itself in size. As of this writing, a number of other groups are either actively developing additional third party software or are contemplating doing so, but it would be inappropriate to be more specific until these packages are announced by the institutions developing them.

Third party software may unfortunately not meet IRAF standards, hence the software may not be usable on all IRAF hosts, nor usable with all the graphics and image display devices supported by IRAF. Applications software which is built according to IRAF standards is automatically portable to all IRAF hosts without modification (although some debugging is typically required on a new host), hence sites considering adding their own software to IRAF are encouraged to model their software after the existing NOAO IRAF applications.

5. Programming Environments

5.1 Overview

It is unrealistic to expect any finite collection of applications packages to provide everything that a particular user or institution needs. To be most useful a system must not only provide a good selection of ready to use applications software, it must make it easy for users to add their own software, or to modify the software provided. Furthermore, implementation and development of even the standard IRAF applications packages is a major project requiring many years of effort, hence the system must minimize the effort required for software development by professional programmers as well as by users. The solution to these problems is a **programming environment**, or more precisely, a set of programming environments, each tailored to a particular type of software and to the level of expertise expected from the programmer.

The term programming environment refers to the languages, i/o libraries, software tools, and so on comprising the environment in which software development takes place. A good programming environment will provide all the facilities commonly required by applications programs, ideally in a form which is high level and easy to use without sacrificing flexibility and efficiency. The facilities provided by the environment should be layered to provide both high and low level facilities and to maximize code sharing and minimize program size. The programming environment should provide machine and device independence (code portability) as an inherent feature of the environment, without requiring an heroic sacrifice or transcendent wisdom on the part of the programmer to produce portable code.

IRAF currently provides three quite different programming environments. The highest level environment is the CL, where the programming language is the command language itself, and the environment is defined by the CL callable packages and tasks. This is a very attractive programming environment for the scientist/user because of its high level, interactive nature, but much work remains to be done before this environment reaches its full potential. At the opposite extreme is the host Fortran interface, which allows Fortran programs written at the host level, outside of IRAF, to access IRAF images and to be called from the CL. This is of interest because it allows existing Fortran programs to be productively used within IRAF with minimal modifications, and because it makes it possible for users to write image operators immediately without having to learn how to use a more complex (and capable) environment.

The third programming environment is that defined by the IRAF VOS. This is the most powerful and best developed environment currently available, and is used to implement nearly all of the existing IRAF systems and applications software. Full access to the VOS facilities and full portability are available only for programs written in the IRAF SPP (subset preprocessor) language, the language used to implement the VOS itself. A C language interface is is also available, but only a small subset of the VOS facilities are available in this interface, and there are serious portability problems associated with the use of this interface in applications programs (it is currently used only in highly controlled systems applications, i.e., the CL). While IRAF does not currently provide a Fortran interface to the VOS facilities, Fortran subroutines may be freely called from SPP programs, allowing major portions of an applications program to be coded in Fortran if desired. There are, however, serious portability problems associated with the direct use of Fortran for applications programs.

Only the SPP language adequately addresses the problem of providing full functionality without compromising portability. This is because the SPP language is an integral part of a carefully conceived, complete *programming environment*, whereas C and Fortran are merely general purpose third generation programming *languages*. Because it is specially designed for large scientific programming applications, the SPP language and associated programming environment will never see widespread usage like C and Fortran, but for the same reasons it is ideally suited to our applications.

5.2 SPP Language Interface

The IRAF SPP (subset preprocessor) language is a general purpose programming language modeled after C but implemented as a Fortran preprocessor. Programming in SPP is conceptually very similar to programming in C; the SPP language provides much the same basic facilities and syntax as C, including pointers, structures, automatic storage allocation, **define** and **include**, C style character data type, and Ratfor style versions of all the usual control constructs. The same problem will generally be solved the same way in both languages. Since the SPP language resembles C but is translated into Fortran, SPP combines the software engineering advantages of C with the scientific programming advantages of Fortran. In addition, since SPP is an integral part of the IRAF system, SPP provides language level support for the VOS and for the IRAF programming environment in general.

The significance of the SPP language cannot be understood by studying only the language itself as one would study C or Fortran. Rather, one must study the programming environment and the role played by the SPP language in that environment. The major components of the IRAF programming environment are the SPP language, the VOS (§6.1), the software tools, e.g., the **xc** compiler, **mkpkg**, etc. (§7.2), the applications libraries, e.g., **xtools**, and the various math libraries, e.g., **curfit**, **surfit**, **iminterp**, etc (§5.6). Considered as a whole, these components define a very rich programming environment. Few systems provide a programming environment of comparable capability, let alone in a machine and device independent format.

The chief problem facing a programmer trying to write their first applications program in IRAF is learning the programming environment and "how things are done" in IRAF. Learning the SPP language itself is generally a simple problem dispensed with in hours or days, depending upon the individual. While most people can be productively generating new programs within a few days, weeks or months may be required to develop a deep understanding of and fluency with the full environment. This is typical of any large software system capable of supporting sophisticated applications programs, and demonstrates that porting applications programs and applications programmers between different programming environments is a myth. In a sense, there are no (nontrivial) portable applications *programs*, only transportable programming *environments*.

Since a program is only as portable as the environment it is written for, there are few portability advantages to programming large applications in a standardized language (a case can however be made for purely numerical routines). In fact the opposite is often the case, since few if any compilers have ever been written which rigorously implement a language standard and nothing more nor less. In the case of a language like Fortran, it is not uncommon for half of the features offered by a particular manufacturer's compiler to be nonstandard extensions to the formal language standard, or even more dangerous, relaxations of subtle restrictions imposed by the standard. It is difficult for a programmer to resist using such extensions even when they know what the nonstandard extensions are, and usually a programmer will be more concerned with getting the program functioning as soon as possible than with making it portable.

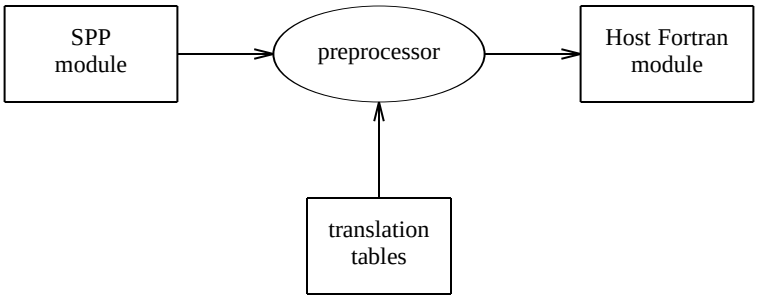

Figure 7. Preprocessor Dataflow

The SPP language solves this problem by providing all the features the programmer needs directly in the language, so that the programmer does not have to do without. If a new feature is needed and can be justified, it can easily be added to the language since IRAF defines the SPP language standard. Since the SPP translator is part of IRAF rather than part of the host system, there is only one translator and the problem of writing code which will be accepted by a variety of host compilers is greatly minimized. The intermediate Fortran generated by the translator uses only the most common features of Fortran, hence is intrinsically highly portable. The intermediate Fortran is prettyprinted (indented to show the structure, etc., so that a human can read it) and may optionally be saved and used for symbolic debugging with the host system debugger.

Since a mechanical translator is used to generate the host Fortran when an SPP program is compiled, nonstandard host Fortran extensions can be used without compromising the portability of applications programs, by simply modifying the host dependent tables used to drive the translation. Since the SPP compiler is part of the IRAF environment rather than the host environment, it understands IRAF virtual filenames, an essential capability for referencing global include files. The **define-include** facility itself is vital for parameterizing the characteristics of the host machine and VOS configuration, as well as for structuring applications software. Since the SPP language places an interface between IRAF applications programs and the host Fortran compiler, our considerable and ever growing investment in applications software is protected from future changes in the Fortran standard.

As the name subset preprocessor implies, the SPP language implements a subset of a planned future language. Most of the limitations of the current SPP language are due to the use of preprocessor technology to carry out the translation. A much more powerful approach is to use a full syntax directed compiler with an associated code generator which generates host Fortran statements rather than assembler instructions. This will greatly improve compile time error checking, increase the portability of both the applications software and the compiler, and will make it possible to include certain advanced features in the language for generalized image and vector processing. This is an exciting area for future research, as compiler technology makes possible the solution of a large class of image processing problems which cannot readily be addressed any other way.

In summary, the IRAF SPP language interface provides a rich scientific programming environment without compromising program portability. Programmers using this environment can concentrate on the problem to be solved without concern for the portability of the resultant software, and are free to use all of the facilities provided by

the language and the environment. All of the facilities one needs for a particular application are likely to either be already available somewhere in the environment, or easily constructed using lower level facilities available in the environment, and are guaranteed to be available in the same form on all IRAF host machines. The proof of the concept of this interface is found in the current IRAF system, where thousands of files in hundreds of directories are routinely moved between quite different IRAF hosts, then compiled and run without any changes whatsoever.

5.3 Host Fortran Interface

The host Fortran program interface (IMFORT) is in most respects the opposite of the SPP/VOS programming environment. The IMFORT interface is a small Fortran callable library which may be linked with host Fortran (or C) programs to get the foreign task command line from the CL and perform some operation upon an IRAF image or images. The host Fortran program may be declared as a foreign task in the CL and accessed much as if it were a conventional IRAF task, using the CL to parse, evaluate, and concatenate the command line to be passed to the foreign task as a string. As a foreign task, the host program may also be run outside the CL, using the host system command interpreter, if desired.

The purpose of the IMFORT interface is to allow the existing Fortran programs in use at a site when IRAF arrives to be modified for use within the IRAF environment with minimal effort. The interface is also useful for the scientist who needs to write a small program and does not want to take the time to learn how to use the SPP/VOS environment. The IMFORT interface consists of only a dozen or so routines hence almost no effort is required to learn how to use the interface. Of course, the IMFORT interface does not provide access to the extensive facilities of the SPP/VOS programming environment, hence is not suitable for the development of large programs. Programs written using the IMFORT interface are generally not portable to other hosts, but this may not be a serious consideration to scientists writing programs for their own personal use.

5.4 IRAF Fortran Interface

As noted earlier, IRAF does not currently have a Fortran applications programming interface, other than the host Fortran program interface. An IRAF Fortran programming environment would provide a subset of the functionality provided by the SPP environment as a higher level library of Fortran callable procedures. This differs from the host Fortran interface in that the resultant programs would be fully integrated into IRAF, with potential access to all SPP environment facilities, whereas the host Fortran interface provides only limited imagefile access and the ability to fetch the CL command line as a string, plus unrestricted access to host system facilities.

We are considering adding such an interface for the scientist/programmer who needs more than the IMFORT interface but is unwilling or unable to invest the time required to learn to use the SPP environment. Unfortunately, the lack of structures, pointers, dynamic memory allocation, **define-include**, filename translation, etc. in ANSI standard Fortran makes it prohibitively difficult to define a Fortran interface with capabilities comparable to the SPP programming environment. Also, the resultant Fortran programs would inevitably use the nonstandard features of the host Fortran compiler and hence would not be portable. If such an interface were made available and then used extensively, it seems likely that it would gradually grow until it approximated the SPP environment in complexity, without the advantage of the more elegant interface made possible by the SPP language.

If an embedded Fortran programming environment is ever added to IRAF it therefore makes sense only if the environment is expressly designed with the scientist/programmer in mind. The interface should provide all the necessary facilities for small scientific programs but nothing more, and it should be possible to become familiar with the use of the interface in a day or less. Simplicity of use should be emphasized rather than efficiency. All large applications projects and all "user qualified" software should continue to be implemented in the SPP language and environment.

5.5 C Language Interface

The IRAF C language interface (library LIBC) consists of a fairly complete UNIX STDIO emulation plus a C binding for a systems programming subset of the IRAF VOS, comparable in capability to a V7 UNIX kernel. All of the standard Berkeley UNIX STDIO facilities are provided, e.g., the definitions in the include files *<stdio.h>* and *<ctype.h>*, and functions such as *fopen*, *fread*, *fwrite*, *getc*, *putc*, *printf*, *scanf*, *malloc*, and so on. The STDIO procedures are implemented as an interface to the IRAF VOS, hence calls to the VOS i/o procedures may be intermixed with calls to the STDIO procedures, and the STDIO emulation is thus part of the portable system. No UNIX sources are used hence a UNIX license is not required to use the interface. Existing UNIX/C programs may be ported to the C language environment with minor modifications (some modifications are always required), assuming that the i/o requirements of the programs are modest.

The C language interface is currently used only to support the CL, which is written in C primarily for historical reasons (the original CL was developed concurrently with the VOS). The C language interface could in principle

be expanded to include more VOS facilities, but the sheer size of the VOS and of the rest of the programming environment makes this impractical. In any event, the SPP language is more suited to scientific programming, avoids the portability problems of calling Fortran library procedures from C, and will always be better integrated into the IRAF programming environment. The use of the C language interface is not recommended except possibly for porting existing large systems programs written in C to IRAF.

5.6 Applications Libraries

The standard applications libraries currently available in IRAF are summarized in Figure 8. All libraries may be called from SPP programs. Only the purely numerical Fortran libraries may be called from Fortran programs. The sources for all libraries used in IRAF are included with the distributed system and are in the public domain. In some cases the sources for the standard numerical libraries have had to be modified slightly to eliminate calls to the Fortran STOP, WRITE, etc. statements, sometimes used in error handlers. Some major public domain math packages have yet to be installed in IRAF, e.g., for nonlinear least squares and for evaluating special functions, for the simple reason that we haven't needed them yet in our applications.

library	*description*
bev	Bevington routines (generally, these should be avoided)
curfit	1-D curve fitting package (SPP)
deboor	DeBoor spline package
gks	IRAF GKS emulator (subset of Fortran binding)
gsurfit	Surface fitting on an irregular grid (SPP)
iminterp	Image interpolation package, equispaced points (SPP)
llsq	Lawson's and Hanson's linear least squares package
ncar	NCAR graphics utilities, GKS version (uses GKS emulator)
nspp	Old NCAR system plot package
surfit	Surface fitting on a regular grid (SPP)
xtools	General tools library for SPP applications programs

Figure 8. Applications Libraries (March 86)

The most heavily used numerical libraries in IRAF are those which were written especially for IRAF (marked SPP in the figure). Our experience has been that most of the generally available interpolation, curve and surface fitting packages are overly general and inefficient for use on bulk image data where the data points tend to be on an even grid, or where the same X vector may be used to fit many Y vectors. The SPP based math packages are nicely packaged, using dynamic memory allocation to internally allocate a descriptor and all working storage, and to hide the details of which of the possible algorithms supported by a package is actually being used. The supported interpolators include nearest neighbor, linear, cubic spline, and third and fifth order divided differences. The supported curve types include linear spline, cubic spline, and the Chebyshev and Legendre orthogonal polynomials. As far as possible the packages are vectorized internally using the VOPS operators, to take advantage of the vector processing hardware anticipated on future machines.

6. The Virtual Operating System (VOS)

6.1 Major Components of the VOS

The primary functions of the VOS are to provide all the basic functionality required by applications programs, and to isolate applications programs from the host system. The VOS defines a complete programming environment suitable both for general programming and for scientific programming in particular. In addition to the standard facilities one expects from a conventional operating system, e.g., file i/o, dynamic memory allocation, process control, exception handling, network communications, etc., the VOS provides many special facilities for scientific programming, e.g., a CL interface, image i/o (access to bulk data arrays on disk), and a graphics subsystem supporting both vector graphics and image display devices. The major subsystems comprising the IRAF VOS are outlined in Figure 9.

CLIO	command language i/o (get/put parameters to the CL)
DBIO	database i/o (not yet implemented)
ETC	exception handling, process control, symbol tables, etc.
FIO	file i/o
FMTIO	formatted i/o (encode/decode, print/scan)
GIO	graphics i/o (both vector graphics and image display access)
IMIO	image i/o (access to bulk data arrays on disk)
KI	kernel interface (network communications)
LIBC	UNIX stdio emulation, C binding for the VOS, used by the CL
MEMIO	memory management, dynamic memory allocation
MTIO	magtape i/o
OSB	bit and byte primitives
TTY	terminal control (*termcap*, *graphcap* access)
VOPS	vector operators (array processing)

Figure 9. Major Subsystems Comprising the IRAF VOS

Although the VOS packages are normally presented as independent packages, there is naturally some vertical structure to the packages. The highest level packages are GIO and IMIO, which depend upon many of the lower level i/o packages. The most fundamental packages are FIO and MEMIO, which are used by everything which does i/o. At the bottom are the KI (the kernel interface) and the kernel itself, which is part of the host system interface (§7.3). All of the VOS code is portable with the exception of certain GIO graphics device kernels, hence the VOS is functionally equivalent on all IRAF hosts.

Most of the capabilities provided by the VOS are already present in existing commercial operating systems or in commercial or public domain libraries available for such systems. It is certainly possible to assemble a functional reduction and analysis system by starting with the facilities provided by a particular host OS, obtaining a few libraries, and building the rest of the software locally. This is the approach most organizations have followed, and it certainly would have been a lot easier (and less controversial) for us to do the same rather than construct an entire virtual operating system as we did.

The chief problem with the off-the-shelf approach is of course that the resulting programming environment is unlikely to be very portable and would very likely be incomplete, forcing applications software to bypass the environment and use host facilities to get the job done. Furthermore, it is hard to produce a consistent, efficient, well engineered *system* by patching together independently developed subsystems, even if the individual subsystems are very good considered all by themselves (and often they are not, nor are they often in the public domain). These problems typically scale as some large power of the size of the system being developed. The off-the-shelf approach shows results sooner, but in the long run it costs far more, particularly if the planned system is large and has to be maintained in numerous configurations on numerous host machines.

The approach we have adopted results in a better system which is easier to port initially to a new machine (because the host interface is small, well isolated, and well defined), and which is much easier to support once the initial port has been carried out. The VOS subsystems are often quite large and are expensive to develop, but they do exactly what we want, fit into the system just right, and once they have been developed they become a permanent fixture in the environment requiring little or no maintenance, freeing our limited resources for interesting new projects.

6.2 The File I/O (FIO) Subsystem

At the heart of the VOS i/o subsystem is **FIO**, the file i/o interface. FIO makes a distinction between two broad classes of file types, text files and binary files. The type of a file must be specified at open time, but once a file has been opened file i/o is device independent. FIO supports a wide range of **standard devices**, e.g., disk resident text and binary files, terminals, magtapes, line printers, IPC (interprocess communications channels), static files (can be preallocated and mapped into virtual memory), network communications channels, the pseudofiles (see below), and text and binary memory-buffer files. Device drivers for **special devices** may be dynamically loaded at run time by applications programs, hence the FIO interface (and all programs which use FIO) may be used to access any physical or abstract device. For example, an applications program may interface an image display device as a binary file and then use IMIO to access the display.

Text files are stored on disk in the host system text file format, e.g., in a format acceptable to host system text file utilities such as an editor or file lister. Reading or writing a text file implies an automatic conversion between the IRAF internal format and the host system format. The internal format is a stream of ASCII characters with linefeed characters delimiting each line of text (as in UNIX). The text file abstraction is required in a portable system to be able to use the host utilities on text files generated by the portable system, and vice versa.

Binary files are unstructured byte stream arrays; data is written to and read from a binary file without any form of conversion. There are two subclasses of binary files, the **streaming** binary files, and the **random access** binary files. The streaming files can only be read and written sequentially; examples are IPC and magtape. Random access binary devices are assumed to have a fixed device block size which may differ for each device. A binary device is characterized by device dependent block size, optimum transfer size, and maximum transfer size parameters read dynamically from the device driver when a file is opened on the device. By default FIO configures its internal buffers automatically based on the device parameters, but the buffer size for a file may be overridden by the user program if desired.

FIO supports a special builtin type of file called the **pseudofile**, a binary streaming file. The pseudofile streams are opened automatically by the system when a task is run. The pseudofile streams of interest to applications programs are STDIN, STDOUT, and STDERR (the standard input, output, and error output streams), and STDGRAPH, STDIMAGE, and STDPLOT (the standard vector graphics, image display, and plotter streams). These streams are normally connected to the terminal, to a graphics device, or to a file by the CL when a task is run. The user may redirect any of these streams on the command line. Pseudofile i/o is multiplexed via IPC to the CL process whence it is directed to the physical device, graphics subkernel, or file connected at task initiation time. Graphics frames output to STDGRAPH are spooled in a buffer in the CL process so that the user may later interact with the graphics output in *cursor mode* (§6.6).

The top level FIO procedures are stream oriented. The FIO **user interface** is a simple open-close, getc-putc, getline-putline, read-write-seek, etc. interface which is quite easy to use. Character data may be accessed a character at a time or a line at a time; terminal i/o is normally a line at a time but a **raw mode** is provided as an option (this is used for keystroke driven programs such as screen editors). Binary data may be read and written in chunks of any size at any position in a file. On random access devices a seek call is required to position within the file. FIO handles record blocking and deblocking, read ahead and write behind, etc., transparently to the applications program. An asynchronous, unbuffered, block oriented, direct to user memory interface is also provided for applications with unusual performance requirements (for binary files only).

6.3 FMTIO, MEMIO, TTY, VOPS, ETC

The **formatted i/o** interface (FMTIO) is concerned with formatting output text and decoding input text. The primary high level stream oriented procedures *scan, fscan, printf, fprintf, sprintf*, etc., are modeled after the UNIX facilities for which they are named. A set of low level string oriented procedures provide a variety of numeric encode/decode functions, a set of general string operator functions, some lexical analysis functions, and a general algebraic expression evaluation function. The FMTIO numeric conversion routines fully support indefinite valued numbers (INDEF).

The **memory i/o** interface (MEMIO) provides a dynamic memory allocation facility which is heavily used throughout the IRAF system. Both **heap** and **stack** facilities are provided. The high level heap management procedures *malloc, calloc, realloc*, and *mfree* are modeled after the comparable UNIX procedures, although there are some minor differences. An additional procedure *vmalloc* is provided to allocate buffers aligned on virtual memory page boundaries. A pair of procedures *begmem* and *fixmem* are provided to dynamically adjust the working set size at runtime, or to simply query the amount of available physical memory if the working set cannot be adjusted. This is used to dynamically tune large-memory algorithms to avoid thrashing. The stack procedures are used mainly to simulate automatic storage allocation, with the advantage that the amount of space to be allocated is a runtime rather than compile time variable. MEMIO relies heavily upon the pointer facility provided by the SPP language.

The terminal capabilities database interface (TTY) provides a basic screen management capability for terminals. The TTY interface uses the Berkeley UNIX **termcap** terminal database, which supports dozens of terminals and which is easily extended by the user. The database capabilities of the TTY interface are also used for the line printer interface and for the IRAF **graphcap** database, used to store device dependent information describing the various graphics terminals, image displays, and plotters supported by IRAF.

The **vector operators** interface (VOPS) is a large library of subroutines, each of which performs some simple operation on one or more one dimensional arrays. Operators provided include the arithmetic operators, sqrt, power, abs, min, max, reciprocal, the trig functions, a full matrix of type conversion operators, fill array, clear array, memory to memory copy, a set of boolean operators, sort, statistical functions (median, average, etc.), rejection mean, weighted sum, lookup table operations, vector interpolation, inner product, vector sum, sum of squares, various linear transformations, convolution, fourier transform operators, and so on. The VOPS operators are written in a generic dialect of the SPP language and are expanded into a full set of type specific operators by the **generic preprocessor** before compilation and insertion into the VOPS library. A full range of datatypes is supported for each operator, including type complex where appropriate.

Using the conditional compilation facilities provided by *mkpkg*, selected VOPS operators may be hand

optimized in assembler or host specific Fortran (e.g., using Fortran vector extensions on vector machines) without compromising the portability of the system. Similarly, selected VOPS operators might be implemented in an array processor on a host which has one; ideally the array processor should be tightly coupled to the cpu for this to be worthwhile (a shared memory interface using MEMIO support is possible). The VOPS operators are used heavily throughout IRAF with the expectation that **vector machines** will become increasingly common in the future.

The ETC package is the catch-all for those VOS facilities too small to warrant full fledged package status. Major ETC subpackages include the **process control** facilities, used to spawn and control connected subprocesses and detached processes, the **exception handling** facilities, used to trap interrupts, post signal handlers, etc., and the **environment** (logical name) facility. ETC also contains the **date and time** facilities, the **device allocation** facilities, a general purpose **symbol table** facility, and a number of other subpackages and miscellaneous system procedures. IRAF relies upon the environment facilities to map virtual filenames to host filenames and to assign logical names to physical devices. The VOS automatically propagates the environment and current default directory to connected subprocesses.

6.4 The Command Language I/O (CLIO) Subsystem

The CL is almost completely invisible to the applications program. The CLIO interface consists of little more than a set of get/put procedures for CL parameter i/o. Parameters may be accessed either by name or by the offset of the parameter in the command line. A task may query the number of positional parameters on the command line, or whether a particular pseudofile stream has been redirected on the command line.

The CLIO interface is very simple at the applications level; all of the complexity and power of the interface is hidden behind the CLIO interface in the CL itself. Parameter requests may be satisfied either directly by the applications process, i.e., when it is run outside the CL, or by the CL at task invocation time or while the task is executing. The CL (i.e., the user) determines how a parameter request is satisfied transparently to the applications program. Some parameter requests result in interactive queries, others are satisfied immediately without a query. If a task repeatedly requests the same CL parameter, a different value may be returned for each request, allowing tasks to be used interactively. By assigning a text file containing a list of values to such a parameter, the user may run such tasks in batch mode. The graphics and image display cursors are implemented as CL parameters, and cursor input may be either interactive (causing cursor mode to be entered) or batch (input is taken from a text file assigned to the cursor type parameter by the user).

6.5 The Image I/O (IMIO) Subsystem

The IMIO interface is used to access bulk data arrays or *images* (rasters, pictures) normally stored in random access binary files on disk. An **image** consists of an N-dimensional array of **pixels** and an associated **image header** describing the physical and derived attributes of the image. Arbitrary user or applications defined attributes may be stored in the image header. The present interface supports images with from zero to seven axes. There are no builtin limits on the size of an image since all data buffers are dynamically allocated. The datatype of the pixels in an image may be any SPP datatype, i.e., *short* (signed 16 bit integer), *long*, *real*, *double*, or *complex*, or the special disk only datatype *ushort* (unsigned 16 bit integer).

IMIO is primarily a conventional binary file i/o type of interface. While it is possible to map all or portions of an image into **virtual memory** if the host system supports such a facility and if a number of runtime conditions are met, all current IRAF applications use only the conventional binary file i/o access method. This is necessary for full portability (a true virtual memory machine is not required to run IRAF) and furthermore is the most flexible and efficient type of access for the majority of our image processing applications. While there are some difficult image analysis applications which benefit significantly from the use of virtual memory, most applications access the entire image sequentially and can easily be programmed using binary file i/o. Sequential whole image operators are most efficiently implemented using binary FIO; the heavy page faulting resulting from sequential image access via a virtual memory interface places a greater load on the system. More importantly, the price of using virtual memory is the loss of *data independence*, which greatly limits the flexibility of the interface.

While IMIO imposes certain restrictions upon the logical representation of an image as seen by an applications program, there are few restrictions on the physical storage format, and indeed IMIO is capable of supporting multiple disk data formats, including site dependent formats if desired. The primary restriction on the physical storage format is that images are assumed to be stored in a noninterleaved **line storage mode**, i.e., like a Fortran array, although the image lines may be aligned on device block boundaries if desired. While no other storage modes are supported by the current interface, we hope to add support for band interleaved, binary nested block (BNB), etc. storage modes in the future. An efficient implementation of the BNB storage format which preserves data independence will probably require language support.

The IMIO **user interface** consists primarily of a set of procedures to get/put image lines and subrasters. The

high level IMIO routines are written in generic SPP and a version of each i/o procedure is available for each SPP datatype, allowing programs to be written to deal with any single datatype or with multiple datatypes. The IMIO interface will automatically coerce the datatype of the pixels when i/o occurs, if the type requested by the applications program does not match that on disk.

Much of the flexibility and efficiency inherent in the IMIO interface derives from the fact that pixel data is buffered internally in IMIO, returning a *pointer* to the buffered line or subraster to the user, rather than copying the data to and from the user buffer. This makes it possible for IMIO to return a pointer directly into the FIO buffer if all the right conditions are met, avoiding a memory to memory copy for the most efficient possible i/o. Leaving the buffer management to IMIO also makes the interface easier to use.

IMIO provides a number of optional features which make certain common types of image applications easier to code. The number of input line buffers may be set by the user to some value greater than one, allowing the use of a **scrolling region** for filtering applications. A program may optionally reference beyond the boundary of an image, with IMIO using the specified **boundary extension** technique (nearest neighbor, constant value, reflect, wrap around, etc.) to generate values for the out of bounds pixels. This is useful for convolution or subraster extraction applications to avoid having to deal with the boundary region as a special case.

Perhaps the most novel, most popular, and most useful feature of IMIO is the built in **image section** capability. Whenever the user enters the name of an image they may optionally append an image section to specify the subset of pixels in the image to be operated upon. For example, if image `pix` is a 512 square, 2-dimensional image, then `pix[*,-*]` is the same image flipped in Y, `pix[*,55]` is a one dimensional image consisting of line 55 of the image, `pix[19:10,50:69:2]` is a 10 by 10 subraster obtained by flipping in X and subsampling by 2 in Y, and so on. If `cube` is a three dimensional image, `cube[*,*,5]` is band 5 of the image cube (a two dimensional subimage), `cube[*,5,*]` is the XZ plane at Y=5, and so on. The image section is processed by IMIO when the image is opened, transparently to the applications program, which sees what appears to be a smaller image, or an image of lesser dimensionality than the original. The image section facility is automatically available for *any* program that uses IMIO, and is only possible by virtue of the data independence provided by the interface.

6.6 The Graphics I/O (GIO) Subsystem

For many scientific applications programs, fast interactive graphics is the key to a good user interface. High quality graphics hardcopy is likewise essential for presenting the final results of data analysis programs. These requirements are the same both for vector graphics applications and for image processing applications, and ideally the same interface should serve both types of applications. Not everyone has ready access to an image display, so it should be possible to run software intended for use with an image display device on a graphics terminal. Likewise, it should be possible to overlay vector graphics on an image display, even if the graphics program was intended for use on a graphics terminal. While an interactive cursor driven graphics interface is desirable for interactive reductions, one should not be forced to use a program interactively, hence the graphics system should allow any cursor driven graphics program to be used noninteractively as well. Lastly, since a great variety of graphics and image display devices are in use and more are being developed every day, the graphics system must make it as easy as possible to interface to new devices and to support multiple devices.

These were the primary performance requirements which the IRAF graphics i/o subsystem (GIO) was designed to meet. GIO provides a uniform, device independent interface for graphics terminals, graphics workstations, raster and pen plotters, laser printers, and image display and image hardcopy devices. GIO is one of the largest subsystems in IRAF, and is unlike most of the IRAF interfaces in that it is not completely self contained, but rather is designed to make use of existing non-IRAF graphics packages such as GKS, CORE, NCAR, and so on. Nonetheless, GIO does provide all of the software necessary to meet its primary requirement of providing fast interactive graphics for IRAF applications normally run on a graphics terminal. GIO can be interfaced to virtually any graphics terminal without modifying or writing any software, and without relinking any executables.

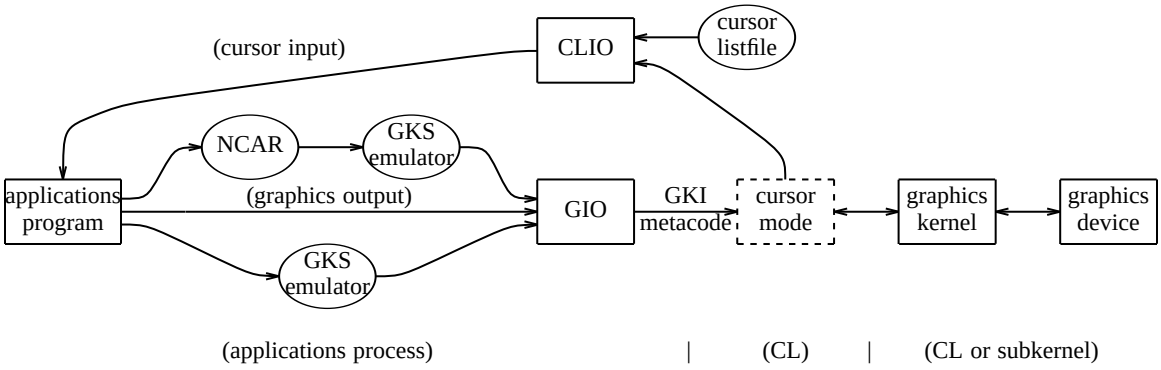

Figure 10. GIO Dataflow

The major components of the GIO subsystem and the flow of data between them (graphics output and cursor input) are shown in Figure 10. A different, somewhat simplified view emphasizing the process structure is given in Figure 2. The first thing to note is that normally only a portion of the graphics system is linked into an applications program. This reduces the size of applications programs, makes it possible to add support for new graphics devices without relinking the system, increases concurrency on single user systems, reduces physical memory requirements on multiuser systems (since multiple users can share the memory used by the graphics kernel process), reduces startup time (since the same kernel process can be used by many tasks), and reduces the need to worry about memory utilization in the graphics kernels, since the kernel has an entire process to itself.

Applications programs normally contain only the device independent, output oriented part of the graphics system. This includes any high level graphics packages such as the NCAR utilities and the GKS emulator, the GIO axis drawing and labelling code, and that part of GIO which transforms vectors input in world coordinates into clipped NDC (normalized device) coordinates. The graphics output of an applications program consists of GKI metacode, a device and machine independent stream of 16 bit signed integer graphics instruction opcodes and data.

The GKI opcodes, as well as the lowest level GIO interface procedures available to the programmer, resemble the graphics primitives of the GKS standard, i.e., polyline, polymarker, polytext, fill area, cell array, and so on. The GIO **programmer interface** includes several layers of higher level calls based on these primitives, providing everything likely to be needed by applications software, e.g., autoscaling routines, coordinate transformations, multiple world coordinate systems including optional log scaling in either axis, both relative and absolute drawing commands (these build up polylines internally), mark drawing routines, vector plotting routines, the standard axis drawing and labelling routines, and so on.

The primary component of the GIO **user interface** is the **cursor mode** facility. The graphics system makes a clear distinction between graphics output and cursor input. Often the task which reads the graphics cursor is different than that used to generate the graphics output. When a graphics frame is output, the world coordinate systems (WCS) associated with the frame and all or part of the frame itself (a stream of GKI metacode instructions beginning with a screen clear) is saved in a cursor mode *frame buffer* in the CL process.

Sometime later the cursor position may be read by the task which generated the frame, by a different task, or by the user by typing a command into the CL. This causes cursor mode to be entered; cursor mode is terminated when the user types a lower case or nonalphanumeric key on the terminal. The cursor position is returned encoded as a string consisting of the fields X, Y, WCS number, key typed, and an optional character string entered by the user. While in cursor mode the user may zoom and pan the buffered frame, repaint the screen, print the cursor position in world coordinates, draw axes around the current window into the buffered frame, annotate the frame, save the frame in a metacode file or reload the frame from such a file, take a "snapshot" of the frame on a plotter device, and so on. Cursor mode reserves the upper case keystrokes for itself, leaving the lower case keystrokes and most of the nonalphanumeric characters for the applications program.

The GKI metacode output by an applications program is normally transmitted via IPC or the network interface to the CL process and then on to a graphics kernel, which may be linked directly into the CL process or which may reside in a subkernel, i.e., in a CL subprocess connected upon demand by the pseudofile i/o system. GKI metacode may also be spooled in a file by specifying the graphics output device **vdm** (virtual device metafile), by redirection of the graphics stream on the command line, or by running the applications process outside the CL with the graphics stream redirected into a file. A variety of utilities are provided for operations upon metacode files, e.g., for decoding the GKI instructions in a metacode file (useful for debugging), for extracting frames from a metacode file, for

printing a directory of the frames in a metacode file, for generating a new metacode file wherein each frame contains a mosaic of N of the frames in the input metacode file, and so on. Spooled metacode may be used as input to any graphics kernel to make plots on any device supported by that kernel.

All of the pieces of the graphics subsystem thus far discussed have been device independent. The device dependent part of the graphics system is the GIO **graphics kernel**. The function of a graphics kernel is to convert a stream of GKI metacode instructions into device instructions to control the device or to perform i/o to the device. Since all WCS to NDC coordinate transformations and clipping are handled by the device (and kernel) independent GIO software, the graphics kernel sees only integer NDC coordinates in the range 0 to 32767. The graphics kernel is an independent module in the system, and GIO may support any number of distinct graphics kernels. A GIO kernel may be a direct interface to a particular device, or an interface to an external graphics library which may support any number of physical or logical devices.

The IRAF system includes one graphics kernel which is completely portable and hence available on any system. The STDGRAPH (standard vector graphics) kernel is used for interactive graphics on the user's graphics terminal. To provide the fastest possible response for interactive applications, the STDGRAPH kernel is linked directly into the CL process. The STDGRAPH kernel is capable of i/o to virtually any graphics terminal which has a serial interface. A **graphcap** entry for the device must be provided to tell the STDGRAPH kernel the characteristics of the device, e.g., how to encode a pen motion command, the resolution of the device, how to plot text, the number of hardware text fonts available, and so on. Tektronix compatible terminals are the most common, but the graphcap facility is general enough to describe most other terminals as well (in fact, the more smarts the terminal has the better). A graphcap entry is a runtime table typically consisting of less than a dozen lines of text; new entries can easily be added by the user.

A GIO kernel is implemented as a *library*, consisting of a pair of open-kernel and close-kernel subroutines, plus one subroutine for each GKI instruction. The GKI interface (graphics kernel interface) may be used to call the kernel subroutines either directly, i.e., if the kernel is linked into the same process as the program using GIO, or indirectly via a GKI metacode data stream transmitted via pseudofile i/o if the kernel resides in a different process. All GIO kernels are also installed in the system linked into compiled IRAF tasks callable either as **subkernels** by the pseudofile i/o system, or by the user as a conventional CL task. When called as a subkernel the GIO kernel reads metacode from a pseudofile stream; when called as a CL task the kernel reads metacode from a file.

The IRAF system currently (March 86) provides kernels for the old NCAR system plot package, for the Calcomp graphics library, and for the SUN-3 implementation of the proposed CGI standard. Eventually, GIO kernels should be available for GKS, CORE, and possibly other standard graphics libraries as well. If a kernel is not already available for the host system and graphics devices used at a particular site, it should not be difficult to generate a new kernel by modifying one of the existing ones. Often it should only be necessary to relink one of the GIO kernels supplied with the system with the local GKS, Calcomp, etc. library to get a functional kernel. As a last resort, a new GIO kernel can be built to talk directly to a specific physical device.

The current GIO subsystem supports vector graphics and batch plotter devices quite well, but has not yet been used extensively for **imaging devices** because there is no standard graphics interface for these devices. A standard set of Fortran callable subroutines for interfacing to imaging devices is currently being defined by an international consortium of astronomical centers. Our intention is to build a GIO kernel which uses this device independent image interface as soon as the interface definition is complete. Implementations of the interface subroutines for the half a dozen or so types of image displays used at NOAO are also planned. Once the image display interface subroutines are defined and a GIO kernel which uses them is in place, users will be able to interface new image devices to IRAF by implementing the standard subroutines, relinking a few executables, and adding a graphcap entry for each new device.

For more detailed information on the design of the GIO subsystem, including specifications for the interface subroutines, for the graphcap facility, and so on, the reader is referred to the document *Graphics I/O Design* (March 85), which is available from the author.

6.7 The Database I/O (DBIO) Subsystem

The DBIO subsystem is the only subsystem in the original VOS design remaining to be implemented. DBIO will be used for image header storage, for intermodule communication in large packages, and for the storage of large catalogs such as those produced by analysis programs as well as existing astronomical catalogs. DBIO will be essentially a record manager type interface. The related CL package DBMS will provide a relational database interface to catalogs and other data maintained under DBIO. The planned database subsystem is a major facility comparable in size and complexity to the existing graphics subsystem. The reader is referred to the document *Design of the IRAF Database Subsystem* (draft, October 85) for additional information on DBIO and DBMS, including a discussion of some of the potential applications of database technology to astronomy.

6.8 Networking Facilities

The portable IRAF system includes support for network access to any physical resource resident on a remote node, including disk binary and text files, magtape devices, terminals, image displays, printer/plotters, and even subprocesses and batch queues. Since this facility is provided by the portable IRAF system the network nodes do not have to run the same operating system. It is permissible for the nodes to be architecturally incompatible computers, provided the higher level IRAF systems or applications software maintains data externally in a machine independent format.

The broad scope of the IRAF networking facilities is made possible by the fact that all access to host system resources in IRAF has to go through the IRAF kernel, a part of the host system interface (§7.3). The IRAF networking capability is provided by a VOS package called the **kernel interface** (KI). The KI is a sysgen option in IRAF and is not required to run the system on a single node. The relation of the KI to the rest of the VOS and to the IRAF kernel is illustrated in Figure 11.

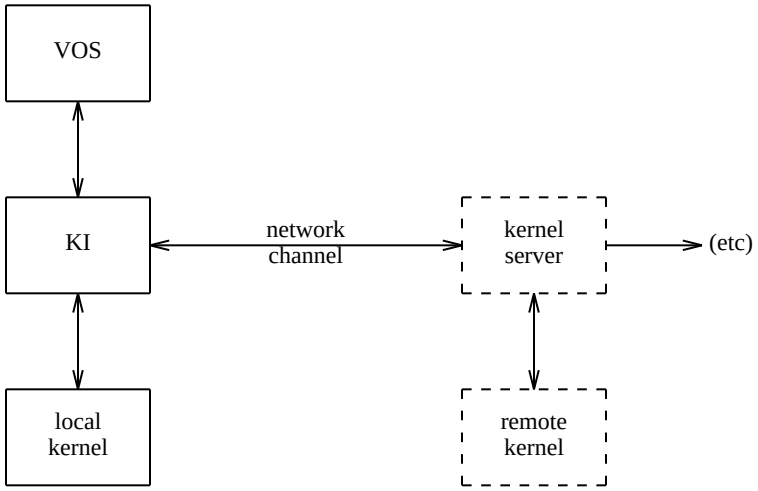

Figure 11. The Kernel Interface

In an IRAF system configured without networking, the VOS code directly calls the procedures forming the IRAF kernel for the local node. In a system configured with networking, the VOS code calls instead KI procedures which are functionally equivalent to the regular kernel procedures. If the kernel resource resides on the local node the KI procedure merely calls the corresponding kernel procedure on the local node, hence the KI adds a fixed overhead of one procedure call when it is present in a system but not used to access remote nodes. If the kernel resource resides on a remote node, the KI encodes the procedure call in a machine independent format and passes it to a **kernel server** on the remote node via a data stream network interface, returning any output arguments to the local VOS via the same interface.

A remote resource is referenced by prefixing the resource name with the node name and an exclamation character, i.e., *node!resource*. For example, the command "`page lyra!dev$graphcap`" might be entered to page the `graphcap` file on node `lyra`. Logical node names may be defined to avoid introducing site dependent information into resource names in portable code. When the first reference to a resource on a remote node is received the KI "connects" the remote node, i.e., it spawns a kernel server process on the remote node. The kernel server remains connected until the client process on the local node terminates, or until an i/o error occurs on the KI channel. In the current implementation of the KI, each client process on the local node requires a dedicated kernel server process on the remote node. A future implementation of the KI may permit a single server process to serve an entire process tree on the local node.

The beauty of the kernel interface is that since it intercepts all kernel requests it automatically provides such exotic network services as the ability to interactively access a remote graphics device, or to spawn and interactively run a subprocess on the remote node, without requiring any changes to the VOS or to applications software. Furthermore, implementation of the KI required no changes to the IRAF kernel (which is unaware that the KI exists), and the KI software itself is portable, as is the kernel server task. The only machine dependent software required is a FIO binary file driver capable of "opening" a "file" (spawning a kernel server process) on a remote node, and providing bidirectional binary communications with the remote server.

The network interface is currently in regular use at NOAO for remote image display, plotter, and file access between VAX nodes running both UNIX and VMS, using a TCP/IP network interface and Ethernet hardware. For example, a user on node A might make a line plot of an image resident on node B, enter cursor mode, and use the "snapshot" facility to dump the plot to a laser printer on node C. We have not yet made use of the remote process and batch queue capabilities. A DECNET interface also exists and will soon be tested in a MicroVax to mainframe configuration.

7. The Host System Interface (HSI)

7.1 Major Components of the Host System Interface

The host system interface (HSI) is the interface between the portable IRAF system and a particular host operating system. While the HSI contains all of the machine dependent or potentially machine dependent code in IRAF, much of the code in the HSI is actually fairly portable. To port IRAF to a new operating system one must implement the HSI. Once the HSI has been implemented for a new OS, the entire VOS and all of the IRAF system packages and NOAO science packages will in principle compile and run without modification (in reality, some testing and bug fixes are always required). Note that once IRAF has been *ported* to a new host OS, i.e., once the HSI has been implemented for a particular host OS, one must still configure the site and device dependent tables for a particular host to *install* IRAF on that host.

The HSI currently consists of the following components. The IRAF **kernel** is a subroutine library containing all the host dependent primitive functions required by the VOS, and is usually the most machine dependent part of the HSI, and the major item to be implemented in a port. The **bootstrap utilities** are a set of utility programs required to compile and maintain the main IRAF system; these are written in C and are mostly portable (they use the kernel facilities when possible). The **hlib** library is a directory containing a number of host and site dependent compile and run time tables used to parameterize the characteristics of the host system. The **as** directory contains the source for any library modules which have been written in assembler for one reason or another; while IRAF currently requires only one assembler module, any library module may be hand optimized in assembler if desired, without compromising the portability of the system. Lastly, the **gdev** directories contain the host dependent i/o interfaces for any binary graphics devices supported by otherwise machine independent GIO kernels. Often it is possible to write a portable (but device dependent) GIO kernel if the i/o functions are factored out into a separate interface.

7.2 The Bootstrap Utilities Package

The bootstrap utilities are required to compile and maintain the rest of the IRAF system. Since the bootstrap utilities must run before IRAF does, they are implemented as *foreign tasks* callable either from the host system or from the CL. Since the bootstrap utilities are required to compile and maintain the VOS as well as the rest of the portable system, they do not use the VOS facilities. Rather, they use a special *bootlib* library which requires some direct access to host facilities but which mostly uses the kernel facilities.

generic	The generic preprocessor
mkpkg	The "make package" library and package maintenance tool
rmbin	Removes binary files in a directory tree
rmfiles	Removes classes of files in a directory tree
rtar	Reads TAR format tape or disk files
spp	The XC compiler for the SPP language
wtar	Writes TAR format tape or disk files
xyacc	YACC compiler-compiler for SPP (requires UNIX license)

Figure 12. The Bootstrap Utilities

The bootstrap utilities are summarized in figure 12. The major utilities are the *mkpkg* program and the *xc* compiler; both of these are required to compile and maintain the portable IRAF system. The *rmbin* and *rmfiles* utilities are used to strip all binaries from the system prior to a full sysgen, or to strip all sources from a runtime system to save disk space. The *tar* format reader/writer programs are used to transport directory trees between IRAF systems running on different host operating systems. For example, one might use *wtar* to make a source only archive of a package in a disk file on a UNIX node, push the file through the network to a VMS node, unpack the archive with *rtar*, and compile and link the new package with *mkpkg*, all without any knowledge of the contents of the package and without editing any files (we do this all the time). The *xyacc* utility is used to make SPP parsers. This utility is not needed other than on our software development machine, since the output of the utility is an SPP module which can be compiled and used on any IRAF host.

7.3 The IRAF Kernel

The IRAF kernel (also known as the OS package) is a library of fifty or so files containing a number of Fortran callable subroutines. The kernel procedures may be written in any language provided they are Fortran callable; all current IRAF kernels are written in C. As far as possible, IRAF is designed to implement all complex functions in the VOS, making the kernel as simple as possible and therefore easier to implement for a new host. The kernel is a well defined, well isolated interface which can be implemented according to specifications without any knowledge of the rest of the system. The current 4.2BSD UNIX/IRAF kernel contains 5900 lines of C code (something like three percent of the full system), half of which is probably in the various FIO device drivers. The IRAF kernel is discussed in detail in the document *A Reference Manual for the IRAF System Interface* (May 84).

Conclusions

The IRAF system provides a large and steadily growing capability for the reduction and analysis of astronomical data, as well as a general purpose image processing and graphics capability useful for image data of any type. The system itself is nonproprietary and no proprietary external libraries are required to run IRAF. IRAF is a machine and device independent system, hence is easily ported to many current machines as well as to future machines. IRAF provides a complete modern programming environment suitable for general software development and for scientific software development in particular.

IRAF has been designed from the beginning with the capabilities of the next generation of computers in mind, hence the system is designed to make use of the vector hardware, networking facilities, bit-mapped graphics displays, large memories, and personal workstations expected to become increasingly available during the next decade. The system has been designed and implemented to a consistently high standard, and the combination of a modern design and many advanced capabilities, plus a high degree of efficiency, portability and device independence insure that the system will continue to grow in capability and use in the years to come.

Acknowledgments

The author wishes 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 (Peter Shames, Tom McGlynn, Jim Rose, Fred Rommelfanger, Cliff Stoll, and Jay Travisano).

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 activites. In particular, the support and encouragement of Buddy Powell, Harvey Butcher, and Garth Illingworth was of critical importance during the first years of the project. In recent years the support of John Jefferies, Steve Ridgway, and Ethan Schreier has been invaluable. Mention should also be made of Don Wells, who in 1978 started in motion the process which eventually led to the creation of the IRAF system.

References

The references listed here pertain only to the IRAF system software. Unless otherwise noted, all papers are by the author. These are mostly design documents; comprehensive user documentation for the programming environment is not yet available. Considerable additional documentation is available for the IRAF system packages and for the NOAO and STScI science packages. Contact the responsible institution directly for information on the science software.

1. Shames, P.M.B, and Tody, D., *A User's Introduction to the IRAF Command Language Version 2.0*, revised February 1986. The current user's guide to the CL.
2. *Detailed Specifications for the IRAF Command Language*, January 1983. The original CL design paper. No longer accurate or comprehensive, but still contains useful information about the inner workings of the CL.
3. *IRAF Standards and Conventions*, August 1983. Coding standards, program design principles, portability considerations for programming in the SPP environment.
4. *A Reference Manual for the IRAF Subset Preprocessor Language*, January 1983. The most up to date documentation currently available for the SPP language proper.

5. *The Role of the Preprocessor,* December 1981. The original design document for the SPP language. Primarily of historical interest. Documents the reasoning which led to the decision to use a preprocessor language in IRAF.

6. *Programmer's Crib Sheet for the IRAF Program Interface,* September 1983. Summarizes the contents of the various i/o subsystems comprising the VOS. Somewhat out of date, but still useful.

7. *Graphics I/O Design,* March 1985. Specifications for the graphics i/o subsystem. Reasonably up to date.

8. *Design of the IRAF Database Subsystem,* draft, October 1985. Presents the conceptual design of the planned database subsystem.

9. *A Reference Manual for the IRAF System Interface,* May 1984. An essential document describing the IRAF kernel, including the principles of operation and specifications for the kernel routines.

10. *UNIX/IRAF Installation and Maintenance Guide,* March 1986.

11. *VMS/IRAF Installation and Maintenance Guide,* March 1986.

12. *A Set of Benchmarks for Measuring IRAF System Performance,* March 1986. Contains comparative benchmarks for IRAF running on VAX/UNIX, VAX/VMS (750,780,8600), the SUN-3, and additional machines in the future.