A Reference Manual
for the
IRAF System Interface

Doug Tody

National Optical Astronomy Observatories*

May 1984

ABSTRACT

The IRAF system interface is the interface between the transportable IRAF system and the host operating system. An overview of the software design of the IRAF system is presented, naming the major interfaces and discussing their relationships. The system interface is shown to consist of a language interface, the subset preprocessor (SPP), and a procedural interface, the IRAF kernel. The reasoning which led to the choice of a Fortran preprocessor for the language interface is reviewed. A reference manual for the IRAF kernel is presented, followed by the detailed technical specifications for the kernel procedures.

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

Contents

1. **Introduction** 1

2. **Structure of the IRAF System Software** 2

3. **The IRAF System Interface** 3
   - 3.1. The Language Interface 4
     - 3.1.1. Fortran 4
     - 3.1.2. Mixing C and Fortran in the same System 5
     - 3.1.3. Critique of C as a Scientific Language 7
     - 3.1.4. The IRAF Subset Preprocessor Language 8
     - 3.1.5. Limitations of the Subset Preprocessor 9
   - 3.2. Bootstrapping the System 9
   - 3.3. The IRAF Kernel 10
   - 3.4. The Virtual Machine Model 10
     - 3.4.1. The Minimal Host Machine 11
     - 3.4.2. The Ideal Host Machine 11

4. **A Reference Manual for the IRAF Kernel** 13
   - 4.1. Conventions 14
   - 4.2. Avoiding Library Conflicts 15
   - 4.3. File I/O 15
     - 4.3.1. Text Files 16
     - 4.3.2. Binary Files 17
     - 4.3.3. Specifying Device Parameters 19
     - 4.3.4. Standard File Devices 20
       - 4.3.4.1. The User Terminal 20
       - 4.3.4.2. The Line Printer Device 21
       - 4.3.4.3. Interprocess Communication 22
       - 4.3.4.4. Imagefile Access 23
       - 4.3.4.5. Magtape Devices 25
   - 4.4. Filename Mapping 28
     - 4.4.1. Virtual Filenames 28
       - 4.4.1.1. Logical Directories and Pathnames 28
       - 4.4.1.2. Filename Extensions 29
     - 4.4.2. Filename Mapping Algorithm 30
   - 4.5. Directory Access 33
   - 4.6. File Management Primitives 33
   - 4.7. Process Control 34
     - 4.7.1. Overview and Terminology 34
     - 4.7.2. Synchronous Subprocesses 35
     - 4.7.3. Standard IPC Commands 38
     - 4.7.4. Example 40
     - 4.7.5. Background Jobs 41
     - 4.7.6. The Process and IRAF Mains 42
       - 4.7.6.1. The Process Main 42
       - 4.7.6.2. The IRAF Main 44
     - 4.7.7. Process Control Primitives 45
   - 4.8. Exception Handling 46
   - 4.9. Memory Management 47
   - 4.10. Procedure Call by Reference 49

4.11. Date and Time 49
4.12. Sending a Command to the Host OS 49

5. **Bit and Byte Primitives** 50
5.1. Bitwise Boolean Primitives 51
5.2. Bitfield Primitives 51
5.3. Byte Primitives 51
5.4. Vector Primitives 53
5.5. MII Format Conversions 53
5.6. Machine Constants for Mathematical Libraries 54

6. **System Parameterization and Tuning** 55

7. **Other Machine Dependencies** 55
7.1. Machine Dependencies in the CL 55

8. **Specifications for the Kernel Procedures** 56

A Reference Manual for the IRAF System Interface

Doug Tody

National Optical Astronomy Observatories*
May 1984

1. Introduction

The IRAF system libraries currently total 42 thousand lines of code. The applications software adds another 75 thousand lines of code, about half of which was imported (i.e., the graphics utilities and math library routines). The UNIX implementation of the machine dependent portion of the IRAF system consists of some 3300 lines of code, or about 3 percent of the current system. The remainder of the system, i.e., approximately 97 percent of the current system, is machine and device independent. It is this 3 percent of the IRAF system which is machine dependent, known as the **system interface**, which is the focus of this document.

The importance of maximizing the transportability of a large software system cannot be overemphasized. The IRAF system is required to run on a variety of different computers and operating systems from the time of its first release to the end of its useful life. The computer the IRAF system is being developed on is already old technology. Two years from now when IRAF is a mature system, it will almost certainly contain 20 to 30 manyears of software. With the increasing dependence on computers for scientific data analysis, and the demand for increasingly powerful software, it is no longer possible to keep up with demand if we have to throw out our systems every 5 or 10 years and start over.

Commercially developed operating systems/programming environments such as UNIX and ADA offer some hope for the future. At present, however, ADA is not widely available and there are at least half a dozen versions of UNIX in use, with no clear cut standard yet to emerge. The different versions of UNIX resemble each other, but there are many differences. UNIX has been steadily evolving for ten years and there is no reason to expect that the process will stop now.

Many manufacturers offer machine dependent extensions to get around the inefficiencies of basic UNIX, and it is desirable to be able to take advantage of these in a production system. Even in a hardcore UNIX system like 4.2BSD, significant efficiency gains are possible by taking advantage of the peculiarities of the 4.2BSD kernel, e.g., by always doing file transfers in units of the machine page size, into buffers aligned on page boundaries. In a large system it is difficult to take advantage of such machine peculiarities unless the system has a well defined and well isolated interface to the host system. There is no denying the fact that despite the many attractive aspects of UNIX, it is nice to have the option of switching to a more efficient operating system if IRAF is to be used in a production environment.

Perhaps the most powerful argument is that despite the increasingly widespread use of UNIX, many of the sites for which IRAF is targeted do not or can not run UNIX on their current computers. The increasing availability of transportable operating systems will make transporting IRAF easier, but is no substitute for a well defined and well isolated system interface.

The **system interface** is the interface between the machine independent IRAF software and the host operating system. The system interface consists of a procedural interface, the IRAF **kernel**, and a language interface, the IRAF Subset Preprocessor language (SPP). Both

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

types of interface are required to isolate the IRAF software from the host system.

We first present an overview of the structure of the IRAF system, naming the major interfaces and explaining their functions. The system interface is introduced and the role it plays in the full system is described. The choice of an implementation language for IRAF is next discussed, concentrating on the role the language interface plays in addressing the transportability problem. Next we define the kernel and discuss its attributes. The conceptual model of the host operating system, or **virtual machine model** assumed by IRAF is presented. The design and functioning of the kernel is discussed in detail, followed by the detailed specifications (**manual pages**) for the subroutines constituting the actual interface.

It is not necessary to carefully read the first part of this document to implement the IRAF system interface for a new host operating system. The first part of this document (the **reference manual**) concentrates on the role the system interface plays in IRAF, the principles underlying the design of the interface, and on how and why the interface came to be what it is. The second part of the document (the **manual pages**) presents the detailed specifications for the individual routines comprising the system interface, and is intended to be self contained. The reference manual tells what modules the interface consists of, how they work together, and why they are designed the way they are. The manual pages tell precisely *what* each routine does, with no attempt to explain why, or how the routine fits into the system.

Interfacing to new **graphics devices** is likely to be one of the most difficult problems to be solved in porting the IRAF system. Nonetheless the graphics device interface is not (or should not be) part of the system interface, and is not discussed here. Ideally, a graphics device will be interfaced to IRAF file i/o as a binary file. The device should have a data driven interface, rather than a control interface, i.e., all device control should be effected by inserting control instructions in the data stream transmitted to the device, rather than by calling system dependent subroutines. The software required to drive the device should be device independent, table driven, and fully portable. Virtually all graphics devices either fit this model, or can be made to fit this model by writing system dependent device drivers to interpret metacode generated by the high level, machine and device independent software.

2. Structure of the IRAF System Software

The major components of the IRAF system are the high level applications and systems **programs** and **packages**, the Command Language (CL), also known as the **user interface**, the **program interface** (library procedures called by programs), and the **system interface**. The system interface is further subdivided into the **kernel** and the **language interface**. Other system modules not relevant to our discussion are the math library and the graphics device interfaces.

The relationship and relative importance of these modules is strongly dependent upon one's point of view; all points of view are equally valid. From the point of view of the (highly idealized) user, looking at the system from the top level, the user is at the center of the system and is in command. The CL appears to be the central piece of software, and applications packages are mere extensions of the CL. To the extent that the user is aware of the host system, the CL appears to be the interface between the user and the host system. The user expects the system to behave predictably and reliably, and be responsive to their commands. To a first approximation, the user does not care what language programs are written in, or how they are interfaced to the host system (real users, of course, are often programmers too, and do care).

From the point of view of the applications programmer, the program they are writing is at the center of the system and is in command (and indeed the program does control the system when it is run). The programmer sees only the abstract problem to be solved and the environment in which the solution must be constructed, defined by the program interface and the SPP language. The CL is an abstract data structure, accessed via the CLIO library in the program interface. The fact that there is a host system underlying the program interface is irrelevant, and is of no concern to the applications programmer. As far as the applications programmer is concerned, the CL could be CL1, CL2, a file, the host OS command interpreter, Forth, Lisp, or anything else. Ideally, the applications programmer does not know that the target language of the SPP is Fortran.

From the point of view of the systems programmer, the kernel is the center of the system, with the host operating system below and the program interface above. The system software is subservient to the program which calls it, and does exactly what it is told to do and nothing more. The CL and the SPP compiler are *applications programs* which use only the facilities of the program and system interfaces.

The structural design of the IRAF software is outlined in the figure below. In general, control and parameters flow downward and data flows both both downward and upward (mostly upward). The procedures at one level do not call procedures at a higher level, e.g., kernel routines are not permitted to call procedures in the program interface libraries. A procedure may call other procedures at the same level or in the next lower level, but calls to routines more than one level lower are avoided, i.e., a program should not bypass the program interface and talk directly to the kernel. IRAF applications programs *never* bypass the system interface to talk directly to the host system.

Structure of the Major IRAF Interfaces

```
user interface
        high level program
                program interface
                        system interface
                                host operating system
```

This structure chart illustrates only the control hierarchy of the major interfaces. Additional structure is imposed on the actual system. Thus, the CL uses only a small portion of the facilities provided by the program interface, calling external programs to perform most functions. Few system or applications programs, on the other hand, use the process control facilities, the part of the program interface most heavily used by the CL (apart from file i/o). The CL and external applications programs are minimally coupled, using only human readable text files for interprocess communication (interprocess communication is implemented as a special file in IRAF).

These restrictions tend to result in more functional programs which can be combined in many ways at the CL level, and which can be used quite productively without the CL. In particular, any IRAF program can easily be debugged without the CL, using the host system debugger, and any IRAF program can be used productively on a system which cannot support the CL. The system perceived by the user at the CL level can easily be extended or modified by the user or by a programmer.

The **process structure** of the IRAF system, with the CL serving up and networking all processes, while fundamental to the design of the IRAF system, is not relevant to a discussion of the system interface because it is all handled above the system interface, i.e., in the machine independent code. Many quite different process structures are possible using the one IRAF system interface. Concentration of most or all of the complex logic required to implement process control, the CL process cache, the CL/program interface, pseudofiles, multiprocessing, i/o redirection, exception handling and error recovery, efficient file access, etc., into the machine *independent* code was a major goal of the IRAF system design.

3. The IRAF System Interface

As has already been noted, the IRAF system interface consists of both a procedural interface or kernel (library of Fortran callable subroutines), and a language interface (preprocessor for Fortran). All communication with the host system is routed through the kernel, minimizing the machine dependence of the system and providing scope for machine dependent optimizations. Similarly, all code other than existing, imported numerical Fortran procedures is processed through the language interface, further minimizing the machine dependence of the system. The language interface, or subset preprocessor (SPP), isolates IRAF programs from the peculiarities of the host Fortran compiler, provides scope for optimization by making it possible to take advantage of the nonstandard features of the compiler without compromising transportability, and does much to correct for the defects of Fortran as a programming language.

3.1. The Language Interface

The kernel alone is not sufficient to solve all the problems of transporting a large software system. There remain many problems associated with the programming language itself. The many problems of transporting even purely numerical software are well known and we will not attempt to discuss them in detail here. The reasoning which led to the decision to implement the IRAF system in a Fortran preprocessor language (SPP) is less obvious, and is probably worth recounting. In the process of retracing the logic which led to the decision to develop SPP, we will come to understand the purpose of the language interface.

For some reason programming languages are one of the most controversial topics in all of programming. No language is perfect, and it is important to try to objectively gauge the advantages and disadvantages of a language for a particular application. It is difficult to make such comparisons without the injection of opinion, and for that we apologize. In the final analysis the choice of a language is probably not all that important, provided the cost of the project is minimized and the resultant code is reliable, portable, readable, and efficient. Only the Fortran and C languages are discussed; these were the only candidates seriously considered in 1982, when the decision to implement the IRAF system in a Fortran preprocessor language was made.

3.1.1. Fortran

Consider the typical scientific applications program. Such a program may need to talk to the CL, access files, access images, access databases, dynamically allocate memory, generate graphics, perform vector operations, and so on. These are all functions provided by the program interface. In addition, in a scientific program there will often be some central transformation or numerical computation which will almost certainly be performed by a Fortran subprogram. There is an enormous amount of high quality Fortran numerical and graphics software available, both commercially and in the public domain, which IRAF programs must have access to. A third of the current IRAF system consists of highly portable, numerical (no i/o) Fortran code, all of it in the public domain.

To be useful, these general purpose, numerical Fortran subroutines must be integrated into an IRAF program to perform some specific function. Here we run into the first serious problem: while Fortran is great for numerical procedures, it has many defects as a general purpose programming language. When it comes to complex systems software, Fortran is vastly inferior to a modern programming language such as C or Pascal. It is very difficult to implement complex nonnumerical applications in standard Fortran; the temptation to use a manufacturer's nonstandard language extensions is often too difficult to resist, and a nonportable program results. Clearly, Fortran is not the language of choice for the IRAF system software, in particular the program interface.

The next question is what to do about the high level part of the scientific applications program, the part which talks to the program interface, performs applications specific functions, and eventually calls the numerical Fortran procedures. In a large scientific package Fortran subprograms will almost certainly be used somewhere in the package, sometimes quite heavily, but the bulk of the software is often very similar to systems software, concerned with allocating resources, managing data structures, doing i/o of various kinds, and directing the flow of control.

This is all nonnumerical programming, for which Fortran is poorly suited. Many Fortran compilers provide nonstandard language extensions which make it easier to code nonnumerical applications in Fortran. Most applications programmers and scientists are not intimately familiar with the Fortran standard, and are more interested in getting a program working than in making it portable, and nonportable code will result. In any case, programmers and scientists should not have to struggle to code an application in a language which was designed for another purpose. We conclude that Fortran is not the language of choice for general scientific programming within a complex system.

A more serious problem with using Fortran for general scientific programming, however, is that the high level portion of an IRAF scientific program depends heavily on the capabilities of the program interface. To produce sophisticated scientific programs with minimum effort we must have a large and powerful program interface, providing the capabilities most often needed by scientific programs. We also need a large and powerful program interface for *system* programs, and in fact the capabilities required by scientific programs are not all that different than those of system programs, so clearly it is desirable if the same program interface can support both kinds of programs.

The next step is to explore the implications of the heavy dependence of a systems or scientific program on the program interface. The program interface is a large interface, consisting of a dozen subsystems containing several hundred procedures. To have a sophisticated, high level, efficient interface, it is necessary to use "include" files to parameterize argument lists, machine parameters, and data structures, we must use dynamic memory management (pointers) for buffer allocation, and something has to be done about error handling and recovery. In short, there is a lot of communication between high level programs and the program interface. This level of communication is only feasible if the program interface and the high level program are written in the *same language*. Standard Fortran provides *almost no language support* for these facilities.

To summarize our reasoning to this point:

- Fortran must be used extensively in the scientific applications.
- Fortran is not the language of choice for IRAF systems software, in particular the program interface.
- Fortran is not the language of choice for general scientific programming, because most scientific programming is nonnumerical in nature, i.e., much like systems programming.
- A single program interface should support both systems programs and scientific programs.
- The level of communication required between a high level program and the program interface requires that both be written in the same language.
- Standard Fortran provides almost no language support for include files, globally defined parameters, dynamic memory management and pointers, data structures, or error recovery. These facilities are required by both systems and applications software.

3.1.2. Mixing C and Fortran in the same System

All of our logic to this point forces us to conclude that standard Fortran just is not suitable for the bulk of the IRAF software. The complexity of large applications packages, not to mention that of the system software, would be unmanageable in straight Fortran. Nonetheless we must still face the requirement that a third or so of the system be existing, imported numerical Fortran procedures. The obvious question to be answered is, if Fortran is such a poor choice for the main programming language of IRAF, can we program in an appropriate modern structured language like C, calling the numerical Fortran functions and subroutines from within C programs?

The answer is sure we can, if our target system supports both a Fortran compiler and a C compiler, but there are serious portability implications. Furthermore, when examined closely C turns out to be not that great a language for general scientific programming either, and the defects of C cannot easily be fixed, whereas those of Fortran can.

Mixing two different languages in the same program is straightforward on many operating systems and possible with difficulty on others. No language standard could attempt to specify how its subprograms would by called by a completely different, unspecified language, so the one thing we can be sure of is that the method used will be system dependent. Even on systems where mixing subprograms from different languages is straightforward, there are many potential problems.

Argument lists present many problems. A potentially deadly problem is the fact that C is a recursive language while (standard) Fortran is not. C expects the arguments to a procedure to be passed on the stack, while Fortran was designed with static storage of argument lists in mind; static storage is somewhat more efficient. Arguments pushed on a stack are usually pushed in the reverse of the order used for static allocation. C is call by value; Fortran is call by reference. Returning a function value is trivial on many systems, but can be a problem. The Fortran standard requires that a function be called as a function (in an expression) and not as a subroutine, while C permits either type of call (all C procedures are functions).

The method used to implement Fortran character strings is machine dependent; some machines may pass two items in the argument list to represent a character string, while others pass only one. On some machines a Fortran character string is implemented with a count byte, on others an end of string marker is used. The C standard requires that a character string be delimited by a trailing zero byte. Thus, on some systems C and Fortran character strings will be equivalent; programs written on such a system are not portable to to systems where strings are implemented differently in the two languages.

Global variables are likely to be a problem, because Fortran common blocks and C external variables are quite different types of data structures. Though it would be poor programming practice to use global variables or common blocks to pass data between C and Fortran procedures, it is sometimes justified for control parameters, in particular in connection with error handling. A C include file cannot be accessed from a Fortran program.

Finally, external identifiers have a set of problems all of their own. On some systems, e.g. VMS and AOS, C and Fortran external identifiers are equivalent, and a procedure is referred to by the same name in both languages. This makes it easy to mix calls in the two languages, but can lead to serious name conflicts in libraries. On other systems, e.g. UNIX, the external identifiers generated for the two languages are *not* equivalent, and a C procedure cannot be called from Fortran unless special provisions are taken (the UNIX Fortran compiler adds an underscore to all Fortran external identifiers).

The problem is not that it is hard to mix the two languages, but that every one of the points mentioned above is a potential machine dependency which is not covered by any standard. We conclude that mixing C and Fortran in the same program is inevitably going to be machine dependent. The problem is controllable only if the number of procedures common to the two languages is small, or if some automated technique can be developed for interfacing the two languages. The former solution is straightforward, the second requires use of some form of *preprocessor* to isolate the machine dependencies.

C and Fortran should not be mixed in applications software because the number of Fortran procedures involved is potentially very large. Since the number of procedures is large, the effort required to port the applications is also potentially large, and it is the applications which change most between system releases. If the applications total, say, 200 thousand lines of code, a new release of the system occurs every 3 months, and it takes two weeks to make all the changes in the high level software necessary to port it, then we have a bad situation. Porting the system should ideally be a simple matter of reading a tape, possibly modifying a config file or two, and running diagnostics.

C and Fortran should not be mixed in the program interface (in the system libraries) because it introduces machine dependency into the program interface where formerly there was none. Less obviously, the problem discussed in §3.1.2 of communication between modules written in different languages is very serious. The modules of the program interface use global **include** files to communicate with one another, and to parameterized the characteristics of the host system. To ensure a reliable and modifiable system, there must be only one copy of these include files in the system. Furthermore, the IRAF error handling scheme, employed in all code above the kernel, is based on the use of a Fortran common and to access this from C code would be awkward and would introduce additional machine dependence.

The only remaining alternative is to write applications programs entirely in Fortran and the system software in C, using a small program interface between the two parts of the system. The problems with this approach were noted in the last section. The small program interface will inevitably prove too restrictive, and more and more scientific programs will be written as "system" programs. These programs will inevitably want to use the numerical Fortran libraries, and the problem of a large interface between two languages resurfaces at a different level in the system.

3.1.3. Critique of C as a Scientific Language

The major strengths of C as a programming language are that it lends itself well to structured, self documenting programming, has great expressive power, strong compile time type checking, tends to result in highly transportable code (when not mixed with other languages), and is efficient for a large range of applications. C has been widely used in computer science research for a decade, and many high quality systems applications are available in the public domain.

As a scientific programming language, however, C has serious shortcomings; C was not designed to be a scientific programming language. The major problem with C as a scientific programming language is that the scientific community already has such a large investment in Fortran, and it is difficult to mix the two languages, as we have already discussed. Upon close analysis we find that there are additional problems, and these deserve mention.

C does not support multidimensional arrays. C provides something similar using pointers, but the feature is really just a side effect of the generality of pointers, and is not fully supported by the language. A multidimensional array cannot be passed to a subprogram along with its dimensions as it can in Fortran. Few C compilers optimize loops, i.e., evaluate common subexpressions (such as array subscripts) only once, or remove constant expressions from inner loops. These problems can be overcome by sophisticated use of pointers, but such hand optimization of code is extra work and increases the complexity of the code. On a machine such as the Cray, which has a Fortran compiler that recognizes vector operations, the problem would be even more severe.

Char and short integer (8 and 16 bit) expressions are evaluated using integer instructions (32 bits on a VAX), and single precision floating expressions are evaluated using double precision instructions. In a tight loop, this will require the addition of a type conversion instruction to promote a variable to the higher precision datatype, and another to convert back to the original datatype after the evaluation. This alone can double the size and execution time of a tight loop. In addition, on many machines double precision floating is not well supported by the hardware and is several times more expensive than single precision floating. C does not support the **complex** datatype, important in some scientific applications.

The C language does not include support for **intrinsic** and **generic** functions. These are used heavily in scientific applications. Typed function calls are required to access the scientific functions, and to perform exponentiation. I am not aware of any C standard for the scientific functions, though most systems appear to have adopted the Fortran standard. The scientific functions, e.g., the trigonometric functions, are evaluated in double precision. This could lead to a serious degradation of performance in a large class of applications.

Despite these quite serious shortcomings, faced with the task of coding a large and complex application I would prefer C to Fortran, if I had only the two languages to choose from.

Fortunately there is a third alternative, the use of a Fortran preprocessor. This approach preserves the best features of Fortran while providing many of the nice features of a modern language such as C, and in addition allows us to provide language level support for the IRAF i/o facilities. The preprocessor approach provides a means of isolating both systems and applications code from the underlying host compiler, making portability a realistic goal.

3.1.4. The IRAF Subset Preprocessor Language

The Subset Preprocessor language (SPP) is a precursor to a full language scheduled for development in 1986. The subset language is a fully defined, self contained language, suitable both for general programming and for numerical scientific programming. The basic language is modeled after both C and Ratfor but is a distinct language and should not be confused with either. SPP is fully integrated into the IRAF system, i.e., SPP provides substantial language support for the program interface, the IRAF system itself is written in SPP, and the SPP compiler is an IRAF applications program written in SPP and using the facilities provided by the program interface (this is not true of the original preprocessor but that is not relevant to the design). The syntax of the SPP language is nearly identical to that of the IRAF command language.

The SPP language is defined in the document *A Reference Manual for the IRAF Subset Preprocessor Language*. The language provides modern control flow constructs, a wide range of datatypes, support for both system and user **include** files, a macro definition facility, free format input, long (readable) identifiers, C-like character constants and strings, Fortran-like arrays, access to the standard Fortran intrinsic and generic functions, powerful error handling facilities, and limited but adequate support for pointers, automatic storage allocation, and data structuring. Since the target language is Fortran, there is no problem calling Fortran subprograms from SPP programs or vice versa. We do require, however, that the Fortran subprograms be purely numerical in nature, i.e., no Fortran i/o is permitted.

The function of the preprocessor is to translate an SPP source file into a highly portable subset of ANSI-66 Fortran. The transformation is governed by a set of machine dependent tables describing the characteristics of the host computer and of the target Fortran compiler. These tables must be edited to port the preprocessor to a new machine; the tables are ideally the only part of the preprocessor which is machine dependent.

Even if it should turn out that all of the necessary machine dependence has not been concentrated into the tables, however, it will often be possible to port the hundreds of thousands of lines of code in the system by modifying or adding a few lines of code to the preprocessor, *because we have placed an interface between the language of the IRAF system and that of the host computer*. The language interface provides both a solution to the problem of transporting software between different contemporary machines, and protection from future changes in the Fortran language.

The principal motivation of the preprocessor approach taken in IRAF is that it provides a real solution to the transportability problem, i.e., one which does not depend upon perfect programmers. An additional incentive is that by defining our own language we can provide excellent support the IRAF i/o facilities, i.e., they can be more than just subroutines and functions.

If one has the freedom of being able to modify the programming language used by applications programs, one can do things that are impractical to do any other way. In other words, it becomes feasible to solve problems that were formerly too difficult to address. An example is the use of lookup tables to implement blocked storage of images, an alternative to line storage mode which is superior in a large class of image processing applications. To do this well requires language support, and it is unlikely that such support will be found in any standard, general purpose programming language. The SPP is intended partially to provide a path for the future development of the IRAF system, in the hope that the system will be able to evolve and be competitive with new systems in coming years.

3.1.5. Limitations of the Subset Preprocessor

No compiled language, SPP included, can guarantee transportability. SPP programs developed on one machine will normally have to tested on one or more other machines before they can be declared portable. SPP suffers from many of the same portability problems as standard Fortran, the main difference being that the output Fortran is very simple, and nonstandard language extensions are strictly controlled. Further discussion of the portability aspects of SPP programs is given in the document *IRAF Standards and Conventions*.

We hope to eventually have IRAF running on several types of machines at the development centers. New releases will be brought up and tested on several different machines before distribution. No large software package developed on a single system is portable; if it is tested on even two different systems, that is much better.

The SPP language also depends on certain side effects which are not specified in the Fortran standard, but which many other systems also depend upon and which are commonly permitted by Fortran compilers. These include:

[1] It must be possible to reference beyond the bounds of an array.

[2] It must be possible to reference a subarray in a call to a subprocedure, i.e., "call copyarray (a[i], b, npix)".

[3] The compiler should permit a procedure to be called with an actual argument of type different than that of the corresponding dummy argument (except type **character**: SPP does not use this type anywhere).

[4] It must be possible to store a machine address or the entry point address of an external procedure in an integer variable.

Common language extensions used by the preprocessor in output code include the nonstandard datatypes such as INTEGER*2, any associated type coercion functions (INT2), and the boolean intrinsic functions if provided by the target compiler.

The output of the current preprocessor is ANSI-66 Fortran only to a first approximation. The following features of Fortran 77 are also used:

general array subscripts
zero-trip do loop checking
save statement
entry statement
generic intrinsic functions (**max** rather than **amax0**, etc.)

All of these extensions are correctable in the preprocessor itself except the use of the **entry** statement, if it should ever prove necessary.

3.2. Bootstrapping the System

Since the SPP is fully integrated into the system -- the program interface is written in SPP and SPP uses the program interface, one may have been wondering how to go about getting it all set up in the first place. The basic procedure for porting IRAF to a new system is as follows. The actual procedure has not yet been fully defined and will be tailored to the individual target systems, i.e., there will be a separate installation guide and distribution package for UNIX, VMS, DG/AOS, and any other systems supported by the IRAF development team.

[1] Implement and test the kernel routines.

[2] Compile the bootstrap SPP, which has already been preprocessed for the target machine (the datatype used to implement **char** must match that expected by the kernel procedures).

[3] Edit the system dependent files **iraf.h**, **config.h**, and the preprocessor tables to define the characteristics of the host machine.

[4] Preprocess and compile the production SPP.

[5] Do a full **sysgen** of the system libraries (the program interface etc.). This takes a couple hours on the VAX/UNIX 11/750 development system; get the config tables right the first time.

[6] Run diagnostics on the system library procedures.

[7] **Make** the applications packages.

Once the system has been ported, installing a new release requires only steps 5 through 7 (sometimes step 4 may also be required), after reading the distribution tape.

3.3. The IRAF Kernel

The IRAF kernel is a set of Fortran callable subroutines. Every effort has been made to make these primitives as simple as possible; the kernel primitives provide raw functionality with minimum overhead. Ideally the kernel primitives should map directly to the kernel of the host operating system. The implementation of the kernel primitives should emphasize simplicity and efficiency; these primitives embody the machine dependence of IRAF and there is little reason to try to make them machine independent. Critical routines should be coded in assembler if a substantial gain in efficiency will result. In IRAF, *all* communication with the host system is routed through the kernel, so kernel efficiency is paramount.

With very few exceptions, applications programs and high level systems modules do not talk directly to the kernel. The kernel primitives are called only by the routines comprising the core of the IRAF **program interface**, the lowest level of the machine independent part of the IRAF system. The core of the program interface, i.e, file i/o, process control, exception handling, memory management, etc., combined with the kernel, constitute a **virtual operating system**, the heart of IRAF. The virtual operating system approach is the key to maximizing transportability without sacrificing either functionality or efficiency.

Ideally all of the machine dependence of the IRAF system is concentrated into the kernel, which should be as small and efficient as possible while offering sufficient raw functionality to support a large and sophisticated system. In practice it is possible to come quite close to this ideal, although the range of host systems on which the kernel can be implemented is finite, being inversely proportional to the richness of function provided by the kernel. We did not consider it acceptable to provide transportability at the expense of a restrictive and limited program interface, so IRAF has a large and quite sophisticated program interface which depends upon a sizable kernel. The IRAF kernel embodies much of the functionality provided by the typical minicomputer operating system, and should be implementable on a wide range of such systems.

3.4. The Virtual Machine Model

The virtual machine model is the conceptual model of the host machine assumed by the kernel. The difficulty of implementing the kernel on a given host depends on how closely the model matches the host operating system. In general, the older large batch oriented machines do not match the model well, and it will be difficult to port the full IRAF system to such a machine. At the other end of the scale are the small 16 bit minicomputers; these machines do not have a sufficiently large memory addressing range to run IRAF. In the middle are the multiprocessing, multiuser, terminal oriented supermicro and minicomputers with large address spaces and large physical memories: these are the machines for which the IRAF system was primarily designed.

Our intent in this section is to summarize the most important features of the virtual machine model. Much of the material given here will be presented again in more detail in later sections. The design of the IRAF system is such that there are actually two distinct virtual machine models, the full model and a subset model. The subset model assumes little more than disk file i/o, and will be presented first.

3.4.1. The Minimal Host Machine

Even though it may be difficult to run the *full* IRAF system on a large batch oriented (timesharing) machine, it should still be possible to run most of the science software on such a system. The IRAF system was designed such that the science software is placed in separate processes which can be run independently of the CL and of each other. These processes actually use only a portion of the full system interface; most of the system and device dependence is in the CL and the graphics control processes, i.e., in the user interface, which is what one has to give up on a batch machine.

On the typical batch oriented timesharing system, the IRAF applications programs would be run under control of the host job control language, reading commands and parameters from an input file, and spooling all textual output into an output file. The IRAF command language would not be used at all; this mode of operation is built into the present system. Little more than file i/o is required to run a program on such a system, though dynamic memory allocation is highly desirable. Exception handling and error recovery can be done without if necessary; process control is not used by applications processes. The i/o subsystems required by an applications program are CL i/o, image i/o, and database i/o, each of which is built upon file i/o.

Applications programs that produce graphics write device independent metacode instructions, rather than talking directly to a graphics device, so a graphics device interface is not required to run an applications program. Graphics output can be discarded, or the output file can be postprocessed to generate graphics hardcopy. Graphics input (cursor readback) is parameterized, so any program that normally reads a cursor can just as easily read coordinates directly from the input file.

All of the software in the science modules is Fortran or Fortran based (C is used only in the CL and in some kernels), so only a Fortran compiler is required. A C compiler is currently required to compile the command language, though it is not necessary to have such a compiler on every physical machine. We can compile the CL on one machine and distribute the object code or executables to other machines of the same type; this will be done, for example, for VAX/VMS.

If it is necessary to port IRAF to a machine which does not have a C compiler, it is feasible to code a basic, stripped down command language in SPP in a few weeks. A command language capable of executing external processes, parsing argument lists, managing parameter files, and redirecting i/o is sufficient to run most of the IRAF software, the main loss being CL scripts. Virtually all of the system and science software is external to the CL and would be unaffected by substitution of a new CL.

3.4.2. The Ideal Host Machine

The minimal IRAF target machine therefore need offer little more than file i/o and a Fortran compiler. One would have to do without a nice user interface, but it should still be possible to do science in the fashion it has traditionally been done on such machines. Fortunately, however, minicomputers of modern design are widely available today and will be increasingly available in the future. We expect that most IRAF implementations will be ports of the full system onto a modern supermicro or minicomputer. Such a host system should provide the following general classes of facilities:

- file i/o, file management
- process control
- exception handling
- memory management
- date and time
- bit and byte primitives

Most of the complexity of the kernel is in file i/o, process control, and exception handling. File management (file deletion, renaming, etc.) is straightforward on virtually any system. Memory management can be some work to implement, but many systems provide dynamic allocation primitives in which case the interface is trivial. The date and time facilities assume that the host provides some sort of facilities for reading the clock time (ideally as a high precision integer) and the cpu time consumed by a process. The bit and byte primitives do not actually do any i/o, and are included in the interface primarily because Fortran is difficult to use for such applications.

The IRAF **file i/o** system deals with two types of files. **Text files** contain only character data, are read and written in units of records (lines of text), and are maintained in a such a form that they can be edited with a host editor. Writing may occur only at the end of file. Reading is normally sequential, but seeking to the beginning of a line prior to a read is permitted. The user terminal is interfaced as a text file, opened by the IRAF main at process startup. Terminal i/o is generally line oriented, but character at a time input is used by some programs, and the system expects to be able to send control codes to the terminal. Text files are accessed synchronously. Character data is always ASCII within IRAF, with the kernel routines mapping to and from the host character set (e.g. EBCDIC).

The second file type is the **binary file**. A binary file is an extendible array of machine bytes. There are two types of binary files, **blocked** (random access) binary files and **streaming** (sequential) binary files. Transfers to and from blocked binary files are always aligned on device block boundaries and are asynchronous. The size of a transfer may be any integral multiple of the device block size, up to a device dependent maximum transfer size. The IRAF file i/o system (FIO) assumes that a file can be extended by overwriting the end of file, and that a partial record can be written at the end of file without the host filling the record to the size of a device block. The device block size is assumed to be device dependent. Devices with different block sizes may coexist on the same system.

Streaming binary files are for devices like magtapes and the interprocess communication (IPC) facilities. Seeks are not permitted on streaming files, and there are no restrictions on block size and alignment of transfers, other than an upper limit on the transfer size. The ideal host system will initiate an asynchronous transfer from either type of binary file directly from the mass storage device into the buffer pointed to by the kernel (or vice versa).

The model does not assume that the host system provides device independent file i/o. A different set of kernel routines are provided for each device interfaced to FIO. On a system which does provide device independent i/o the kernel routines may be coded as calls (or multiple entry points) to a single set of interface subroutines. Standard file devices include disk resident text and binary files, the IPC facilities, magtapes, line printers, and (currently) the image display devices and the batch plotters. The special devices are normally interfaced to FIO as binary files. The IPC files are streaming binary files.

Although it is highly desirable that the host provide a hierarchical files system, it is not required. IRAF tends to generate and use lots of small files. FIO maps virtual filenames into machine dependent filenames, and will pass only filenames acceptable to the host system to the kernel routines. It should be possible to read filenames from a host directory, determine if a file exists and is accessible, delete a file, and rename a file. The model assumes that there are two types of filenames: files within the current directory (directory name omitted), and files within a particular directory, and that both types of files can be simultaneously accessed. The directory name is assumed to be a string which can be prepended to the filename to produce a pathname. Multiple versions of a file are neither assumed nor supported.

Magnetic tape devices are interfaced as streaming binary files. A magnetic tape is either empty or consists of one or more files, each delimited by an end of file (EOF) mark, with an end of tape (EOT) mark following the last file on the tape. The kernel routine opens the device positioned to the first record of a specific file or EOT. Tape density may be manually set on the device, or may be set at allocate time or at open time (the IRAF software will work regardless of which method is used). A separate file open is required to access each file on the tape, i.e., FIO will not try to read or write beyond a tape mark.

Record and file skipping primitives are desirable for tape positioning in the kernel open procedure, but are not assumed by the model. Tape records may be variable in length. No attempt will be made to position the tape beyond EOT. FIO knows nothing about labeled tapes or multivolume tapes; if it is necessary to deal with such tapes, the details should be handled either by the host system or by the kernel routines. All IRAF programs which read and write magtape files can also be used to read and write disk files.

The virtual machine model assumes that a parent process can spawn one or more child **subprocesses**, to execute concurrently with the parent, with bidirectional streaming binary communications channels connected to the parent. The IPC facilities are used only to communicate with child processes; the model does not assume that any one process can talk to any other process. The model assumes that an IPC channel can only be opened when a child process is spawned; the two functions are bound into the same kernel primitive. A child process is assumed to inherit the same current working directory as the parent. The child is not assumed to inherit environment or logical name tables, open files, or the parent's address space.

Exception handling is very machine dependent and is difficult to model. Fortunately the IRAF system will probably still be usable even if the host does not entirely fit the model, since exceptions are not the norm. A parent process is assumed to be able to interrupt a child process. For error recovery and process shutdown to occur properly control should transfer to an interrupt handler in the child process when the interrupt signal is sent.

All exceptions occurring during execution of a process should be caught by the kernel and mapped into the exception classes assumed by the kernel. The model assumes that all exceptions can be caught, that control can be transferred to an exception handler, and that execution can resume following processing of the exception. The host and the kernel should let the high level software process all exceptions and handle error recovery.

In summary, IRAF can be used to do science on a limited, batch oriented host machine, at the expense of a limited user interface and considerable hacking of the released system. The ideal host system will provide a hierarchical files system, large asynchronous file transfers directly into process memory, multiprocessing, efficient binary interprocess communication facilities, dynamic memory management facilities, and high level exception handling.

4. A Reference Manual for the IRAF Kernel

The kernel is a set of SPP or Fortran callable subroutines. The syntax and semantics of these routines, i.e., the external specifications of the interface, are the same for all machines. The code beneath the interface will in general be quite different for different operating systems. Any language may be used to implement the kernel routines, provided the routines are Fortran callable. Typed functions are avoided in the kernel where possible to avoid the problems of passing a function value between routines written in different languages.

The method chosen to implement a kernel routine for a given host should be dictated by the characteristics of the host and by the external specifications of the routine, not by the method chosen to implement the same kernel routine for some other host. Nonetheless it is often possible to reuse code from an existing interface when coding an interface for a new host. This occurs most often in the numerical procedures, e.g., the bit and byte primitives. Furthermore, some routines are placed in the kernel only because they are potentially machine dependent; these routines need only be examined to see if they need to be modified for the new host.

The kernel routines are found in the **OS package**, the interface to the host Operating System (OS). The OS package is maintained in the logical directory **sys$os**, and the kernel routines are archived in the system library **lib$libos.a**.

4.1. Conventions

At the kernel level, data is accessed in units of **machine bytes**. The size of a machine byte in bits and the number of bytes per SPP data type are both parameterized and are assumed to be machine dependent. It is fundamentally assumed throughout IRAF that an integral number of bytes will fit in each of the language datatypes. Conversion between byte units and SPP units is handled by the high level code; the kernel routines deal with data in byte units.

All offsets in IRAF are **one-indexed**, including in the kernel routines. Thus, the first byte in a file is at offset 1, and if the device block size is 512 bytes, the second device block is at offset 513. The first bit in a word is bit number 1. Many operating systems employ zero-indexing instead of one-indexing, and the implementor must be especially careful to avoid off by one errors on such systems.

All **character strings** are packed in the high level code before transmission to the kernel, and strings returned by the kernel are unpacked by the high level code into SPP strings. The packed string is passed in an array of SPP type **char**. The format of a packed string is a sequence of zero or more characters packed one character per byte and delimited by end-of-string (EOS). SPP strings are ASCII while packed strings use the host character set. The EOS delimiter occupies one character unit of storage but is not counted in the length of the string. Thus if a kernel routine returns a packed string of at most *maxch* characters, up to *maxch* characters are returned followed by an EOS.

Kernel procedures should call only other kernel procedures or the host system. Calls to program interface routines are forbidden to avoid problems with **reentrancy** and backwards library references (and because code which references upwards is usually poorly structured). A program crash should never occur as a result of a call to a kernel procedure. Illegal operations should result in return of an error status to the routine which called the kernel procedure. The SPP error handling facilities must not be used in kernel procedures, even if a kernel procedure can be coded in SPP. This is because the high level code does not error check kernel procedures, and because printing an error message involves calls to FIO and possible reentrancy.

Any kernel procedure which can fail to perform its function returns an integer status code as its final argument. Other integer codes are used to parameterize input arguments, e.g., the file access modes. Codes which are used only in a particular procedure are documented in the specifications for that procedure. The codes used extensively in the kernel are shown in the table below. The magic integer values given in the table must agree with those in the SPP global include file **iraf.h**.

Kernel Constants

name	*value*	*usage*
ERR	−1	function was unsuccessful
EOS	'\0'	end of string delimiter
OK	0	function successfully completed
NO	0	no (false)
YES	1	yes (true)

The names of kernel routines never exceed six characters, and only alphanumeric letters are used. The Fortran implicit datatyping convention (I through N for integer identifiers) is not used in IRAF. The IRAF naming convention is package prefix plus function plus optional type suffix letter. This convention is little used in the kernel because there are few functions, but it does occur in a few places, e.g. the bitwise boolean functions **and** and **or**. The procedure naming convention and other IRAF conventions are further discussed in the document *IRAF Standards and Conventions*.

4.2. Avoiding Library Conflicts

Only documented OS interface routines should be callable from the high level SPP and Fortran code. If at all possible, all non-interface kernel subprocedures and host system calls referenced in the kernel should be named such that they are not Fortran callable. All external identifiers used in IRAF code adhere to the Fortran standard, i.e., at most six alphanumeric characters, the first character being a letter. Non-interface kernel procedures are guaranteed not to cause library conflicts with the high level software provided they are not legal Fortran identifiers.

For example, on UNIX systems, the Fortran compiler appends a hidden underscore character to all Fortran external identifiers. No standard C library or system procedures have names ending in an underscore, so library conflicts do not arise. On a VMS system, the VMS system service procedures all have external names beginning with the package prefix "sys$", hence the system service procedures are guaranteed not to cause library conflicts with standard Fortran identifiers.

If there is no way to avoid library conflicts by using a naming convention at the kernel level, it is possible to modify the SPP to map identifiers in a way which avoids the conflicts (e.g., by use of **define** statements in the global include file **iraf.h**). This approach is less desirable because it involves modification of high level code, and because it does nothing to avoid library conflicts with Fortran subprograms which are not preprocessed. Furthermore, it is important that the mapping of SPP identifiers to plain vanilla Fortran identifiers be simple and predictable, to ease interpretation of host Fortran compiler error messages, and to make the host system debugger easy to use with SPP programs.

4.3. File I/O

The file i/o subsystem is the most critical i/o subsystem in IRAF. No process can run without file i/o, and the high level system code and applications programs are all built upon file i/o. Many programs are i/o intensive, and an efficient file i/o system is vital to the functioning of such programs. The high level of functionality and device independence provided by the IRAF file i/o subsystem is critical to minimizing the complexity and maximizing the flexibility of all code which uses file i/o. In particular, the database and image i/o subsystems are heavily dependent upon file i/o; the IRAF file i/o system was designed expressly to provide the kinds of facilities required by these and similar applications.

Most of the complexity of the file i/o system is in the machine independent FIO interface. FIO handles all buffer allocation and management including management of buffer caches, record blocking and deblocking, and read ahead and write behind. FIO makes all files and file storage devices look the same, and allows new devices to be interfaced dynamically at run time without modifying the system. The database facilities rely on FIO for efficient random file access, which requires use of a buffer cache to to minimize file faults. Image i/o relies on FIO for efficient sequential file access, which requires asynchronous i/o and large buffers.

Kernel support for file i/o consists of a few simple file management primitives, e.g. for file deletion and renaming, plus the "device drivers" for the standard devices. There are two basic types of files, **text files** and **binary files**. The device driver for a text device consists of 8 subroutines; the driver for a binary device consists of 6 subroutines. A different set of driver subroutines are required for each device interfaced to FIO. All system and device dependence is hidden within and beneath these subroutines. Most devices are interfaced to FIO as binary files. Kernel device drivers are closely matched in capabilities to the actual device drivers found on many systems.

The file access modes, device parameters, and other codes used to communicate with the kernel file i/o primitives are summarized in the table below.

FIO Kernel Constants		
name	*value*	*usage*
BOF	−3	beginning of file
EOF	−2	end of file
READ_ONLY	1	file access modes
READ_WRITE	2	
WRITE_ONLY	3	
APPEND	4	write at EOF
NEW_FILE	5	create a new file
TEXT_FILE	11	file types
BINARY_FILE	12	
FSTT_BLKSIZE	1	device block size
FSTT_FILSIZE	2	file size, bytes
FSTT_OPTBUFSIZE	3	optimum transfer size
FSTT_MAXBUFSIZE	4	maximum transfer size

4.3.1. Text Files

A **text file** is a sequence of lines of text, i.e., of characters. Examples of text files are parameter files, list files, program source files, and **terminals**. Although it is not strictly required, it is desirable that text files be maintained in such a form that they can be accessed by the host system file editor and other host utilities. The principal function of the text file primitives is to convert text data from the host format to the IRAF internal format, and vice versa.

The physical representation of a text file is hidden beneath the kernel interface and is not known to an IRAF program. The logical (IRAF) and physical (host system) representations of a text file will in general be quite different. On some systems it may be possible to represent a single logical text file in any of several different physical representations, and the kernel primitives will have to be able to recognize and deal with all such representations. On other systems there may be only a single physical format for text files, or there may be no distinction between text files and binary files.

The **logical representation** of a text file is a sequence of lines of text. Each line of text consists of zero or more ASCII characters terminated by the **newline** character. The newline character defaults to ASCII LF (linefeed), but some other character may be substituted if desired. IRAF assumes that any ASCII character can be stored in a text file; in particular, case is significant, and control characters may be embedded in the text. There is no fixed limit on the number of characters per line. It may not be possible to edit a file containing arbitrarily long lines or embedded control characters with some host system editors, but such files are rare.

Character data is represented within IRAF with the SPP datatype **char**. On many systems, char is implemented as the (nonstandard) Fortran datatype INTEGER*2. The read and write primitives for a text file must convert SPP arrays of ASCII char to and from the internal host representation. This conversion usually involves a packing or unpacking operation, and may also involve conversion between ASCII and some other character set, e.g., EBCDIC. Regardless of the precision of the datatype used to implement char on a given host system, characters are limited to ASCII values, i.e., 0 to 127 decimal (negative valued characters are permitted only in SPP variables and arrays).

The kernel primitives used to access ordinary disk resident text files, i.e., the "device driver" primitives for an ordinary text file, are shown below. The calling sequences for other text file devices are identical if the two character device code for the new device is substituted for the "tx" suffix shown. A device driver is installed in FIO by passing the entry points of the subroutines to FIO with **fopntx** or **fdevtx**; the entry point addresses of the 8 subroutines are saved in the FIO device table.

The **zopntx** primitive opens a text file or creates a new one, returning the channel number (an integer magic number) or ERR as its status value. All subsequent references to the file are by this channel number. The file access modes are listed in the table in §4.3. Output text is assumed to be buffered; **zflstx** is called by FIO to flush any buffered output to the device when the file is closed, or when file output is flushed by the applications program.

Text File Primitives

zopntx (osfn, mode, chan)	open or create a textfile
zclstx (chan, status)	close a textfile
zgettx (chan, text, maxch, status)	get next record
zputtx (chan, text, nchars, status)	put next record
zflstx (chan, status)	flush output
znottx (chan, loffset)	note file position
zsektx (chan, loffset, status)	seek to a line
zstttx (chan, param, lvalue)	get file status

If the physical file is record oriented, there will normally be one newline delimited line of text per record. A sequence of characters output with **zputtx** is however not necessarily terminated with a newline. The **zputtx** primitive is called to write the FIO output buffer when (1) newline is seen, (2) the buffer fills, (3) the output is flushed, or (4) the file is closed. Thus if a very long line is written, several calls to **zputtx** may be required to output the full line. Conversely, if the input record contains more than *maxch* characters, **zgettx** should return the remainder of the record in the next call. In no case should more than maxch characters be returned, as the output buffer would be overrun.

A **zgettx** call with **maxch=1** has a special meaning when the input device is a terminal. This call will switch the terminal from line mode to **character mode**, causing **zgettx** to return immediately each time a key is typed on the terminal. This highly interactive mode is useful for programs like screen editors, and is discussed further in §4.3.4.1. Character mode applies only to terminal input; if individual characters are to be output, the output must be flushed after each character is written.

Text files are virtually always accessed sequentially. Writing is permitted only at end of file (EOF). A file opened for reading is initially positioned to the beginning of file (BOF). Seeking to the beginning of any line in the file is permitted prior to a read. The seek offset must be BOF, EOF, or an offset returned by a prior call to the **znottx** primitive, which returns the file offset of the last text line read or of the next text line to be written. Seeks on text files are restricted to lines; seeks to individual characters are not permitted. The long integer file offset returned by **znottx** is a magic number, i.e., the value is assumed to be machine dependent.

4.3.2. Binary Files

A **binary file** is an extendible array of bytes accessed in segments the size of a device block. The principal difference between a text file and a binary file is that character data is converted in some machine dependent fashion when a text file is accessed, whereas data is copied to and from a binary file without change. A second difference is that only binary files are randomly accessible for both reading and writing at any offset. Binary files are used to implement interprocess communication (IPC) files, database files (datafiles), picture storage files (imagefiles), and so on. Special devices such as magtape, the line printer, image display devices, and process memory are interfaced to FIO as binary files.

The fundamental unit of storage in a binary file is the **machine byte**. The number of bits per byte is presumed to be machine dependent, although IRAF has thus far been used only on machines with 8 bit bytes. IRAF assumes only that there are an integral number of bytes in each SPP or Fortran datatype. The SPP datatype **char** should not be confused with the machine byte. The char is the fundamental unit of storage in SPP programs; the number of machine bytes per SPP char is greater than or equal to one and is given by the machine dependent constant SZB_CHAR, defined in **iraf.h**. Though the distinction between chars and machine bytes is important in the high level system code, it is of no concern in the kernel since the kernel routines access binary files only in units of machine bytes.

Binary files are further differentiated into **blocked** (random access) binary files and **streaming** (sequential) binary files. The most common blocked binary file is the binary random access disk file. IPC files and magtape files are typical streaming binary files. The **device block size** is used to differentiate between blocked and streaming binary files. A device with a block size greater than or equal to one byte is understood to be blocked, whereas a device with a block size of zero is understood to be a streaming file. Transfers to and from blocked devices are always aligned on device block boundaries. There are no alignment restrictions for streaming files.

Binary File Primitives

zopnbf (osfn, mode, chan)	open or create a binary file
zclsbf (chan, status)	close a binary file
zardbf (chan, buf, maxbytes, loffset)	initiate a read at loffset
zawrbf (chan, buf, nbytes, loffset)	initiate a write at loffset
zawtbf (chan, status)	wait for transfer to complete
zsttbf (chan, param, lvalue)	get file status

The kernel primitives used to access ordinary disk resident binary files, i.e., the "device driver" primitives for a binary file, are shown above. The calling sequences for other binary file devices are identical if the two character device code for the new device is substituted for the "bf" suffix shown. The device driver for a binary file is particularly simple since all buffering is performed at a high level. A binary file is opened or created with **zopnbf**, which returns the channel number or ERR as its final argument. All subsequent references to the file are by channel number.

The kernel primitives for a binary file closely approximate the functionality of the typical host system device driver. Ideally an asynchronous kernel read or write to a binary file will translate into a DMA transfer directly from the data buffer to the device, or vice versa. FIO guarantees that only a single transfer will be pending on a channel at a time, i.e., that a new i/o request will not be issued until any previous transfer is completed. There is no **seek** primitive for binary files since the absolute file offset is specified in a read or write request (the file offset argument should be ignored for a streaming file). There is no **note** primitive since there is no concept of the current file position for a binary file at the kernel level.

The **zardbf** and **zawrbf** (asynchronous read and write) primitives should initiate a transfer and return immediately. No status value is returned by these primitives: rather, the number of bytes read or written or ERR is returned in the next call to **zawtbf** (asynchronous wait). A byte count of zero on a read indicates end of file. It is not an error if fewer than **maxbytes** bytes can be read; **zardbf** should return immediately with whatever data it was able to read, rather than try to read exactly maxbytes bytes. In no case should more than maxbytes bytes be returned, as this would overflow the caller's buffer. If the size of the input block or record is greater than maxbytes bytes when reading from a streaming file, data is assumed to be lost. An attempt to read or write before BOF or after EOF is illegal and will be caught by FIO.

The status value ERR should be returned for all illegal requests or i/o errors. FIO will always call **zawtbf** after a transfer for synchronization and to check the status value. Repeated calls to **zawtbf** after a single i/o transfer should continue to return the same status. Errors should not "stick", i.e., an error status should be cleared when the next transfer is initiated.

It is fundamentally assumed that it is possible to extend a file by overwriting EOF in a call to **zawrbf**. It is further assumed that the last block in a file need not be full. For example, suppose the device block size is 512 bytes, the FIO buffer size is 512 bytes, and we are writing to a file 1024 bytes long. We write 18 bytes at file offset 1025, the fourth block in the file, and then close the file. When the file is subsequently reopened, FIO will call **zsttbf** to get the file size, which should be 1042 bytes. If the program calling FIO then attempts to write 50 bytes at EOF, FIO will call **zardbf** to initiate a read of 512 bytes at file offset 1025, and a subsequent call to **zawtbf** will return a byte count of 18. FIO will copy the 50 bytes into the buffer starting at byte offset 19 and eventually write the buffer to the file at file offset 1025, overwriting EOF and extending the file.

4.3.3. Specifying Device Parameters

Each device interfaced to FIO has a unique status primitive callable while the file is open to obtain values for the device parameters. These parameters reflect the characteristics of the *device* or *filesystem* on which the file is stored. The status primitives for disk resident text and binary files are **zstttx** and **zsttbf**. These primitives are the only file status primitives available for a file while it is open; **zfinfo** is not called to get information on open files.

The device block size and file size parameters are currently not used for text files, although they are read when the file is opened. The file size parameter is not needed for text files because **zsektx** is used to seek to EOF on a text file. All four parameters are required for binary files.

FSTT_BLKSIZE

> The device block size in bytes (binary files only). A block size of zero indicates a streaming device; no alignment checking will be performed, and only sequential i/o will be permitted. If the block size is greater than or equal to one, the device is understood to be a random access binary file with the indicated device block size. File reads and writes will be aligned on device block boundaries, although if the block size is given as 1 byte (e.g., if process memory is accessed as a file) there is effectively no restriction on block alignment.

FSTT_FILSIZE

> The file size in bytes; zero should be returned for a new file. Not used for streaming files. FIO will ask for this once, when the file is opened, if the file is a regular disk resident binary file. Thereafter FIO keeps track of the file size itself. If necessary the kernel can get the file size from the host system before opening the file.

FSTT_OPTBUFSIZE

> The optimum transfer or buffer size for "normal" file access. This parameter defines the default FIO buffer size for both read and write access. The optimum transfer size typically depends on the characteristics of the i/o system of the host, and may also depend on the characteristics of the device or of the file system on which the file is found. For example, the optimum transfer size of a file system configured for the storage of images (large files) may well be larger than that for a file system configured for general use (predominantly small files).

FSTT_MAXBUFSIZE

The maximum permissible transfer size in a single read or write request. This parameter determines the maximum FIO buffer size, and hence the maximum size of a FIO read or write request.

The optimum buffer size for magtape devices is usually different (larger) for reading than for writing; the default magtape buffer sizes are set by system tuning parameters in **config.h** (§6). FIO automatically adjusts the internal FIO buffer size for a file to be an integral multiple of the device block size. The FIO buffer size may be further controlled at a high level by advising FIO that i/o to a file is to be highly random or highly sequential. With all this going on in the high level code, it is inadvisable to try to tune the system by adjusting the device parameters. The file status parameters should reflect the physical characteristics of the device or files system on which the file is resident.

For example, consider a random access binary disk file on a UNIX system. The device block size will typically be 512 bytes and is generally wired into the status primitive as a constant. The file size may be obtained at any time from the inode for the file. The optimum buffer size is 512 bytes on V7 UNIX, 1024 bytes on 4.1BSD, and dependent on how a filesystem is configured on 4.2BSD. There is no maximum transfer size for a disk file, so the maximum integer value is returned. If the file happens to be a pipe, the block size would be given as 0, the file size is ignored, the optimum transfer size is arbitrary, e.g. 2048 bytes, and the maximum transfer size is typically 4096 bytes.

4.3.4. Standard File Devices

The kernel routines for the ordinary text and binary disk files have already been presented. An arbitrary number of other devices may be simultaneously interfaced to FIO. The standard devices are disk, memory, terminal, line printer, IPC, magtape, and the pseudofiles (STDIN, STDOUT, etc.). The memory and pseudofile interfaces are machine independent and will not be discussed here. The IRAF development system currently also supports file interfaces for image display devices and plotters, but the graphics device interfaces are being redesigned to use the ISO standard Graphical Kernel System (GKS) graphics device interface, so we will not discuss those devices here.

Each device is interfaced with a distinct set of kernel interface routines to give the implementor maximum scope for tailoring the interface to a device. If the host system provides device independent file i/o at a low level, it may be possible to use the same kernel routines for more than one device. For example, the text file driver might be used for both disk resident text files and terminals, and the IPC, magtape, and line printer devices might resolve into calls to the kernel routines for a disk resident binary file. This approach offers maximum flexibility for minimum effort and should be followed if the host system permits. On the other extreme, a host might not have something fundamental like IPC channels, and it might be necessary to build the driver from the ground up using non-file resources such as shared memory.

4.3.4.1. The User Terminal

Terminal devices are interfaced to FIO as text files. The device code is "ty". The driver subroutines are shown below. The legal access modes for a terminal are READ_ONLY, READ_WRITE, WRITE_ONLY, and APPEND. Seeking to offsets other than BOF or EOF is illegal; seeks to BOF and EOF should be ignored.

Terminal Driver

zopnty (osfn, mode, chan)	open a terminal file
zclsty (chan, status)	close a terminal
zgetty (chan, text, maxch, status)	get next record
zputty (chan, text, nchars, status)	put next record
zflsty (chan, status)	flush output

znotty (chan, loffset)	not used
zsekty (chan, loffset, status)	not used
zsttty (chan, param, lvalue)	get file status

When an IRAF process is run from the CL it communicates with the CL via IPC files; when not run from the CL, an IRAF process assumes it is talking to a terminal. The terminal driver is therefore linked into every IRAF main. The main assumes that the terminal is already open when an IRAF process starts up; the **zopnty** and **zclsty** routines are used only when a terminal is directly accessed by a program.

If possible the terminal driver should be set up so that input can come from either a terminal or an ordinary text file, allowing IRAF processes to be run in batch mode taking input from a file. On a batch oriented system the "terminal" driver would be the same as the text file driver, and input would always come from a file.

Terminal input is normally line oriented. The host terminal driver accumulates each input line, handling character, word, and line deletions and other editing functions, echoing all normal characters, checking for control characters (e.g. interrupt), and returning a line of text to **zgetty** when carriage return is hit. The line returned by **zgetty** to the calling program should always be terminated by a **newline**.

If **zgetty** is called with **maxch=1** the terminal is put into raw character mode. In this mode **zgetty** returns each character as it is typed, control characters have no special significance (as far as possible), and characters are not automatically echoed to the terminal. A subsequent call with maxch greater than one causes a mode switch back to line input mode, followed by accumulation of the next input line.

The IRAF system includes device independent software for terminal control and vector graphics, and expects to be able to send device dependent control sequences to the terminal. Any program which does anything out of the ordinary with a terminal, e.g., clearing the screen or underlining characters, uses the TTY interface to generate the device dependent control sequences necessary to control the terminal. Ordinary output to the terminal, however, is not processed with the TTY interface.

The control characters commonly present in ordinary text are **newline**, **carriage return**, and **tab**. The **newline** character delimits lines of text and should result in a carriage return followed by a line feed. **Zputtx** may be called repeatedly to build up a line of text; the output line should not be broken until newline is sent. The carriage return character should cause a carriage return without a line feed. If the host system terminal driver can conditionally expand tabs, tab characters present in the text should be passed on to the host driver. The terminal should be allowed to expand tabs if possible as it is much faster, especially when working from a modem. On many systems it will be necessary to map newlines upon output, but all other control characters should be left alone.

4.3.4.2. The Line Printer Device

Line printers are interfaced at the kernel level as binary files. At the FIO level a line printer may be opened as either a text file or a binary file. If opened as a text file at the FIO level, textual output is processed under control of the device independent TTY interface, generating the control sequences necessary to control the device, then the output is packed and written to the device as a binary byte stream. If the printer is opened as a binary file at the high level, binary data is passed from the applications program through FIO and on to the kernel without modification.

Line Printer Driver

zopnlp (osfn, mode, chan)	open printer or spoolfile
zclslp (chan, status)	close printer

zardlp (chan, buf, maxbytes, notused)	initiate a read
zawrlp (chan, buf, nbytes, notused)	initiate a write
zawtlp (chan, status)	wait for transfer to complete
zsttlp (chan, param, lvalue)	get file status

The line printer is a streaming device. Currently only APPEND mode is used, although there is nothing in the FIO interface to prevent reading from a line printer device. The filename argument is the **logical name** of the printer device, as defined by the CL environment variable **printer** and as found in the **dev$termcap** file. These logical device names are quite system dependent, and in general it will be necessary to add new device entries to the termcap file, and change the name of the default printer device by modifying the **set printer** declaration in **lib$clpackage.cl**.

On some systems or for some devices it may be desirable to spool printer output in an ordinary binary file opened by **zopnlp**, disposing of the file to the host system when **zclslp** is called, or at some later time. This is desirable when the line printer device is so slow that asynchronous printing is desired, or when the printer device is located on some other machine and the spoolfile must be pushed through a network before it can be output to the device.

In general it should not be necessary to modify printer data upon output. The high level code processes form feeds, expands tabs, generates control sequences to underline characters, breaks long lines, maps newline into the device end of line sequence, pads with nulls (or any other character) to generate delays, and so on, as directed by the **dev$termcap** file. If the host system driver insists on processing printer output itself, it may be necessary to modify the termcap entry for the printer to generate whatever control sequences the host system requires (the newline sequence is likely to be a problem). The termcap entry for a printer is potentially machine dependent since raw output to a line printer may not be feasible on some systems, and it may be easier to edit the termcap file than to filter the output stream in the driver.

Part of the reason for implementing the printer interface as a binary file was to provide a convenient and efficient means of passing bitmaps to printer devices. If a bitmap is to be written to a printer device, ideally the device will be data driven and it will be possible to pass data directly to the device without translation. If this is not the case, the driver must make the device look like it is data driven by scanning the data stream for a control sequence indicating a change to bitmap mode, then poking the host driver to change to bitmap mode. Since the device is data driven at the kernel level it will still be possible to spool the output in a file and process it on a remote network node.

4.3.4.3. Interprocess Communication

Interprocess communication (IPC) channels are necessitated by the multiprocess nature of IRAF. When a subprocess is spawned by the CL (or by any IRAF process) it is connected to its parent by two IPC channels, one for reading and one for writing. An IPC channel is a record oriented streaming binary file.

IPC Driver

zopnpr (osfn, mode, chan)	not used
zclspr (chan, status)	not used
zardpr (chan, buf, maxbytes, notused)	initiate a read
zawrpr (chan, buf, nbytes, notused)	initiate a write
zawtpr (chan, status)	wait for transfer to complete
zsttpr (chan, param, lvalue)	get file status

The IPC channels are set up when a subprocess is spawned, and a process may use IPC facilities only to talk to its parent and its children (the process structure is a tree, not a graph). Since the opening of an IPC channel is bound to the spawning of a subprocess, the open and close primitives are not used for IPC files. The **prconnect** procedure (not a kernel primitive), called by the parent to spawn a subprocess, sets up the IPC channels and installs the IPC driver in FIO, returning two binary file descriptors to the calling program. The **prconnect** procedure is very much like a file **open**, except that the "file" it opens is active. The IRAF main in the child process senses that it has been spawned by an IRAF process, and installs the same IPC driver in its FIO, connecting the IPC channels to the streams CLIN and CLOUT.

Since the IPC channels are read and written by concurrent processes, some care is necessary to ensure synchronization and to avoid deadlocks. Fortunately most of the necessary logic is built into the high level protocols of the CL interface, and an understanding of these protocols is not necessary to implement the IPC driver. It is however essential that the low level protocols of an IPC channel be implemented properly or deadlock may occur.

An IPC channel is **record oriented**. This means that if **zawrpr** is called by process A to write N bytes, and **zardpr** is called by process B to read from the same channel, N bytes will be read by process B. If the IPC facilities provided by the host are sufficiently sophisticated, records may be **queued** in an IPC channel. The writing process should block when the IPC channel fills and no more records can be queued. The reading process should block when it attempts to read from an empty channel.

For example, suppose process A writes an N byte record and then an M byte record. If **zardpr** is called by process B to read from the channel, it should return to its caller the first record of length N bytes. A second call will be required to read the next record of length M bytes. On some systems, e.g. UNIX, the IPC facilities are not record oriented and the first read might return either N bytes or N+M bytes, depending on unpredictable system timing details. Hence the IPC driver for a UNIX system must impose a record structure upon the UNIX "pipe" used as the IPC channel.

On other systems the IPC facilities may be limited to the transfer of single records, i.e., process B will have to read a record before process A can transmit the next record. This is the lowest common denominator, and hence the protocol chosen for the IPC driver. Despite the use of the lowest common denominator for the low level protocol, a high bandwidth can be achieved for IPC channels if the maximum transfer size is large and records are queued. The high level protocol ensures that only one process will be writing or reading at a time, thus preventing deadlock. The high level protocol also uses special data records as semaphores to achieve synchronization and permit record queuing.

Most modern operating systems provide some sort of interprocess communications facilities which the IPC driver can use. UNIX calls it a pipe or a socket, VAX/VMS calls it a mailbox, and DG/AOS calls it an IPC port. If the host system has no such facility, or if the facility provided by the host is inefficient, an IPC driver can often be built using shared memory (use a circular buffer to implement the queue). As a last resort, a real driver can be coded and installed in the host system. On **multi-processor** systems the IPC facilities should allow the parent and child processes to reside on different processors.

4.3.4.4. Imagefile Access

Imagefiles, i.e., bulk data files, are a special kind of binary file. The ordinary disk binary file driver may be used to access imagefiles, but imagefiles have certain properties which can be exploited on some systems for increased i/o efficiency. The image i/o software (IMIO) therefore uses a special kernel driver to access imagefiles. Since this driver is a special case of the ordinary binary file driver, a transportable version of the driver which simply calls the ordinary binary file driver is included in the standard distribution. The transportable driver may easily be replaced by a machine dependent version to optimize image i/o for the host system.

Imagefiles differ from ordinary binary files in that the size of the image is known when the **pixel storage file** is created. Furthermore, images do not dynamically change in size at run time. On many systems it is possible to preallocate the pixel storage file before writing any data into it, rather than creating the file by writing at EOF.

Preallocation of a file makes it feasible for the host system to allocate **contiguous storage** for the file. Use of a preallocated, fixed size file also makes it possible on some systems to map the file into **virtual memory**. If image access is expected to be sequential, or if the host system does not support virtual memory, it is often possible to **directly access** the file via the host system device driver, bypassing the host files system software and significantly reducing the overhead of file access (e.g., eliminating any intermediate buffering by the host system).

Static File Driver

zopnsf (osfn, mode, chan)	open static file
zclssf (chan, status)	close static file
zardsf (chan, buf, maxbytes, loffset)	initiate a read
zawrsf (chan, buf, nbytes, loffset)	initiate a write
zawtsf (chan, status)	wait for transfer to complete
zsttsf (chan, param, lvalue)	get file status
zfaloc (osfn, nbytes, status)	preallocate a binary file

The use of a file i/o interface to implement virtual memory access to files is desirable to minimize the machine dependence of applications which use virtual memory. The functional behavior of the static file driver is the same whether it maps file segments into virtual memory or copies file segments into physical memory. If IRAF is to be used on a system which does not provide virtual memory facilities, the image processing software will work without modification, provided the physical memory requirements of the software are reasonable.

FIO divides a file up into segments of equal size, where the size of a segment is equivalent to the size of a file buffer and is an integral multiple of the virtual memory page size. IMIO ensures that the pixel data in the pixel storage file begins on a block boundary, and is an integral number of pages in length. Furthermore, when FIO allocates a file buffer, it ensures that the buffer is aligned on a virtual memory page boundary. The virtual memory page size is parameterized in **config.h**, and is set to 1 on a nonvirtual machine.

Provided that the buffer and the file data are both properly aligned, **zardsf** may be used to map a file segment into memory. The file buffer pages are first deleted and then remapped onto the new file segment. If the buffer is written into, **zawrsf** will eventually be called to update the segment, i.e., flush modified pages to the image file (the pages should not be unmapped). If desired a single FIO buffer may be allocated the size of the entire image and all reads and writes will reference this single buffer with minimum overhead. Alternatively the image may be mapped in segments; reusing the same buffers avoids flushing the system page cache when sequentially accessing a large image.

When an image section is read or written by IMIO, the interface returns a pointer to a buffer containing the pixels. If all of the necessary conditions are met (e.g., no subsampling, no datatype conversion, etc.), IMIO will return a pointer directly into the file buffer, otherwise IMIO extracts the pixels from the file buffer into a separate buffer. If the file buffer is mapped onto the imagefile, IMIO thus returns a pointer directly into the imagefile without performing any i/o (until the data is referenced). Thus it is possible to exploit virtual memory for image access without restricting the flexibility of programs which operate upon images; general image sections may be referenced, datatypes need not agree, etc., yet i/o will still be optimal for simple operations.

For example, suppose an entire large image is to be mapped into virtual memory. FIO does not allocate any buffers until the first i/o on a file occurs. IMIO will be called by the applications program to read a "subraster" the size of the entire image. If the image can be directly accessed, IMIO will set the FIO buffer size to the size of the image, then issue a **seek** and a **read** to read the pixels. FIO will allocate the buffer, aligned on a page boundary, then call **zardsf** which maps the buffer onto the pixel storage file.

If all the necessary conditions are met, IMIO will return a pointer into the FIO buffer and hence to the segment of memory mapped onto the pixel storage file. If this is not possible, IMIO will allocate a new buffer of its own and perform some transformation upon the pixels in the FIO buffer, writing the transformed pixels into the IMIO buffer. The IMIO pointer will be dereferenced in an subprogram argument list in the applications program, and the SPP or Fortran subprogram will see what appears to be a static array.

Most virtual memory implementations are designed more for random access than for **sequential access**. Some systems, e.g. VAX/VMS, allow a variable number of pages (the page fault cluster) to be read or written when a page fault occurs. Other systems, e.g., DG/AOS, read or write a single page for each fault. Even when the page fault cluster can be made large to minimize faulting when sequentially accessing a large image, i/o is not optimal because paging is not asynchronous, and because the heavy faulting tends to flush the process and system page caches. Thus for sequential image operations conventional double buffering with large buffers and large DMA transfers direct from disk to memory is preferable. This level of i/o is available via QIO calls on VAX/VMS and is feasible via *physio* calls on UNIX, if images are static and contiguous or nearly contiguous.

If the host system provides both virtual memory facilities and low level asynchronous i/o, the static file driver should ideally be capable of performing i/o by either technique. The choice of a technique may be based upon the alignment criteria and upon the size of the transfer. If the alignment criteria are not met or if the size of the transfer is below a threshold, conventional i/o should be used. If the size of the transfer is large, e.g., some sizable fraction of the working set size, virtual i/o should be used. Virtual memory should only be used for images which are to be accessed randomly, so the page fault cluster should be small.

4.3.4.5. Magtape Devices

The magnetic tape device interface is the most complex file device interface in the IRAF system. Operating systems vary greatly in the type of i/o facilities provided for magtape access, making it difficult to design a machine independent interface. Some systems provide primitive access to the drive, permitting file and record skipping, writing of tape marks, and so on, while others permit only sequential access in the forward direction. Magtape access is further complicated by the storage of multiple files on a tape, by variable size records, the occasional need to swap bytes, and the need to specify the density. Error recovery is particularly difficult for magtape devices because it is possible to lose track of the position of the tape: whereas most binary devices are accessed by absolute offset, magtapes are accessed relative to the current position.

To avoid having to deal with this level of complexity in the kernel, the magtape device driver has been subdivided into a machine independent part and a unique magtape device interface. The standard streaming binary file driver subroutines are coded in SPP and are portable. An inner set of six "zz" routines are defined especially for accessing magtape devices. The portable driver routines constitute the MTIO interface and are not part of the kernel. MTIO opens and initializes multiple magtape devices, keeps track of the file position, and handles error recovery. The kernel routines are responsible for physically positioning the tape and for reading and writing records.

Magtape Driver

zopnmt (osfn, mode, chan)	open a magtape file
zclsmt (chan, status)	close magtape device
zardmt (chan, buf, maxbytes, notused)	initiate a read
zawrmt (chan, buf, nbytes, notused)	initiate a write
zawtmt (chan, status)	wait for transfer to complete
zsttmt (chan, param, lvalue)	get file status

A magtape device must be **allocated** at the CL level before the device can be accessed. A file on a magtape device is opened by calling the program interface procedure **mtopen** in an applications program. Once opened, a magtape file is accessed via the FIO interface and hence benefits from the buffer management facilities provided by FIO. Use of the FIO interface also provides device independence, allowing programs which access magtape to be used to (sequentially) access any other binary file. In particular, any IRAF program which commonly accesses magtape may also be used to access a disk file. This permits use of the FITS reader and writer, for example, for image transmission between stranger machines in a local area network.

When a magtape device is allocated a device **lock file** is written into the IRAF public directory **dev$** by the high level code. In addition to telling other IRAF processes that the device has been allocated, the lock file is used to keep track of the tape position while the device is closed. When a device is closed, either normally or during error recovery, a new lock file is written recording the current position of the drive as well as various statistics (owner, time of last access, number of records read or written, etc.). When a device is opened the lock file is read to determine the current position. The **system.devstatus** program prints the contents of the device lock file; if there is no lock file, the IRAF system assumes that the device has not been allocated.

Each file on a reel must be opened individually, just as each file on a disk must be opened individually. A pair of calls to **mtopen** and **close** (FIO) are required to access each file. Drives are referred to by the logical names "mta", "mtb", and so on. The assignment of logical drives to physical devices is system dependent. The logical drive number, density, absolute file number on the tape, absolute record number within the file, access mode, and FIO buffer size (most of which can be defaulted) are specified in the file "name" argument when the file is opened. For example, the filespec "mtb1600[3,10]" refers to record 10 of file 3 of logical drive "mtb" at 1600 bpi. The minimum filespec consists of just the logical drive name; everything else is optional.

The **mtopen** procedure parses the magtape filespec to determine whether a magtape device or a disk binary file is being referenced. If a disk file is named, **mtopen** reduces into a conventional call to **open**. If a magtape file is named, the filespec is parsed to determine the logical drive, density, and the file and record numbers to which the tape is to be opened. If this information is legal, if the drive is allocated, and if the drive is not already open, **mtopen** reads the lock file to determine the current position. FIO is then called to open the device. A global common is used to pass device dependent information from **mtopen** to **zopnmt**, since device dependent information cannot be passed through FIO.

The magtape kernel primitives are shown below. Our intent here is only to introduce the routines and discuss the role they fulfill in the MTIO interface. Detailed specifications for the routines are given in the manual pages.

Magtape Kernel Primitives

zzopmt (drive, density, mode, oldrec, oldfile, newfile, chan)	open
zzclmt (chan, mode, nrecords, nfiles, status)	close
zzrdmt (chan, buf, maxbytes)	aread
zzwrmt (chan, buf, nbytes)	awrite
zzwtmt (chan, nrecords, nfiles, status)	await
zzrwmt (chan, status)	arewind

The **zopnmt** procedure is called by FIO to open a magtape device. The only legal access modes for magtape files are READ_ONLY and WRITE_ONLY. The device parameters are retrieved from the global common prepared by **mtopen** and passed on to the kernel primitive **zzopmt** to physically open the drive. The kernel open primitive sets the density of the drive and opens the drive with the desired access mode, leaving the tape positioned to record 1 of the desired file.

The exact position of the tape at open time, i.e., both file and record numbers (one-indexed), is passed to **zzopmt** to facilitate positioning the tape. Many systems can skip records and tapemarks in either the forward or reverse direction, and it is easy to position the tape on such systems given the current position and the new position. The current record number is needed to tell **zzopmt** when the current file is already rewound. On some systems it is not possible to backspace to the last tapemark, and the only way to rewind the current file or position to a previous file is to rewind the tape and read forward.

Upon input to **zzopmt**, the *newfile* argument specifies the number of the file to which the tape is to be positioned, or the magic number EOT. Upon output, *newfile* contains the number of the file to which the tape was actually positioned. The high level code assumes that the tape does not move when the device is closed and subsequently reopened; if this is not the case, **zzopmt** should ignore the old position arguments.

The **zzrdmt** and **zzwrmt** primitives initiate an asynchronous read or write of a record at the current tape position. The number of bytes read or written and the number of records and or files skipped in the operation are returned in the next call to **zzstmt**. If a tape mark is seen when attempting to read the next record, **zzwtmt** should return a byte count of zero to signal EOF. It does not matter whether the tape is left positioned before or after the tape mark, provided the record and file counts are accurate.

A file containing zero records marks logical EOT. If physical EOT is seen either the host system or the kernel should signal the operator to mount the next reel, returning only after a record has been read or written on the next reel. The high level code does not try to reread or rewrite records. Error retry should be handled either by the kernel routines or preferably by the host driver. The kernel should return ERR only if it cannot read or write a record. It is not an error if less data is read than was requested, or if more data was available in the record than was read, resulting in loss of data. Loss of data is unlikely because FIO generally allocates a large (16K or 32K) input buffer when reading magtapes.

If an error occurs when accessing the magtape device, e.g. a keyboard interrupt, the high level code will mark the position of the tape as undefined and call **zzclmt** to close the device. When the device is subsequently reopened, **mtopen** will see that the position of the tape is undefined and will rewind the tape before calling **zzopmt** to open and position the drive. Since this is handled by MTIO, the kernel routines need not be concerned with error recovery except possibly to abort a tape motion if an interrupt occurs, to prevent runaway (hopefully this will not be necessary on most systems).

The **zzclmt** primitive will be called to close the device upon normal termination or during error recovery. MTIO assumes that **zzclmt** will write an EOT mark (two tapemarks) *at the current tape position* when a tape opened with write permission is closed. This is the only way in which MTIO can write EOF and EOT marks on the tape. To avoid clobbering tapes, **zzopmt** may need to open the drive read-only while positioning the tape. Since an interrupt may occur while the tape is being positioned, **zzopmt** should return the OS channel argument immediately after the channel has been opened, before positioning the tape.

In summary, the magtape kernel primitives are highly machine independent because they permit access to only a single file at a time, reading or writing sequentially in the forward direction. No primitive tape positioning commands are required except **zzrwmt**, and that can be implemented with a call to **zzopmt** if necessary (the difference is that **zzrwmt** may be asynchronous). No assumptions are made about where the tape is left positioned if an error occurs or if a tapemark is read or written. All writing of tapemarks is left to the kernel or to the host system.

4.4. Filename Mapping

The syntax of a filename is highly system dependent. This poses a major obstacle to transporting a large system such as IRAF, since the standard distribution consists of several thousand files in several dozen directories. Many of those files are referred to by name in the high level program sources, and use of host system dependent filenames in such a context would make the IRAF system very difficult to transport.

To avoid this problem only **virtual filenames** (VFNs) are used in IRAF source files. FIO converts a VFN into a host system dependent filename (OSFN) whenever a filename is passed to a kernel routine. Conversely, when FIO reads a directory it converts the list of OS filenames in the host directory into a list of virtual filenames. The kernel routines see only machine dependent filenames packed as Fortran character constants.

While it is not necessary to study filename mapping to implement the kernel, filename mapping is a vital part of the system interface and an understanding of the mapping algorithm employed is necessary to adjust the machine dependent parameters controlling the mapping. The filename mapping parameters are given in the system configuration file **lib$config.h**.

4.4.1. Virtual Filenames

A VFN consists of three fields, the **directory**, the **root**, and the **extension**. Directories are specified by a **logical directory** name followed by a **pathname** to a subdirectory. Either the logical directory name or the pathname may be omitted. If the logical directory field is omitted the current directory is assumed. The extension field is optional and is used to specify the file type, e.g., CL source, SPP source, object module, and so on. Either the logical directory delimiter character **$** or the subdirectory delimiter character **/** may delimit the directory field, and a period delimits the root and extension fields.

dir **$** *path* **/** *root* **.** *extn*

The combined length of the root and extension fields is limited to 32 characters. The legal character set is **A-Za-z0-9_.**, i.e., the upper and lower case alphanumerics (case is significant), underscore, and period. Other characters may be permitted on some systems, but if present the filename is machine dependent. The first character of a filename is not special, i.e., the first character may be a number, underscore, or any other filename character. Purely numeric filenames are permitted. The VFN syntax does not support VAX/VMS-like version numbers. A file naming syntax would not be sufficient to emulate versions; extensive FIO support would also be required on systems other than VMS.

The following are all legal virtual filenames (avoiding directory specifications for the moment).

```
20
02Jan83
Makefile
10.20.11
M92_data.Mar84
extract_spectrum.x
_allocate
```

4.4.1.1. Logical Directories and Pathnames

The use of logical directories and pathnames is perhaps best explained by an example. Consider the VFN **plot$graph.x**, specifying the file **graph.x** in the logical directory **plot**. The logical directory **plot** is defined in the CL environment (file **lib$clpackage.cl**) as follows.

```
set   plot  = "pkg$plot/"
set   pkg   = "iraf$pkg/"
```

These definitions state that **plot** is a subdirectory of **pkg**, and that **pkg** is a subdirectory of **iraf**, the root directory of the IRAF system. The definition for the root directory is necessarily both machine and configuration dependent. On a VAX/VMS system **iraf** might be defined as follows:

set iraf = "dra0:[iraf]"

Recursively expanding the original VFN produces the following partially machine dependent filename:

dra0:[iraf]pkg/plot/graph.x

The final, fully translated, machine dependent filename is produced by folding the subdirectory names into the VMS directory prefix, mapping the remaining filename (which does not change in this case), and concatenating:

dra0:[iraf.pkg.plot]graph.x

The important thing here is that while there may be many directories in the system, *only the definition of the IRAF root directory is machine dependent.* Filenames in package script tasks, in the **help** database, in makefiles, and so on inevitably include references to subdirectories, hence the VFN syntax must recognize and map subdirectory references to fully address the problems of machine independence.

Even when porting the system to another host running the same operating system the root directory may (and usually will) change. Since all logical directories and filenames are defined in terms of the root directory, since the root is defined at runtime, and since filenames are mapped at runtime, the system may be ported to another machine running the same operating system by editing only one file, *without having to recompile the system.* The importance of not having to recompile the system becomes clear when the local hardware configuration changes or when installing periodic updates at a site with multiple host computers all running the same operating system.

Filename Mapping Primitives

zfsubd (osdir, subdir, new_osdir, maxch, nchars)	fold subdir into osdir
zfxdir (osfn, osdir, maxch, nchars)	get directory prefix
zfpath (osfn, pathname, maxch, nchars)	get absolute pathname

The primitives used to map filenames are shown above. The **zfsubd** primitive folds a subdirectory name into a machine dependent directory name (OSDIR), producing the OSDIR name of the subdirectory as output. The subdirectory name ".." refers to the next higher directory in the hierarchy, allowing upwards directory references. The form of a host directory name is undefined, hence the primitive **zfxdir** is required to extract a machine dependent directory prefix from a host filename (OSFN). Directory expansion does not guarantee that the OSFN produced is independent of the current working directory, hence the **zfpath** primitive is provided to convert OSFNs into absolute pathnames.

4.4.1.2. Filename Extensions

Filename **extensions**, like directories, pose a problem because different operating systems use different extensions for the same logical file types. Thus a Fortran source file might have the extensions ".f", ".f77", and ".for" on various systems. IRAF defines a standard set of file extensions to be used in virtual filenames. Filename extensions are mapped by string substitution when a file is referenced; unrecognized extensions are left alone. The standard extensions used in IRAF virtual filenames are essentially those used by UNIX, plus extensions for the special IRAF file types (e.g., CL script files and parameter files).

To illustrate the mapping of filename extensions, consider the IRAF system library **lib$libos.a**, which contains the kernel routines in object form. On a UNIX system this might be expanded as "**/usr/iraf/lib/libos.a**", whereas on a VMS system it might be converted to **dra0:[iraf.lib]libos.olb**.

The standard IRAF filename extensions are listed in the table below. Those which are system dependent and are normally mapped are marked at the right.

Standard Filename Extensions

Extension	Usage	Mapped
.a	Library file (archive)	**
.c	C language source	**
.cl	Command Language script file	
.com	Global common declaration	
.db	database file	
.e	executable image	**
.f	Fortran 77 source file	**
.h	SPP header file	
.hlp	*Lroff* format help text	
.ms	*Troff* format text	
.o	object module	**
.par	CL parameter file	
.pix	pixel storage file	
.x	SPP language source	

For the convenience of the user working interactively within the IRAF environment, FIO permits virtual and host system dependent filenames to be used interchangeably. Any arbitrary file or directory in the host system may be referenced as an argument to an IRAF program, even if the host directory has not been assigned a logical name. The filename mapping scheme thus provides indirect support for pathnames, by permitting the use of OS dependent pathnames to specify files when working interactively. If a machine dependent filename is given, mapping of the root and extension fields is disabled.

4.4.2. Filename Mapping Algorithm

The primary requirement for filename mapping is that the process be reversible, i.e., it must be possible to map a VFN to an OSFN and later recover the original VFN by applying the reverse mapping. The following additional requirements led to the selection of the algorithm described in this section.

- There should be no efficiency penalty for simple filenames.
- The algorithm must permit multiple processes to access the same directory without contention.
- The mapping must be transparent, i.e., reversible by inspection of the host system directory, making it easy to work with directories and files at the host system level.
- The reverse mapping (OSFN to VFN) should be efficient, i.e., there must not be a serious degradation of performance for template expansion and directory listings.
- The mapping should permit use of IRAF, with some loss of efficiency, on a computer with a flat directory system.

The algorithm selected consists of two phases. If the maximum information content of a host system filename is sufficiently large, the first phase will succeed in generating a unique mapping with no significant overhead. If the first phase fails, the second phase guarantees a unique mapping on any system with minimal overhead. The first phase maps the VFN into the

OSFN character set using **escape sequences** to map non-OSFN characters, preserving the information content of a filename by increasing its length. If the length of the OSFN thus generated exceeds the maximum filename length permitted by the host system, the second phase accesses an **auxiliary hidden file** to recover the excess information.

The operation of the mapping algorithm differs slightly depending on whether an existing file is to be accessed or a new file is to be created. The procedure followed to generate a unique OSFN when opening an existing file is outlined below. The complications caused by multiple logical directories, filename extensions, and transparency to machine dependent filenames are not relevant to the algorithm and are omitted.

```
algorithm vfn_to_osfn
begin
        # Phase one: encode VFN using only legal OSFN characters.

        map vfn to osfn using escape sequence encoding
        if (length of osfn is within host limit)
            return (osfn)

        # Phase two.  Access or read auxiliary file to get OSFN.

        squeeze osfn to legal host filename length
        if (squeezed osfn is degenerate) {
            extract unique_osfn for named vfn from mapping file
            return (unique_osfn)
        } else
            return (osfn)
end
```

Escape sequence encoding is a technique for mapping illegal characters into sequences of legal characters. A single illegal character is mapped into two legal characters. Strings of illegal characters, e.g., a sequence of characters of the wrong case, are prefixed by a font change sequence. For example, suppose the host system permits only upper case alphanumeric characters in filenames. Lower case is the dominant case in VFNs, so case will be inverted in the mapping and upper case in a VFN must be escaped. If we pick the letter Y as our escape character, the following mappings might be established (these are the defaults for VMS and AOS):

vfn	*osfn*	*usage*
y	Y0	the escape character itself
tolower	Y1	switch to primary case
toupper	Y2	switch to secondary case
_	Y3	underscore
.	Y4	period
A-Z	YA-YZ	upper case letters

The use of escape sequences can result in confusing mappings, but if the escape character is chosen carefully such cases will be rare. Most filenames are quite ordinary and will map with at most a case conversion. Some examples of filenames which do not map trivially are given below. The maximum length of a filename extension on the host system is assumed to be 3 characters in these examples. Any host limit on the maximum number of characters in the root is ignored. For the purposes of illustration, we assume that the first character of the OS filename cannot be a number, necessitating use of a no-op sequence.

20	Y120

02Jan83	Y102YJAN83
Makefile	YMAKEFILE
10.20.11	Y110Y420.11
M92_data.Mar84	YM92Y3DATAY4YMAR84
extract_spectrum.x	EXTRACTY3SPECTRUM.X
_allocate	Y3ALLOCATE
yy.tab.c	Y0Y0Y4TAB.C
README	Y2README

Escape sequence encoding will probably suffice for most filename mapping, particularly if the host system permits long filenames (e.g., AOS/VS currently permits filenames of up to 32 characters). If the encoded filename is too long for the host system, auxiliary files must be used to store the excess information. A single **mapping file** is used for the entire directory to permit efficient inverse mapping when expanding filename templates and listing directories, and to avoid wasting disk space by generating many small files.

If escape sequence encoding produces an OSFN longer than the maximum OS filename length N, then characters must be discarded to produce an N character filename. This is done by squeezing the long OSFN, preserving the first few and final characters of the root, and the first character of the extension (if any). For example, if the OSFN is CONCATENATE.PAR and N is 9, the squeezed OSFN will be CONCATEEP.PAR (the mapping is the same as that employed for long identifiers in the SPP, except that the first character of the extension is appended). Once this is done, of course, the mapping is no longer guaranteed to be unique.

More often than not a squeezed OSFN will be unique within the context of a single directory. If this is the case it is not necessary to read the mapping file to convert a VFN to an OSFN, although it is always necessary to read the mapping file to carry out the inverse transformation. If the mapping is unique within a directory, a null file with the same root name as the primary file but with the (default) extension **.zmu** is created to indicate that the mapping is unique (e.g., CONCATEEP.ZMU and CONCATEEX.ZMU). If a second file is created with the same root OSFN and the mapping is no longer unique, the **.zmu** directory entry is simply deleted, and the mapping file will have to be read whenever either file is accessed.

The utility of the **.zmu** file is based on the assumption that the determining the existence of a file is a much less expensive operation on most systems than opening, reading, and closing the mapping file. Furthermore, the **.zmu** file is a null length file, i.e., just an entry in a directory, so no disk space is wasted. Use of the advisory **.zmu** file does however involve an assumption that the host system permits filename extensions. If this is not the case, set the maximum length of a filename extension to zero in **config.h**, and FIO will not generate the files.

The mapping file is effectively an extension of the directory and hence will lead to contention problems when **concurrent processes** try to access the same directory. A process must not be allowed to read the mapping file while another process is modifying it, and under no circumstances may two processes write to the mapping file at the same time. This requires that a process which wishes to modify the mapping file place a lock on the file before accessing it, and that a process be capable of waiting if the mapping file is locked. The mapping data must not be buffered, i.e., the file should be reread every time a (degenerate) file is accessed. Fortunately contention should be rare since most file accesses to not require use of the mapping file.

In summary, the overhead of the filename mapping algorithm should be insignificant when (1) accessing files with simple names, and (2) accessing files with long names for which the mapping is unique. A small but fixed overhead is incurred when a file with a long name is created or deleted, when a directory is read, and when a file is accessed for which the mapping is degenerate. If the host computer has a decent files system the algorithm will incur negligible overhead for all operations.

4.5. Directory Access

The capability to read filenames from a host directory is required for the expansion of filename templates and for the directory listing program. At the program interface level a directory appears to be a simple text file, i.e., an unordered list of virtual filenames. A directory file is opened at the applications level with **diropen** and successive VFNs are read with **getline**. The driver procedures used to interface a directory to FIO as a text file are machine independent. The kernel primitives called to read OS filenames from a directory are machine dependent and are summarized below.

Directory Access Primitives

zopdir (osfn, chan)	open a directory
zcldir (chan, status)	close directory
zgfdir (chan, osfn, maxch, status)	get next OSFN

Directory files are read-only and are accessed sequentially. A single filename is returned in each call to **zgfdir** as a packed string, returning as status the length of the string or EOF. Filenames may be returned in any order; all filenames in the directory should be returned (there are no "hidden" files at this level). Raw OS dependent filenames should be returned. The inverse mapping from OSFN to VFN is carried out in the machine independent code.

If the host system does not permit direct access to a directory file, or does not provide a primitive which returns successive filenames, it may be necessary to read the entire contents of the directory into a buffer at **zopdir** time, returning successive filenames from the internal buffer in **zgfdir** calls, and deleting the buffer at **zcldir** time.

4.6. File Management Primitives

The kernel provides a number of general file management primitives for miscellaneous operations upon files and directories. These are summarized in the table below. These primitives are all alike in that they operate upon files by name rather than by channel number. The file management primitives read, write, create, and delete the *directory entries* for files; none access the actual file data. No primitive which operates upon a file by name will be called while the file is open for i/o.

File Management Primitives

zfacss (osfn, mode, type, status)	access file
zfchdr (new_directory, status)	change directory
zfdele (osfn, status)	delete a file
zfinfo (osfn, out_struct, status)	get info on a file
zfmkcp (old_osfn, new_osfn, status)	make null copy of a file
zfpath (osfn, pathname, maxch, status)	osfn to pathname
zfprot (osfn, prot_flag, status)	file protection
zfrnam (old_osfn, new_osfn, status)	rename a file

The **zfacss** primitive is used to determine whether or not a file exists, is accessible with the given permissions, or is a text or binary file. The **zfinfo** primitive returns a data structure defining the file type, access modes, owner, size, creation date, time of last modify, and so on. Information not returned includes whether the file is a text or binary file, and whether or not the file is protected from deletion, because this information is expensive to determine on some systems. The **zfprot** primitive is called to place or remove delete protection on a file, and to test whether or not a file is protected.

Primitives for accessing directories are limited to **zfpath**, which returns the pathname of a file or of the current working directory, and **zfchdr**, which changes the current directory. There are no primitives for creating or deleting new subdirectories: thus far no program has needed such a primitive. Directory creation and manipulation is probably best left to the host system.

The assumption that there are only two basic file types, text and binary, is overly simplistic when it comes to copying an arbitrary file. If an executable file is copied as a binary file, for example, the copy will not be executable. The problem is that a conventional binary file copy operation copies only the file data: the directory entry is not copied, and information is lost. The **zfmkcp** primitive makes a zero length file which inherits all the system dependent attributes of another file, excluding the filename, length, and owner. The new file is subsequently opened for appending as either a text or binary file, and the file data is copied in the conventional manner. Directory files cannot be copied.

4.7. Process Control

Process control, interprocess communication, exception handling, and error recovery are probably the most complex and subtle services provided by the IRAF virtual operating system, and the most likely to be machine dependent. Despite the effort to isolate the machine dependence into the kernel and to make the kernel primitives as simple and self contained as possible, a high level understanding of the subtleties of process control may be necessary to debug system problems. An introduction to process control in IRAF is therefore presented to supplement the specifications for the kernel primitives. It should be possible for a systems programmer implementing the kernel to skim or skip most of this section, referring to it only to resolve ambiguities in the kernel specifications.

The CL is currently the only process in the IRAF system which spawns other processes. In this section we present an overview of process control in the CL, defining important terms, discussing the conceptual model of a subprocess as a command file, and describing the architecture of the process control subsystem. The synchronous protocol for communicating with subprocesses is discussed, as is the asynchronous file oriented protocol for communicating with background jobs. The function of the IRAF main is described, including error recovery and implementation strategies for interfacing asynchronous processes to a host system window manager or job status terminal. The kernel primitives for process control are presented at the end of the section.

4.7.1. Overview and Terminology

From the point of view of the CL, an executable program is a **command file** containing a sequence of commands to be parsed and executed. Once opened or **connected**, a compiled program is equivalent to a script task or the user terminal; the CL does not know or care where the commands are coming from. Any CL command that can be executed from the user terminal can also be executed by a compiled program or a script task. Calls to external programs, script tasks, the terminal, or any other **logical task** may be nested until the CL runs out of file descriptors or stack space.

A **program** is a compiled logical task. An arbitrary number of programs may be linked together to form a single physical **process**, i.e., executable file. When an IRAF process is executed the **IRAF Main** (main routine or driver procedure) in the subprocess initializes the process data structures and then enters an interpreter loop awaiting a command from the input file, which may be an IPC file (the CL), a terminal, or a text file. Hence, not only does a subprocess look like a file to the CL, the CL looks like a file to a subprocess.

Task termination occurs when the CL reads either end of file (EOF) or the command **bye**. When a script task terminates the script file is closed; when a program terminates the associated process may be **disconnected**. A subprocess is disconnected by sending the command **bye** to the IRAF Main in the subprocess. The IRAF main cleans up the files system and dynamic memory, calls any procedures posted by the user program with **onexit**, and then returns to its caller (the **process main**), causing process exit.

When a process spawns a subprocess, the original process is called the **parent** process, and the subprocess is called the **child** process. A parent process may have several child processes, but a child process may have only a single parent, hence the process structure of IRAF is a rooted tree. The root process is the interactive CL; the user terminal (or a window manager process) is the parent of the CL. Since the CL is the only process which spawns other processes, an IRAF process tree has only two levels.

Processes communicate with each other only via **IPC channels** or ordinary disk files. Since the only way to open an IPC channel is to connect a subprocess, a process may communicate only with its parent or with one of its children. Separate channels are used for reading and writing, rather than a single read-write channel, to conform to the logical model of a subprocess as a text file and to facilitate record queueing in the write channel. Since only IPC channels and ordinary files are used for interprocess communication and synchronization, the parent and child processes may reside on separate processors in a **multiple processor** system configuration.

The IPC channels are used to pass commands, data, and control parameters. While a program is executing it sends commands to the CL, most commonly to get or put the values of **CL parameters**. In a get parameter operation the CL responds by printing the value of the parameter on its output, just as it does when the same command is typed in interactively. The output of the CL is connected to the input IPC channel of the child process, which reads and decodes the value of the parameter, returning the binary value to the calling program.

The standard input, standard output, standard error output, standard graphics output, etc. of the child process are known as **pseudofiles** because the actual files are opened and controlled entirely by the CL. A read or write to a pseudofile in the child process is converted into a command to read or write a binary block of data and sent over the IPC channel along with the binary data block. Hence, even though most traffic on the IPC channels is ASCII text, the channels are implemented as binary files. A large transfer block size is used for all pseudofiles except STDERR to maximize throughput. All parameter i/o and pseudofile i/o is multiplexed into a single command stream and transmitted over the two IPC channels.

To minimize process connects, the CL maintains a **process cache** of connected but generally idle subprocesses. A process is connected and placed in the cache when a program in that process is run. A process will remain in the cache, i.e., remain connected to the CL process, until either a new process connect forces the process out of the cache or until the cache is flushed. Since programs execute serially, at most one cached process will be active at a time. Since several subprocesses may simultaneously be connected and each subprocess may contain an arbitrary number of programs, the cache can greatly reduce the average overhead required to run an external program, while permitting dynamic linking of programs at run time.

The CL executes programs serially much as a program executes subroutines serially. The protocol used to communicate with a connected subprocess is synchronous. To execute a program or general command block as a **background job**, i.e., asynchronously, the CL spawns a copy of itself which executes independently of the parent CL. The child CL inherits the full interior state of the parent CL, including the metacode for the command to be executed, all loaded packages and parameter files, the environment list, and the dictionary and stacks. Open files are not inherited, the child CL is not connected to the parent CL, and the CL subprocess is not placed in the process cache of the parent. The child CL manages its own process cache and executes external programs using the synchronous protocol. A child CL may spawn a child CL of its own.

4.7.2. Synchronous Subprocesses

The sequence of actions required to synchronously execute an external compiled program are summarized below. Only those actions required to execute a program are shown; the process cache is a local optimization hidden within the CL which is not relevant to a discussion of the synchronous subprogram protocol. Everything shown is machine independent except the process main and the IPC driver. Only a single process may be in control at any one time.

A process may be spawned by another IRAF process or by the host command interpreter or JCL. When a process is spawned the host system transfers control to a standard place known as the **process main**. The process main must determine what type of parent it has and open the type of input and output channels required by the parent. If the parent is another process IPC channels are opened. The process main then calls the IRAF Main which initializes IRAF i/o and enters the Main interpreter loop to read commands from the parent.

Process Startup

Parent process:
spawn the subprocess
open IPC channels between parent and child
send commands to initialize environment list in child

Process Main in child:
determine whether input device is a terminal, text file, or process
open input and output channels (e.g. the IPC channels)
call the IRAF Main

IRAF Main in child:
save process status for error restart
initialize file i/o, dynamic memory, and error handling
post default exception handlers
enter interpreter loop, reading commands from parent

An IRAF process will interpret and execute successive commands in its input file until it encounters either EOF or the command **bye**. The first block of commands read by the Main will normally be a sequence of **set** statements initializing the process environment (defining logical directories and devices, etc.). A number of calls to the programs resident in the process will normally follow. When a program runs it assumes control and begins issuing commands to the parent to read and write parameters and pseudofiles. The IRAF i/o system is reset to its default initial state each time a program terminates (this can be overridden by a program if desired).

Process Execution

Parent process:

send name of program to be run to the IRAF Main in the child process
redirect command input to child, i.e., transfer control to the
program in the child process

Program in child process:

(we were called by the interpreter in the IRAF Main)
execute, sending commands to parent to read and write parameters
and pseudofiles
return to caller (the IRAF Main) when done

IRAF Main:

flush STDOUT, close any open files, etc.
send the command **bye** to parent to signal that program has
completed and to return control to the parent
enter interpreter loop, reading commands from parent

A process shuts down or exits only when commanded to do so by the parent, i.e., when an input file read returns EOF or the command **bye** is executed. Any user defined procedures posted with **onexit** calls during process execution will be executed during process shutdown. Control eventually returns to the process main which takes whatever system dependent actions are required to terminate the process.

Process Shutdown

Parent process:

if no further programs are to be run, send the command **bye**
to the child to initiate process shutdown

IRAF Main:

disable IPC output (parent is no longer reading)
call any user procedures posted with **onexit**, i.e., flagged
to be executed upon process shutdown
return to caller (the process main) when done

Process Main:

terminate subprocess

Parent process:

disconnect child process

If a process issues a read request on an IPC channel and there is no input in the channel, the reading process will block, hence reading from an empty IPC channel causes process synchronization. Successive writes are queued until the channel is full, hence writing is generally asynchronous and some degree of overlapped execution is possible. Traffic on the IPC channels is restricted to the small set of commands described in the next section.

4.7.3. Standard IPC Commands

Although in principle a subprocess may send any legal CL command to the CL process, in practice only a small subset of commands are permitted in order to minimize the size of the interface. A larger interface would mean more dependence upon the characteristics of a particular CL, making it more difficult to modify the CL and to support several different versions of the CL.

The IPC interface commands described in this section are a high level protocol implemented entirely above the kernel routines to support execution of external programs by the CL. If the parent process were not the CL and if a new IRAF Main were implemented (the IRAF Main is an ordinary SPP procedure), then a quite different protocol could be devised.

The **IRAF Main requests**, i.e., the IPC commands sent to the IRAF Main by the parent process, are shown below in the order in which they are normally sent to the child process. Italicized text denotes dummy parameters to be replaced by the name or value of the actual parameter when the command is issued. Keywords are shown in boldface. Optional characters or arguments are delimited by square brackets. All commands are ASCII lines of text terminated by the **newline** character.

set *variable* = *string*
set *@fname*

> Set the value of an environment variable or set the environment from a file. If the variable does not exist it is created; if it does exist the new value silently replaces the old value. The **set** statement is used to pass the environment list of the parent to the child process when the subprocess is connected. The second form of the **set** statement reads a list of **set** declarations from a text file, and is especially useful in debug mode.

?

> Print the names of all user programs linked into the process in tabular form (i.e. print a menu) on the standard output. This command is not currently used by the CL; it is most useful when debugging a process run directly from the host command interpreter.

[**$**] *program* [<[*fname*]], [[*stream*[(T|B)]]>[*fname*]], [[*stream*]>>[*fname*]]

> Execute the named program. The environment should have been initialized by the time a program is run. If a dollar sign is prefixed to the command name, the cpu and clock time consumed by the process are printed on the standard error output when the task terminates. If a pseudofile stream has been redirected by the parent or is to be redirected by the child, this should be indicated on the command line. Thus the IRAF Main command
>
> count <
>
> would run the program *count*, informing the IRAF Main that the standard input has already been redirected by the parent (some programs need to know). If redirection to or from a named file is indicated, the IRAF Main will open the file and redirect the indicated stream before running the program. Pseudofiles streams are denoted by the numerals 1 through 6, corresponding to STDIN, STDOUT, STDERR, STDGRAPH, STDIMAGE, and STDPLOT. If output is being redirected into a new file and **T** or **B** appears in the argument (e.g., "4B>file"), a text or binary file will be created as specified. If the file type suffix is omitted and the output stream is STDOUT or STDERR a text file will be created, otherwise a binary file is created. For example, the command

count <, > file

directs the Main to flag the standard input as redirected, open the new text file "file" as the standard output of the the program *count*, and then run the program. When the program terminates the Main will automatically close the output file.

bye

Commands the subprocess to shutdown and exit. The subprocess must not read or write the IPC channels once this command has been received; the CL disconnects the subprocess immediately after sending **bye**. User procedures posted with **onexit** are called during process shutdown. If an irrecoverable error occurs during normal process shutdown it will cause an immediate **panic shutdown** of the process. The kernel writes an error message to the process standard error channel (e.g. the user terminal) when a panic shutdown occurs.

Although it might appear that initialization of the process environment list via a sequence of **set** commands is inefficient, the **set** commands are buffered and transmitted to the child process in large binary IPC blocks to minimize the overhead. The amount of data transmitted is not significantly different than it would be if the environment list were transmitted as a binary array, and matching of the internal environment list data structures in the two processes is not required. Furthermore, the **set** command is ideal when debugging a process or when running a process in batch mode with a previously prepared command input file.

The IRAF Main commands are used both by the CL to run an external compiled program and by the programmer when debugging a process at the host system level. The IRAF Main knows whether it is being used interactively or not, and modifies the interface protocol slightly when used interactively to provide a better user interface. For example, the Main issues a command prompt only when being used interactively. The form of a parameter request and of a pseudofile read or write request is also slightly different in the two cases.

The **CL requests**, i.e., the commands sent by a running program to the CL (or any other parent process) are shown below. These are the only commands which the child can legally send to the parent, and hence the only commands the interpreter in the parent need recognize. The noninteractive syntax is shown. If the parent process is not the CL a completely different protocol can be used. When a subprocess is run interactively the **xmit** and **xfer** requests are omitted (only the data is sent) and the newline is omitted after a parameter read request.

param =

The parent process is directed to print the single-line value of the named parameter in ASCII on the child's input IPC channel. The child decodes the response line and returns a binary value to the program.

param = *value*

The parent process is directed to set the value of the named parameter to the indicated ASCII value. The child does not expect a response, and parameter write requests may be queued in the output IPC channel.

xmit (*pseudofile*, *nchars*)

The parent process is directed to read exactly *nchars* chars of binary data from the IPC channel and transmit it without interpretation to the indicated pseudofile. The child does not expect a response, and pseudofile write requests may be queued in the output IPC channel. Pseudofiles are denoted by the numerals 1 through 6, corresponding to STDIN, STDOUT, STDERR, STDGRAPH, STDIMAGE, and STDPLOT.

xfer (*pseudofile*, *maxchars*)

The parent process is directed to read up to *maxchars* chars of binary data from the indicated pseudofile and transmit it without interpretation to the input IPC channel of the child. The binary data block should be preceded by an ASCII integer count of the actual number of chars in the data block.

bye

Normal program termination. Control is transferred from the child to the parent. The child returns to the interpreter loop in the IRAF Main, awaiting the next command from the parent.

error (*errnum*, "*errmsg*")

Abnormal program termination. An irrecoverable error has occurred during the execution of the program (or of the IRAF Main), and the CL is directed to take an error action for error number *errnum*. The child returns to the interpreter loop in the IRAF Main, awaiting the next command from the parent. If the error is not caught and handled by an error handler in a CL script, the error message *errmsg* is printed on the standard error output of the CL and the child process is commanded to shutdown.

4.7.4. Example

By this point process control probably sounds much more complicated than it actually is. A brief example should illustrate the simplicity of the CL/IPC interface. Consider the CL command

cl> **set | match tty**

which prints the values of all environment entries containing the substring **tty**. The CL task **set** is a builtin function of the CL and hence does not use the IPC interface. We assume that the process **system$x_system.e**, which contains the program *match*, has already been connected so that it is not necessary to pass the environment to the child. The traffic over the IPC channels is shown below. If a running IRAF system is available the process side of this example can be duplicated by typing **echo=yes** followed by the command shown above.

CL	*Process*
match <	
	pattern=
tty	
	metacharacters=
yes	
	stop=
no	
	xfer(1,1024)
1024 *(1024 chars of data sent to child)*	
	xfer(1,1024)
368 *(368 chars of data sent to child)*	
	xmit(2,63) *(63 chars of data sent to CL)* **bye**

Each line of text shown in the example is transmitted through the appropriate IPC channel as a single distinct record. Commands are shown in boldface. The italicized records represent raw data blocks. The process **system$x_system.e** contains the fifty or so executable programs in the **system** package and hence is a good example of the use of multitasking and the process cache to minimize process connects (as well as disk space for executable images).

4.7.5. Background Jobs

IRAF process control does not support fully asynchronous subprocess execution for the following reasons:

- The parent and child processes are tightly bound, i.e., while an external program is executing the CL process is subservient to the applications program. The fact that the CL is a separate process is an irrelevant detail to the applications program. From the point of view of the applications program the CL is a database interface called by CLIO. Applications programs are not fully functional unless connected to a CL at run time, and a synchronous, interactive interface between the two processes is assumed.
- From the point of view of the user or of a CL script, external programs are subroutines. Subroutines execute serially in the context of the calling program. In this case the context is defined by the state of the data structures of the CL, i.e., the dictionary, environment list, loaded packages and tasks, parameters, and so on.
- The user does not care whether a task is a subprocess or a CL script, and tends to think in terms of **command blocks** rather than individual commands. A command block is a user specified sequence of commands to be compiled and executed as a single unit. Asynchronous subprocesses are not interesting; what we want is an asynchronous command block.
- It is much more difficult to define a machine independent process control interface for asynchronous subprocesses than for synchronous subprocesses. The problem is similar to that of designing a multiprocessing operating system, with the CL acting as the operating system kernel and the user as the cpu. Asynchronous subprocess execution is inconsistent with the conceptual model of subprocesses and users as command files, and is extremely difficult to implement in a portable system in any case.

For these and other reasons, background job execution is implemented in IRAF by spawning a copy of the foreground CL which executes as a **detached process**, rather than as a connected subprocess. The child CL manages its own process cache independently of the parent. All connected subprocesses execute synchronously, i.e., only one process in a tree of connected processes may be active at a time. Since the child is never connected to the parent, the background CL may execute at any time (e.g. in a batch queue), and may continue to execute after the parent process has terminated (if the host system permits).

The child CL inherits the data structures of the parent as they existed immediately after translating the command block into metacode and just prior to execution. The parent's data structures are propagated to the child by writing them into a binary file which is subsequently opened and read by the child. Open files are not inherited. The command block executes in the child in exactly the same context as it would have had if executed in the parent, with exactly the same results.

On many systems background job execution will be predominantly noninteractive, particularly if background jobs are placed into a batch queue. Even if a background job is run noninteractively, however, there is no guarantee that the job will not require interaction during execution, for example if the user forgot to set the value of a parameter when the job was submitted. Rather than aborting a background job which needs to query for a parameter, the CL provides a limited but portable method for servicing queries from background jobs. Extensive interaction with background jobs is beyond the capabilities of the portable IRAF system but is not ruled out; interfacing to **window management** facilities is straightforward if the host system provides such facilities, and is described in the next section.

The CL has a builtin capability for generating and servicing queries from noninteractive background jobs. Such queries might be normal and expected, e.g. if a background job executes concurrently with the interactive CL and limited interaction is desired, or might be a failsafe, e.g. if a background job has consumed several hours of cpu time and the job would have to be resubmitted if it were to abort because it could not satisfy a parameter request.

The CL will automatically initiate a query request sequence whenever it is executing in the background and an attempt to service a query by reading from the process standard input returns EOF. To initiate a query request the CL writes the query prompt into a **service request file**, writes a status message to the standard error output of the CL process noting that the job is stopped waiting for parameter input, and enters a loop waiting for the **query response file** to be created. When the query response file becomes accessible the CL opens it, reads the contents to satisfy the original read request, deletes the file, and continues normal execution.

If there is no response the background CL will eventually timeout, writing a fatal error message to the standard error output of the process. Queries from background jobs are normally satisfied from an interactive CL using the builtin task **service**, which types the service request file on the terminal, deletes the file, reads the user's response from the terminal, and writes the response into the query response file. If the query response is unacceptable another query will be generated and the interchange is repeated. The use of simple text files for interprocess communication makes the technique very general, and in principle there is nothing to prevent the technique from being used to service requests from jobs run either as detached subprocesses or in batch queues.

4.7.6. The Process and IRAF Mains

The roles of the Process and IRAF Mains should already be clear from the previous sections. The process main is machine dependent and is called by the host operating system when a process is executed. The IRAF Main is a portable SPP procedure which is called by the process main during process startup, which acts as a simple command interpreter during process execution, and which returns control to the process main during process shutdown.

4.7.6.1. The Process Main

The **process main** is a part of the kernel, but unlike any other kernel procedure it is not Fortran callable (in fact it is not necessarily a procedure at all). The process main further differs from any other kernel procedure in that it calls a high level procedure, the IRAF Main. Since the process main is not Fortran callable, however, there is no possibility of recursion.

The primary functions of the process main are to open and initialize the process i/o channels and to call the IRAF Main. The process i/o channels are the standard input, standard output, and standard error output of the process. The process or device to which the channels are connected is both system dependent and dependent on how the process was spawned.

The process main can be coded in assembler if necessary on almost any system. Typically the host operating system will upon process entry transfer control to a predefined address or external identifier, and this entry point should be the process main. On many modern systems it will be possible to code the main in a high level language; this is desirable provided the high level language does not have a main of its own which necessitates loading a lot of extraneous code which will never be used (since IRAF does all i/o via the IRAF kernel). On a UNIX system, for example, the process main is implemented as the C procedure "main" with no overhead, so there is nothing to be gained by coding the process main in assembler.

procedure process_main

```
input_chan:     process standard input channel
output_chan:    process standard output channel
errout_chan:    process standard error output channel

begin
        # Determine type of output device and connect channels.

        if (we are a connected subprocess) {
            connect input_chan and output_chan to IPC channels
            connect errout_chan to the user terminal (i.e, to the
                standard error output channel of the parent process)

        } else if (we are a detached process) {
            if (window management facilities are available)
                connect all channels to window manager
            else {
                connect input_chan such that a read will return EOF
                if (we are executing in a batch queue)
                    connect output channels to files
                else {
                    connect both output_chan and errout_chan to the user
                        terminal (i.e, to the standard error output channel
                        of the parent process, if the terminal can be
                        written to by multiple processes)
                }
            }

        } else if (we were run from the host command interpreter) {
            if (we were called interactively)
                connect channels to user terminal
            else {
                connect input_chan and output_chan to job input
                    and output files, and errout_chan to operator
                    console or system dayfile.
            }
        }

        # Call the IRAF Main, the command interpreter or driver of an
        # IRAF process.

        call iraf_main, passing the channel numbers and identifying
            the driver and protocol to be used by the Main

        # We get here only after the parent has commanded the IRAF
        # Main to shutdown, and after shutdown has successfully
        # completed.  Fatal termination occurs elsewhere.

        close channels if necessary
        normal exit, i.e., terminate process
end
```

An IRAF process may easily be used in a completely batch mode by connecting the process channels to text files. On an interactive system the channels of a detached process may be connected directly to the user terminal, but it can be annoying for the user if background processes are intermittently writing to the terminal while the user is trying to do something else (e.g. trying to edit a file using a screen editor). Having multiple processes trying to simultaneously read from a terminal is disastrous.

The best solution to the problem of multiple processes trying to read or write from the user terminal is some sort of **window manager**. A simple window manager which can handle output from multiple simultaneous IRAF processes but which will only allow a single process to read is not difficult to code on many systems, provided one admits that the capability is machine dependent. A full up window manager such as is provided on many modern microcomputers is a much more difficult problem and should not be attempted as an add-on, as it really needs to be integrated into the operating system to work well. A better approach is to buy a microcomputer which comes with a bit-mapped terminal and a fully integrated window manager, or to buy a smart terminal which has window management capabilities.

If a window manager is to be provided as an add-on to a system which does not already have one, it should be implemented as a single process handling all i/o to the terminal. The CL will be run from the window manager process and detached processes will talk directly to the window manager process using multiplexed IPC channels. Such an add-on window manager is unlikely to be fully usable for non-IRAF processes (e.g. the host system screen editor) unless the host operating system is modified in fundamental ways. If the window manager has to be built from scratch consider coding it as an IRAF process with an extended system interface, so that it will be at least partially portable.

4.7.6.2. The IRAF Main

The IRAF Main is the "main program" of an IRAF process. The primary function of the Main is to interpret and execute commands from the standard input of the CL process (the stream CLIN) until either EOF is seen or the command **bye** is received by the Main (as opposed to a program called by the Main). The Main is mechanically generated by the SPP compiler when the **task** statement is encountered in an SPP program; the source is in the file **main.x** in the logical directory **sys$system**.

The secondary functions of the Main are to initialize the IRAF i/o system and participate in error recovery. The first time the Main is called it initializes the i/o system and posts a default set of **exception handlers**. The Main can only be called again (without recursion) during an **error restart**. If an irrecoverable error occurs during error restart, a panic exit occurs, i.e., the process dies.

Error restart takes place when a uncaught hardware or software exception occurs, or when an error action is taken by a program and no user **error handler** is posted. All exceptions and errors may ideally be caught and processed by a user exception handler or error handler, without error restart occurring. When error restart occurs the hardware stack is reset and control transfers to the marked position within the process main. The process main calls the IRAF Main, which knows that it has been called during error restart.

When the IRAF Main is called during error restart the first thing it does is call any user procedures posted with **onerror**. If an irrecoverable error occurs during execution of an **onerror** error recovery procedure, **error recursion** occurs and a panic exit results. When the **onerror** procedures have successfully executed the Main sends the **error** statement to the CL (i.e., to the stream CLOUT) and reenters its interpreter loop, awaiting the next command from the CL. If no user error handler is posted at the CL level (error handling was not implemented at the CL level at the time when this was written), then the CL will direct the child process to shutdown to ensure that dynamic memory space is reclaimed, and to ensure that a user program is not left in a bad state by the error.

4.7.7. Process Control Primitives

We are now in a position to define and understand the kernel primitives necessary to implement process control. There are 9 such primitives, excluding the process main and the exception handling primitives. The mnemonic "pid" refers to the **process id**, a unique magic integer assigned by the host operating system at process creation time.

Process Control Primitives

zopcpr (process, inchan, outchan, pid)	open connected subprocess
zclcpr (pid, exit_status)	close connected subprocess
zintpr (pid, exception, status)	interrupt connected subprocess
zopdpr (process, bkgfile, jobnum)	open or queue detached process
zcldpr (jobnum, killflag, exit_status)	close or dequeue detached process
zgtpid (pid)	get process id of current process
zpanic (errcode, errmsg)	panic exit
zsvjmp (jumpbuf, status)	save process status
zdojmp (jumpbuf, status)	restore process status

Separate sets of primitives are defined for connected and detached subprocesses. A subprocess is connected with **zopcpr**, which spawns the subprocess and opens the IPC channels. The child is assumed to inherit the current working directory of the parent. Connection of the IPC channels to FIO and transmission of the environment list to the subprocess is left to the high level code.

A connected subprocess is disconnected with **zclcpr**, which waits (indefinitely) for the subprocess to exit and returns the exit status code, e.g. OK. The high level code must command the subprocess to shutdown before calling **zclcpr** or deadlock will occur. The high level code guarantees that **zclcpr** will be called to close any subprocess opened with **zopcpr**. The **zintpr** primitive raises the interrupt exception X_INT in a connected subprocess.

A detached process is spawned or submitted to a batch queue with **zopdpr**. It is up to **zopdpr** to pass the name of the background file on to the child by some means (the background file tells the detached process what to do). A detached process may be killed or removed from the batch queue by a call to **zcldpr**. The high level code will call **zcldpr** if the detached process terminates while the parent is still executing, but there is no guarantee that a detached process will be closed.

The remaining primitives are used by all processes. The **zgtpid** primitive returns the process id of the process which calls it; this is useful for constructing unique temporary file names. The **zpanic** primitive is called when error recovery fails, i.e., when an error occurs during error recovery, causing error recursion. A panic shutdown causes immediate process termination, posting an error message to the process standard error output and returning an integer error code to the parent process.

The IRAF main calls **zsvjmp** to save the process control status for error restart. The process status is saved in *jumpbuf*, allowing several jump points to be simultaneously defined. A subsequent call to **zdojmp** restores the process status, causing a return from the matching **zsvjmp** call *in the context of the procedure which originally called* **zsvjmp**. The *status* argument is input to **zdojmp** and output by **zsvjmp**, and is zero on the first call to **zsvjmp**, making it possible for the procedure which calls **zsvjmp** to determine how it was entered. These extremely machine dependent routines are patterned after the UNIX **setjmp** and **longjmp** primitives, but are Fortran callable. They will almost certainly have to be written in assembler since they fiddle with the hardware stack and registers.

On a **multiple processor** system it should be possible to spawn both connected and detached processes on a remote processor. For example, if the parent process resides on a diskless node in a cluster, it may be desirable to run subprocesses that do heavy i/o on the remote file server processor which has a high i/o bandwidth to disk. On such a system advice on the optimal processor type should be encoded as extra information in the process file name passed to **zopcpr** or **zopdpr**; this will require modification of the CL **task** statement for such processes.

4.8. Exception Handling

An exception is an asynchronous event, i.e., an interrupt. Typical **hardware exceptions** are an attempt to access an unmapped region of memory, illegal instruction, integer overflow, or divide by zero. A hardware exception occurs when the hardware detects an error condition while executing a hardware instruction. Typical **software exceptions** are interrupt and kill. A software exception occurs when a program, e.g., the terminal driver or an applications program like the CL, sends an interrupt or **signal** to a process. When an exception occurs program execution is interrupted and control transfers to an **exception handler**, i.e., to a previously posted system or user procedure.

The ability to post an exception handler or to send a signal to a process is fundamental in any multiprocessing operating system, but regrettably there are still some older systems that do not make such facilities available to applications code. Hopefully yours is not such a system. Even if a system provides exception handling facilities, the set of exceptions defined for a particular computer or operating system is very system dependent. Exception handling can be subtly machine dependent, e.g., it is not always possible to disable an exception, and it is not always possible to resume program execution (restart the interrupted instruction) following an exception.

The IRAF Main posts a default set of exception handlers during process startup. All host system exceptions that might occur during the execution of an IRAF process should be catchable by one of the default exception handlers. If an exception is not caught and is instead handled by the host, the process will almost certainly die without the knowledge of the CL, leading at best to a cryptic "write to a subprocess with no reader" error message, and at worst to deadlock. Since error recovery and process shutdown will be skipped if an uncatchable exception occurs, disk data structures may be corrupted.

The virtual system recognizes only four classes of exceptions; all possible host system exceptions should either be mapped into one of these exceptions or caught in the kernel and mapped into ERR.

Virtual Machine Exceptions

exception	*code*	*meaning*
X_ACV	501	access violation
X_ARITH	502	arithmetic error
X_INT	503	interrupt
X_IPC	504	write to IPC with no reader

The largest class of exceptions on many systems will be the access violations. This class includes such things as illegal memory reference, illegal instruction, illegal system call, and so on. Arithmetic exceptions include divide by zero, integer overflow, and the like. Interrupt is the exception raised by **zintpr** or by the host system terminal driver when the interrupt sequence is typed (e.g. ctrl/c).

Exception Handling Primitives

zxwhen (exception, handler, old_handler)	post an exception
zxgmes (os_exception, outstr, maxch)	get OS code and message

An exception handler is posted for a virtual exception with the primitive **zxwhen**. All host exceptions in the indicated virtual exception class are affected. The argument *handler* is either the entry point address of the new user exception handler or the magic value X_IGNORE (null). The address of the old handler or X_IGNORE is returned as the third argument, making it possible for the high level code to chain exception handlers or repost old exception handlers. The calling sequence for a user exception handler is as follows:

user_handler (exception, next_handler)

The user exception handler is called with the integer code for the actual virtual exception as the first argument. The integer code of the last machine exception and a packed character string describing the exception may be obtained by a subsequent call to **zxgmes**. A program which uses a machine exception code is machine dependent, but machine exception codes can be parameterized and some programs need to know. The CL, for example, has to be able to recognize the machine exception for a write to a process (an IPC channel) with no reader.

When an exception occurs control actually transfers to the **kernel exception handler**, which maps the machine exception into a virtual exception, looks at the kernel exception table to determine what type of action is required, and then calls the user exception handlers. A user exception handler is expected to handle the exception in some application dependent way and then either change the process context by calling **zdojmp** or **zrestt**, or return control to the kernel exception handler. If control returns to the kernel handler the output argument **next_handler** will contain either the entry point address of the next exception handler or X_IGNORE. The high level code assumes that once an exception handler is posted it stays posted, i.e., is not reset when an exception occurs.

Few programs actually post exception handlers; most just post an error handler with **onerror**. Such an error handler will be called by the IRAF Main either when an exception occurs or when an error action is taken, i.e., when a program is aborted for whatever reason. If a user exception handler is not posted the default handler will be called, causing error restart of the Main, calls to all **onerror** procedures, and transmission of the **error** statement to the CL. If the CL is interrupted while executing an external program it passes the interrupt on to the child with **zintpr** and then resumes normal processing. The external program retains control, and therefore can choose to either ignore the interrupt or take some application dependent action.

4.9. Memory Management

The IRAF system relies heavily on memory management for dynamic buffer allocation in both system and applications software. Both stack and heap storage are provided at the program interface level. The **stack** is used primarily for "automatic" storage allocation, i.e., for buffers which are allocated upon entry to a procedure and deallocated upon exit from the procedure. Stack management incurs very little overhead for small buffers. The **heap** is a more general storage mechanism; buffers may be allocated and deallocated in any order, and allocation and deallocation may occur in different procedures. A heap buffer may be reallocated, i.e., changed in size. The stack is implemented portably in terms of the heap, and hence need not concern us further here.

Memory Management Primitives

zmaloc (buffer, nbytes, status)	allocate a buffer
zmfree (buffer, status)	deallocate a buffer
zraloc (buffer, nbytes, status)	reallocate a buffer
zlocva (variable, address)	get address of a variable
zawset (bestsize, newsize, oldsize, textsize)	adjust working set size

Buffer space is allocated on the heap by the primitive **zmaloc**. The address of a buffer at least *nbytes* in size is returned as the argument *buffer*. Nothing is assumed about the alignment of the buffer. The contents of the buffer are not assumed to be initialized.

The buffer address returned by **zmaloc** is in units of SPP **chars** rather than in physical units. The **zlocva** primitive returns the address of a **csilrd** variable, array, or array element in the same units. By using char address units and by doing all pointer dereferencing by subscripting Fortran arrays, we avoid building knowledge of the memory addressing characteristics of the host system into SPP programs. The zero point of a char address is undefined; negative addresses are possible depending on the implementation. It must be possible to store an address in an integer variable, and it must be possible to perform *signed integer* comparisons and arithmetic on addresses.

A buffer allocated with **zmaloc** may be reallocated with **zraloc**. In this case the *buffer* argument is used both for input and for output. If the input value is NULL a new buffer should be allocated, otherwise the size of the buffer should be either increased or decreased depending on the value of *nbytes*. The buffer may be moved if necessary, provided the contents of the buffer are preserved. This primitive may be implemented as a call to **zmaloc** followed by an array copy and a call to **zmfree** if desired, saving one kernel primitive, with significant loss of efficiency in some applications.

A buffer allocated with **zmaloc** is deallocated with **zmfree**. Deallocation need not involve physically returning memory pages to the operating system; if the buffer is small this will not be possible. The buffer being deallocated need not be at the end of the process address space.

The **zlocva** primitive returns the address in char units of the first argument as the integer value of the second argument. Since Fortran is call by reference, this is a simple matter of copying the pointer to the first argument (as opposed to the value pointed to) to the integer location pointed to by the second argument. Only arguments of datatypes **csilrd** are permitted in calls to **zlocva**. Arguments of Fortran type COMPLEX, CHARACTER, and EXTERNAL are sometimes passed using more than one physical argument (depending on the host compiler) and hence cannot be used in procedures that operate upon an arbitrary datatype.

The **zawset** primitive is used both to determine and to change the amount of physical memory in machine bytes available to a process, i.e., the **working set size** on a virtual memory machine. If called with a *bestsize* of zero the current working set size and text segment size is returned. If called with nonzero *bestsize* on a virtual memory machine the working set size will be adjusted up or down as indicated, returning the actual new working set size in *newsize* and the old working set size in *oldsize*. It is not an error if the amount of memory requested cannot be allocated; the high level code will ask for what it wants but take what it gets. High level routines which need lots of memory rely on this primitive to avoid running out of memory on nonvirtual machines and to avoid thrashing on virtual machines.

The high level code (**malloc**) converts the address returned by the kernel primitives into an integer valued offset (array index) into the **Mem** common. The dynamically allocated buffer (which has nothing to do with the Mem common) is referenced by indexing off the end of an **Mem** array. The portable MEMIO code ensures alignment between the physical buffer and the "pointer" returned to the user (index into Mem). This technique should work on any machine which permits referencing off the end of an array.

There is one place in the system code, however, which does something trickier and which should be checked on a new system. The **stropen** routine in FIO uses the same pointer technique (for efficiency reasons) to reference a *statically* allocated **char** array in the user program. If the compiler does not guarantee that a statically allocated **char** array will be aligned with the array **Memc** in the Mem common this will not work, and **stropen** will have to be modified.

4.10. Procedure Call by Reference

Fortran allows an external procedure to be passed by reference to a subprogram via the subprogram argument list. An external procedure passed as an argument to a subprogram may be called by the subprogram but may not be saved and called at some later time. IRAF (e.g. FIO and IMIO) requires the capability to save a reference to a procedure in an integer variable for execution at some later time.

The **zlocpr** primitive is used to determine the entry point address (EPA) of an external procedure. IRAF assumes that the EPA of a procedure may be stored in an integer variable and that two procedures with the same EPA are identically the same procedure. No other operations are permitted on EPA values, e.g., signed comparisons and arithmetic are not permitted.

Call by Reference Primitives

```
zlocpr (proc, entry_point_address)
 zcall1 (procedure, arg1)
 zcall2 (procedure, arg1, arg2)
 zcall3 (procedure, arg1, arg2, arg3)
 zcall4 (procedure, arg1, arg2, arg3, arg4)
 zcall5 (procedure, arg1, arg2, arg3, arg4, arg5)
```

A **zcall***n* primitive is used to call an external subroutine referenced by the integer variable *procedure*, the entry point address of the procedure returned in a prior call to **zlocpr**. Only subroutines may be called by reference; there is no comparable facility for functions. The datatypes of the arguments are unspecified but are restricted to the SPP datatypes **csilrd**.

4.11. Date and Time

Kernel primitives are required to read the system clock, to determine the amount of cpu time consumed by a process (for performance measurements), and to generate time delays.

Date and Time Primitives

zgtime (local_time, cpu_time)	get clock time and cpu time
ztslee (delay)	countdown timer

The **zgtime** primitive returns two long integer arguments. The local standard time in integer seconds since midnight on January 1, 1980 is returned as the first argument (the "clock" time). The second argument is the total cpu time consumed by the process since process execution, in units of milliseconds. The countdown timer primitive **ztslee** suspends execution of the calling process for the specified number of integer seconds. There is currently no provision for generating delays of less than one second.

4.12. Sending a Command to the Host OS

The ability to send an explicitly machine dependent command to the host system command interpreter is required by the CL and by some of the system utilities. Any program which uses this command is bypassing the system interface and is system dependent. Nonetheless it is very useful for the *user* to be able to send a command to the host without leaving the IRAF environment, and certain of the system utilities are much easier to code given the capability (e.g., **diskspace** and **spy**). These utilities help provide a consistent user interface on all systems, and in many cases such a utility program can be built in a few minutes for a new system. No science program or essential system utility bypasses the system interface in this fashion.

Host OS Command Primitive

zoscmd (cmd, stdout, stderr, status)   send a command to the host OS

The command *cmd* may be any packed string acceptable to the host system. The call does not return until the command has been executed. The status OK or ERR is returned indicating success or failure. If either of the filename strings *stdout* or *stderr* is nonnull the associated output stream of the command will be directed (if possible) to the named text file.

5. Bit and Byte Primitives

The bit and byte primitives are not considered true kernel procedures since they are purely numerical and are only potentially machine dependent. These primitives are more properly part of the **program interface** than the kernel, since they are callable from ordinary applications programs. Both SPP or Fortran and C versions of most of the routines are supplied with the system which will port to most modern minicomputers and some large computers. The source directory is **sys$osb**. The following classes of routines are required:

- bitwise boolean operations (and, or, etc.)
- bitfield insertion and extraction
- byte primitives (copy, swap, string pack/unpack)
- type conversion for byte, unsigned short datatypes
- machine independent integer format conversions

The IRAF system uses 8 standard datatypes in compiled SPP programs, as shown in the table below. Variables and arrays may be declared and accessed conventionally using any of these datatypes. Data may additionally be stored on disk, in images, and in memory in packed char or integer arrays in the exotic datatypes **unsigned byte**, **unsigned short**, and **packed string**.

Standard SPP Datatypes

name	*suffix*	*Fortran equivalent*
bool	b	LOGICAL
char	c	nonstandard
short	s	nonstandard
int	i	INTEGER
long	l	nonstandard
real	r	REAL
double	d	DOUBLE PRECISION
complex	x	COMPLEX

The char and short datatypes are commonly implemented as INTEGER*2, and long as INTEGER*4, but all could be implemented as the standard INTEGER if necessary. To save space char may be implemented using a signed byte datatype if the host Fortran compiler provides one, provided the special datatype chosen may be equivalenced with the standard datatypes (the minimum precision of a char is 8 bits signed). IRAF assumes that the 7 types **csilrdx** may be equivalenced in common (e.g. in **Mem**). The standard type suffixes **bcsilrdx** are commonly appended to procedure names to identify the datatype or datatypes upon which the procedure operates.

5.1. Bitwise Boolean Primitives

The bitwise boolean primitives are used to set and clear bits or bitfields in integer variables. The practice is portable provided the minimum precision of an integer variable (16 bits) is not exceeded. Primitives are provided for the 3 integer datatypes, i.e., short, int, and long, denoted by the suffix **[sl]** in the table below. In other words, the notation **and[sl]** refers to the procedures **ands** and **andl**. These quasi-primitives, unlike the true kernel primitives, are user callable and are implemented as *functions*.

Bitwise Boolean Primitives

and,and[sl] (a, b)	int = and (a, b)
or,or[sl] (a, b)	int = or (a, b)
xor,xor[sl] (a, b)	int = xor (a, b)
not,not[sl] (a, b)	int = not (a, b)

Bitwise boolean primitives are provided in many Fortran compilers as integer intrinsic functions. If this is the case it suffices (and is more efficient) to place a **define** statement in **iraf.h** to map the IRAF name for the function to that recognized by the host Fortran compiler. For example,

define and iand

would cause all occurrences of the identifier **and** in SPP programs to be replaced by **iand** in the Fortran output, which the host compiler would hopefully compile using inline code.

5.2. Bitfield Primitives

A **bitfield** is an unsigned integer segment of a bit array, where the number of bits in the segment must be less than or equal to NBITS_INT, the number of bits in an integer. A **bit array** is a sequence of bits stored one bit per bit in a char or integer array. The essential thing about a bit array is that byte and word boundaries are irrelevant, i.e., a bitfield may straddle a word boundary.

Bitfield Primitives

bitpak (intval, bit_array, bit_offset, nbits)	integer → bitfield
int = bitupk (bit_array, bit_offset, nbits)	bitfield → integer

Bit offsets range from 1, not 0, to MAX_INT. A bitfield is zero-extended when unpacked by **bitupk**, and unset bits are zeroed when an integer is packed into a bitfield by **bitpak**. If the integer is too large to fit in the bitfield it is truncated. These primitives should be implemented in assembler on a machine like the VAX which has bitfield instructions.

5.3. Byte Primitives

The byte primitives are difficult to use portably in high level code without building knowledge of the sizes of the SPP datatypes in bytes into the code. Fortunately the byte primitives are rarely used; the most common usage is in programs used to transport data between machines (e.g., a magtape reader program). A number of machine constants are defined in **iraf.h** to allow parameterization of programs which operate on data in units of bytes.

Machine Parameters for Byte Data	
name	*definition*
SZB_CHAR	machine bytes per char
NBITS_BYTE	nbits in a machine byte
SZ_*type*	size of datatype *type* (upper case) in chars

On most machines the byte primitives can be written in Fortran by representing a byte array as an array of CHARACTER*1. This suffices for programs which merely move bytes around, but not for programs which do numerical comparisons and arithmetic operations upon character data using CHAR and ICHAR, because the collating sequence for CHARACTER data in Fortran is not necessarily ASCII.

Nonetheless CHAR and ICHAR can be used on most machines to operate upon bytes, i.e., upon non-CHARACTER data stored in CHARACTER*1 arrays. Of course we are asking for trouble using CHARACTER for non-CHARACTER operations, so routines which do so are potentially machine dependent. Both Fortran and C versions of most of the byte primitives are supplied. The **bytmov** primitive should be written in assembler on a machine such as the VAX which can perform the operation in a single instruction (it is even more important to perform this optimization for the **amov** vector operators, which are more widely used).

Byte Primitives

bytmov (a, aoff, b, boff, nbytes)	move an array of bytes
bswap2 (a, b, nbytes)	swap every pair of bytes
bswap4 (a, b, nbytes)	swap every 4 bytes
chrpak (a, aoff, b, boff, nchars)	pack chars into bytes
chrupk (a, aoff, b, boff, nchars)	unpack bytes into chars
strpak (a, b, maxchars)	SPP string → byte-packed string
strupk (a, b, maxchars)	byte-packed string → SPP string

The **bytmov** primitive moves a portion of a byte array into a portion of another byte array. The move is nondestructive, i.e., if the input and output arrays overlap data must not be destroyed. The **zlocva** primitive may be used to determine if the arrays will overlap. Byte swapping is performed by the **bswap2** and **bswap4** primitives, which swap every 2 bytes or every 4 bytes, respectively, regardless of the number of bytes per short or long integer on the host machine. These routines are used primarily to swap bytes in interchange data before is it unpacked into host integers (or after packing into interchange format), hence the primitives are defined independently of the host word size. A 2 byte swap interchanges successive pairs of bytes; a 4 byte swap of two 4 byte integers rearranges the bytes as 12345678 → 43218765.

The **chrpak** and **chrupk** primitives pack and unpack SPP chars into bytes, performing sign extension in the unpacking operation. The mapping is nondestructive, i.e., the input and output arrays may be the same, and the numeric value of a character is not changed by the mapping (the collating sequence is not changed by the mapping). If SZB_CHAR is 1, **chrpak** and **chrupk** are equivalent, and if the input and output arrays are the same or do not overlap they are equivalent to **bytmov**.

The **strpak** and **strupk** primitives pack and unpack SPP strings into packed strings. A packed string is a sequence of zero or more characters, packed one character per byte, delimited by end-of-string (EOS). While SPP strings are always ASCII the collating sequence of a packed string is whatever is used for character data by the host machine. The mapping is nondestructive in the sense that the input and output arrays may be the same. Since the collating sequence may be changed in the mapping and the mapping need not be one-to-one, information may be lost if an arbitrary string is packed and later unpacked.

A packed string is not the same as a Fortran CHARACTER variable or constant. Many

Fortran compilers use two physical arguments to pass a Fortran CHARACTER argument to a subprocedure, while a packed string is always passed by reference like an ordinary integer array. There is no machine independent way to fake a Fortran string in an argument list. Furthermore, packed strings are heavily used in the kernel for machine dependent filenames, and these file names typically contain characters not permitted by the restrictive Fortran standard character set. The packed string format is equivalent to that expected by the C language.

5.4. Vector Primitives

Nearly all of the operators in the vector operators package (VOPS, **sys$vops**) are machine independent. The exceptions are the **acht** primitives used to change the datatype of a vector to or from one of the special datatypes **unsigned byte** and **unsigned short**. An **acht** operator is provided for every possible type conversion in the set of datatypes **csilrdx** plus unsigned byte (**B**) and unsigned short (**U**), for a total of 9 squared or 81 operators in all. The **bool** datatype is not supported by VOPS.

Two type suffixes are used to specify the type conversion performed by an operator; for example, **achtir** will convert an integer array into a real array. In the table below the underscore stands for the set of datatypes **UBcsilrdx**, hence each the operators shown is actually a generic operator consisting of 9 type specific operators. Both C and Fortran sources are provided for all primitives, the C sources being more efficient. The Fortran operators will work on many hosts but are potentially machine dependent and should be checked. The C versions are more efficient since Fortran does not support the unsigned datatypes and a masking operation must be performed to undo sign extension when converting from unsigned to signed.

Machine Dependent Vector Primitives

acht_b (a, b, npix)	SPP datatype → unsigned byte
acht_u (a, b, npix)	SPP datatype → unsigned short
achtb_ (a, b, npix)	unsigned byte → SPP datatype
achtu_ (a, b, npix)	unsigned short → SPP datatype

Many of the conversions do not preserve precision, i.e., double to real or integer to unsigned byte. The imaginary part of a complex number is discarded when converting to some other datatype, and the imaginary part is set to zero when converting a non-complex datatype to complex. All type conversion operators allow the conversion to be performed in place, i.e., the input and output arrays may be the same.

5.5. MII Format Conversions

The Machine Independent Integer format (MII) is used to transport binary integer data between computers. The format is independent of the transmission medium, and hence might be used to transport data via magnetic tape, over a local area network, or over a modem. The MII integer format is equivalent to that defined by the FITS image interchange format. The MII primitives are used in the IRAF FITS reader and writer programs and will probably be used in the GKS (Graphical Kernel System) software to implement a machine independent VDM (Virtual Device Metafile).

MII defines 3 integer datatypes, 8 bit unsigned integer, 16 bit twos-complement signed integer, and 32 bit twos-complement signed integer. An integer array in MII format may be thought of as a stream of 8-bit bytes. In each 2 and 4 byte integer successive bytes are written in order of decreasing significance. The sign bit is in the first byte of each 2 or 4 byte integer. For example, two 16 bit integers would be represented in MII format as the following sequence of 4 bytes.

byte	*significance*

1	high byte of first integer, including sign bit
2	low byte of first integer
3	high byte of second integer, including sign bit
4	low byte of second integer

The order of the bits within a byte is (must be) standardized at the hardware level, else we would not be able to transmit character data via cardimage tapes and modems. Hence the sign bit is bit 200B (octal) of the first byte of an MII integer, the most significant bit is bit 100B of the first byte, and so on.

MII Primitives

miipak (spp, mii, nelems, spp_type, mii_type)	SPP → MII
miiupk (mii, spp, nelems, mii_type, spp_type)	MII → SPP
intlen = miilen (n_mii_elements, mii_type)	get size of array

SPP or Fortran integer data is converted to MII format with **miipak**, and MII data is converted to SPP format with **miiupk**. The argument *spp* refers to an SPP integer array of datatype *spp_type*, and *mii* refers to an MII byte stream of MII datatype *mii_type*. The legal integer values of *mii_type* are 8, 16, and 32. MII data is stored in an SPP array of type **int**. The length of the SPP integer array required to store *n_mii_elements* of MII type *mii_type* is returned by the primitive **miilen**.

An SPP implementation of the MII primitives which should work for all host machines with 16 or 32 bit twos-complement signed integer datatypes is supplied with the system. Most modern minicomputers fall into this class. All one need do to use the supplied MII primitives is determine whether or not byte swapping is necessary. If the parameter BYTE_SWAP2 is defined as YES in **iraf.h** then **bswap2** will be called to swap 2 byte MII integers, producing an SPP **short** as output. If the parameter BYTE_SWAP4 is defined as YES then **bswap4** will be called to swap 4 byte MII integers, producing an SPP **long** as output.

5.6. Machine Constants for Mathematical Libraries

The most often used machine constants are parameterized in include files for use within SPP programs. Defined constants are the most readable and efficient way to represent machine constants, but not the most accurate for floating point quantities. Furthermore, SPP defined constants cannot be used within Fortran procedures; functions returning the machine constants are often used instead. The most widely used set of such functions appears to be those developed at Bell Laboratories for the *Port* mathematical subroutine library.

Mathematical Machine Constants

int = i1mach (parameter)	get INTEGER machine constants
real = r1mach (parameter)	get REAL machine constants
double = d1mach (parameter)	get DOUBLE PRECISION machine constants

These routines are used in many numerical packages, including the IEEE signal processing routines and the NCAR graphics software. Documentation is given in the Fortran sources; values of the constants have already been prepared for most of the computers used at scientific centers. The integer codes of the machine parameters are parameterized in **lib$mach.h** for use in SPP programs.

6. System Parameterization and Tuning

All machine dependent system and language parameters are defined in the two SPP include files **lib$iraf.h** and **lib$config.h**, in the C include file **cl$config.h**, and in the CL script file **lib$clpackage.cl**. Additional machine dependent include files will often be used (*should* be used) to implement the kernel for a particular machine but these are too host specific to be described here.

The include file **lib$iraf.h** is automatically loaded by the SPP whenever an SPP source file is preprocessed, hence the defines in **iraf.h** are effectively part of the SPP language. This global include file defines the language parameters (e.g., EOF, ERR) as well as many machine constants. The two configuration files **lib$config.h** and **cl$config.h** define additional constants pertaining to the host OS as well as various size limiting and heuristic parameters used to tune IRAF for optimum performance on a given host system. The CL script file **lib$clpackage.cl** contains **set environment** declarations for all system directories and default devices.

Documentation for the individual machine constants and system tuning parameters is maintained directly in the source files to ensure that it is up to date. Some of the parameters in **config.h** pertain only to the inner workings of the program interface and changes can be expected in successive releases of the IRAF system, without corresponding changes to the external specifications of the program interface.

7. Other Machine Dependencies

In the ideal world all of the machine dependence of the system would be concentrated into the kernel and a few include files. While this has been achieved for the scientific software some of the system utilities currently bypass the system interface and are machine dependent. This is not necessarily a bad thing; if a certain capability is only needed by one or two system utilities the machine dependence of the system will often be less if we take the simple way out and write a system dependent program than if we further expand the formal system interface.

The machine dependent utilities are in the packages **system** and **softools**. All machine dependent procedures are listed in the README files in the package directories. The string MACHDEP is placed in the source files to mark any machine dependent code segments. The machine dependent utilities are not essential to the use of the system, but are useful for maintaining the system. Examples of machine dependent software utilities include **make**, **xcompile**, and the generic preprocessor. These utilities will all eventually be replaced by portable programs. Machine dependent system utilities include **allocate** and **deallocate**, **edit** (the interface to the host editor), and **diskspace**. These utilities are actually just command level interfaces to the host system command interpreter, and are easy to modify for a new host.

7.1. Machine Dependencies in the CL

As noted earlier, from a structural design point of view the Command Language is an IRAF applications program; as such it is highly portable. The CL is not completely portable because it is written in C. Since the CL is written in C it cannot reference the standard include files **iraf.h** and **config.h**, and therefore has its own machine dependent **config.h** include file instead. Since the CL requires file i/o, process control and exception handling, formated i/o, the TTY interface, etc., it is interfaced to the IRAF program interface (the CL has a special Main of its own but this is portable).

The C code in the CL is purely numeric, like the Fortran in SPP based applications programs. All communication with the host system is via the IRAF program interface. The program interface is written in SPP, i.e., Fortran, and Fortran procedures cannot portably be called from C (see §3.1.2). To render the bulk of the code portable a special C callable subset of the program interface is defined for the CL, and all CL calls to program interface routines are via this interface. Efficiency is not a problem because the CL does little i/o; the CL is the control center of the system, and tends to be compute bound when it is not idle waiting for a command.

In summary, to interface the CL to a new system it is necessary to first edit the **cl$config.h**, which parameterizes the characteristics of the host system and which contains the system tuning parameters affecting the CL. The CL interface to the subset program interface consists of a set of rather simple interface procedures in the file **cl$machdep.h**. If you are lucky these will not have to be changed to port the CL to your host computer, but even if the routines have to be modified they are few in number and quite simple in nature.

8. Specifications for the Kernel Procedures

The remainder of this document consists of a summary of the machine dependent procedures, followed by the detailed technical specifications for each procedure. Only the specifications for the kernel primitives are given; the bitwise boolean primitives are part of the program interface and are documented elsewhere. Most likely either the Fortran or C code supplied for the bitwise primitives will be usable on a new host system without significant modification.

While the kernel consists of quite a few procedures, this does not necessarily mean that it is going to be harder to implement than if there were only a quarter or a third as many procedures. Our design goal was to minimize the complexity and size of the kernel, and we felt that it was more important to define simple, single function primitives than to minimize the *number* of primitives.

A large part of the kernel consists of device driver subroutines; on a system which provides device independent i/o these may map to the same low level procedures and a count of the number of driver subroutines will have little meaning. On a system which does not have device independent i/o it will be easier to implement separate procedures for each device than to try to make the host system look like it has device independent i/o (FIO already does that anyhow). Furthermore, the provision for separate drivers makes it easy to optimize i/o for a particular device, and makes it possible to dynamically interface new devices to FIO without modifying the basic system.

All kernel procedures are called by virtual operating system procedures in the program interface. The kernel procedures are *not* callable directly from applications code. Extensive error checking is performed by the high level code before a kernel procedure is called, hence error checking in kernel procedures is largely redundant and should be omitted if it will significantly compromise performance. Do not get clever with kernel procedures; *keep it simple*.

Summary of Kernel Constants		
name	*value*	*usage*
APPEND	4	write at EOF
BINARY_FILE	12	
BOF	−3	beginning of file
EOF	−2	end of file
EOS	'\0'	end of string delimiter
ERR	−1	function was unsuccessful
FI_DIRECTORY	2	directory file (**zfinfo**)
FI_EXECUTABLE	3	executable file
FI_REGULAR	1	ordinary file
FI_SPECIAL	4	special file
FSTT_BLKSIZE	1	device block size
FSTT_FILSIZE	2	file size, bytes
FSTT_MAXBUFSIZE	4	maximum transfer size
FSTT_OPTBUFSIZE	3	optimum transfer size
LEN_JUMPBUF	??	integer length of **zsvjmp** buffer
NEW_FILE	5	create a new file
NO	0	no (false)
OK	0	function successfully completed
PR_CONNECTED	1	connected subprocess
PR_DETACHED	2	detached process
PR_HOST	3	process spawned by host
QUERY_PROTECTION	2	query file protection (**zfprot**)
READ_ONLY	1	file access modes
READ_WRITE	2	
REMOVE_PROTECTION	0	remove file protection (**zfprot**)
SET_PROTECTION	1	set file protection (**zfprot**)
SZ_LINE	161	default textfile line length
TEXT_FILE	11	file types
WRITE_ONLY	3	
X_ACV	501	access violation
X_ARITH	502	arithmetic error
X_INT	503	interrupt
X_IPC	504	write to IPC with no reader
YES	1	yes (true)

Summary of Machine Dependent Procedures

Kernel Primitives

zawset (bestsize, newsize, oldsize, textsize)	adjust working set size
zcall*n* (procedure, arg1,...,arg*n*)	call by reference
zclcpr (pid, exit_status)	close connected subprocess
zcldir (chan, status)	close directory
zcldpr (jobcode, killflag, exit_status)	close or dequeue detached process
zdojmp (jumpbuf, status)	restore process status
zfacss (osfn, mode, type, status)	access file
zfaloc (osfn, nbytes, status)	preallocate a binary file
zfchdr (new_directory, status)	change directory
zfdele (osfn, status)	delete a file
zfgcwd (osdir, maxch, status)	get current working directory
zfinfo (osfn, out_struct, status)	get info on a file
zfmkcp (old_osfn, new_osfn, status)	make null copy of a file
zfpath (osfn, pathname, maxch, status)	osfn to pathname
zfprot (osfn, prot_flag, status)	file protection
zfrnam (old_osfn, new_osfn, status)	rename a file
zfsubd (osdir, subdir, new_osdir, maxch, nchars)	get subdirectory name
zfxdir (osfn, osdir, maxch, status)	extract OS directory prefix
zgfdir (chan, osfn, maxch, status)	get next OSFN from directory
zgtime (local_time, cpu_time)	get clock time and cpu time
zgtpid (pid)	get process id of current process
zintpr (pid, exception, status)	interrupt connected subprocess
zlocpr (proc, entry_point_address)	get EPA of a procedure
zlocva (variable, address)	get address of a variable
zmaloc (buffer, nbytes, status)	allocate a buffer
zmfree (buffer, status)	deallocate a buffer
zopcpr (process, inchan, outchan, pid)	open connected subprocess
zopdir (osfn, chan)	open a directory
zopdpr (process, bkgfile, jobcode)	open or queue detached process
zoscmd (cmd, stdout, stderr, status)	send a command to the host JCL
zpanic (errcode, errmsg)	panic exit
zraloc (buffer, nbytes, status)	reallocate a buffer
zsvjmp (jumpbuf, status)	save process status
ztslee (delay_in_seconds)	countdown timer
zxgmes (os_exception, errmsg, maxch)	get exception code and message
zxwhen (exception, new_handler, old_handler)	post an exception

Text File Device Drivers

device	*code*	*opn*	*cls*	*get*	*put*	*fls*	*not*	*sek*	*stt*
normal (disk)	tx	*	*	*	*	*	*	*	*
terminal	ty	*	*	*	*	*			*

Binary File Device Drivers							
device	*code*	*opn*	*cls*	*ard*	*awr*	*awt*	*stt*
normal (disk)	bf	*	*	*	*	*	*
line printer	lp	*	*		*	*	*
IPC	pr			*	*	*	*
static file	sf	*	*	*	*	*	*
magtape	mt	*	*	*	*	*	*

Magtape Device Primitives

zzopmt (drive, density, mode, oldrec, oldfile, newfile, chan)	open
zzclmt (chan, mode, nrecords, nfiles, status)	close
zzrdmt (chan, buf, maxbytes)	aread
zzwrmt (chan, buf, nbytes)	awrite
zzwtmt (chan, nrecords, nfiles, status)	await
zzrwmt (chan, status)	arewind

Bit and Byte Primitives

and,and[sl] (a, b)	bitwise and
or,or[sl] (a, b)	bitwise or
xor,xor[sl] (a, b)	exclusive or
not,not[sl] (a, b)	complement
bitpak (intval, bit_array, bit_offset, nbits)	integer → bitfield
int = **bitupk** (bit_array, bit_offset, nbits)	bitfield → integer
bytmov (a, aoff, b, boff, nbytes)	move an array of bytes
bswap2 (a, b, nbytes)	swap every pair of bytes
bswap4 (a, b, nbytes)	swap every 4 bytes
chrpak (a, aoff, b, boff, nchars)	pack chars into bytes
chrupk (a, aoff, b, boff, nchars)	unpack bytes into chars
strpak (a, b, maxchars)	SPP string → byte-packed string
strupk (a, b, maxchars)	byte-packed string → SPP string
acht_b (a, b, npix)	SPP datatype → unsigned byte
acht_u (a, b, npix)	SPP datatype → unsigned short
achtb_ (a, b, npix)	unsigned byte → SPP datatype
achtu_ (a, b, npix)	unsigned short → SPP datatype
miipak (spp, mii, nelems, spp_type, mii_type)	SPP → MII
miiupk (mii, spp, nelems, mii_type, spp_type)	MII → SPP
intlen = **miilen** (n_mii_elements, mii_type)	get length of MII array
int = **i1mach** (parameter)	get int machine constants
real = **r1mach** (parameter)	get real machine constants
double = **d1mach** (parameter)	get double machine constants