IRAF Standards and Conventions

Elwood Downey
George Jacoby
Vesa Junkkarinen
Steve Ridgway
Paul Schmidtke
Charles Slaughter
Douglas Tody
Francisco Valdes

Kitt Peak National Observatory*
August 1983

ABSTRACT

Clearly defined and consistently applied standards and conventions are essential in reducing the "number of degrees of freedom" which a user or programmer must deal with when using a large system. The IRAF system and applications software is being built in accord with the standards and conventions described in this document. These include system wide standards for data structures and files, standard coding practices, coding standards, and standards for documentation. Wherever possible, the IRAF project has adopted or adapted existing standards and conventions that are in widespread use in other systems.

*Kitt Peak National Observatory is operated by the Association of Universities for Research in Astronomy, Inc. under contract with the National Science Foundation.

Contents

IRAF Standards and Conventions

Elwood Downey
George Jacoby
Vesa Junkkarinen
Steve Ridgway
Paul Schmidtke
Charles Slaughter
Douglas Tody
Francisco Valdes

Kitt Peak National Observatory*
August 1983

1. Introduction

Clearly defined and consistently applied standards and conventions are essential in reducing the "number of degrees of freedom" which a user or programmer must deal with when using a large system. The user benefits from consistently applied naming conventions for packages and tasks, and from a logical and consistently applied scheme for ordering the parameters to a task. The programmer who must read code written by other programmers benefits from the application of good programming practices and a uniform style.

The IRAF system and applications software is being built in accord with the standards and conventions described in this document. These include system wide standards for data structures and files, coding standards, standards for numerical libraries, and standards for documentation.

Whenever possible, the IRAF project has adopted or adapted existing standards and conventions that are in widespread use in other systems. Examples are the standard filename extensions, which are adopted from UNIX (the IRAF software development system), and the coding standard for the SPP language, which is consistent with the coding standard for the C language, on which the design of the SPP language was based.

1.1. Official Acceptance Procedure

Software developed by programmers outside of the IRAF group must conform to the standards presented in this document to be accepted as a supported product, or to be included in the IRAF distribution.

Software developed by programmers within the IRAF group will be inspected periodically by another member of the IRAF group to check for adherence to the standards, for constructs that could cause transportability problems, and to ensure that the code is upwards compatible with future versions of the SPP language compiler and program libraries. IRAF group members will gladly provide this service to anyone outside the group who would like to have their code checked.

2. System Standards

This section defines those standards and conventions which pervade the system. An example of such a fundamental convention is one-indexing. Others include the standard for generating and using virtual file names, and the procedure naming convention for the system libraries.

---

*Kitt Peak National Observatory is operated by the Association of Universities for Research in Astronomy, Inc. under contract with the National Science Foundation.

2.1. Standard Data Structures

This section describes the fundamental data structures used by the IRAF system and applications tasks. An IRAF applications package should not access data structures other than those described here. Applications may build their own high level structures upon text or binary files if necessary, but the standard high level structures (particularly the imagefile and the datafile) should be used when applicable.

2.1.1. Text and Binary Files

The most primitive data structure in the IRAF system is the file. At the most basic level there are two types of files, *text files* and *binary files*.

Text files may contain only character data. The fundamental unit of storage is the line of text. Character data in text files is maintained in a form which is OS (operating system) dependent, and text files may be edited, printed, and so on with the utilities provided by the host OS. Character data is stored in text files in the character set of the host OS, which is not necessarily ASCII. Text files are normally accessed sequentially, and writing is permitted only at EOF (end of file).

Examples of text files include program source files, CL parameter files, list files, ASCII card image files, CL script files, and the session logfile. Text files are often used for descriptor files which are read at run time by table driven software.

Binary files are read and written only by IRAF tasks. The fundamental unit of storage is the *char*. Data is stored in binary files in the form in which it appears internally in IRAF tasks, without any form of type conversion. Binary files are generally not transportable between different machines. Binary files may (normally) be read and written at random.

Any device which supports reads and writes in some form may be made to appear to be a binary file, subject to possible restrictions on seeks and writing at EOF. FIO (the IRAF file i/o package) supports devices of arbitrary blocksize, and i/o to binary files is very efficient and may be optimized according to the type of access expected.

Examples of binary files include imagefiles, datafiles, a graphics stream, a FITS file, and memory.

Although a binary file may be used to store any kind of data, including text, files which contain only text should be maintained as text files.

2.1.2. Parameter Files

The **parameter file**, a text file, is used to store the parameters and associated information (type, mode, prompt, default value, etc.) for a task. Parameter files are read and written by the CL, and are normally invisible both to the user and to the applications task.

The *default* parameter file for a task must reside in the same directory as the executable file or script file associated with the task. The root name of the parameter file is the name of the task. Parameter files have the extension ".par".

The logical directory **uparm** should be defined by the user to provide a place to store *updated* versions of parameter files. When updating a parameter file, the CL will prepend the first two characters of the package name to the parameter file name (to avoid redefinitions), and save the resultant file in **uparm**. This package prefix should be omitted from the names of default parameter files in the package directory.

task.par	default parameter file
pk_task.par	updated parameter file (package *pk*...)

2.1.3. Imagefiles

The **imagefile** is used to store bulk data arrays of arbitrary dimension, size, and datatype. Images of up to seven dimensions are currently supported. The length of a dimension is limited by the size of a long integer on the host machine. A full range of datatypes, from unsigned char through complex, are supported.

The fundamental unit of storage for an imagefile is the *pixel*. All the pixels in an image must be of the same datatype. The dimensions, size, and datatype of an image are fixed when the image is created.

2.1.3.1. standard nomenclature for images

The axes of a two dimensional image divide the image into *lines* and *columns*. A three dimensional image consists of one or more *bands*, each of which is a two dimensional image, all of which are the same size and datatype.

The names of procedures, variables, and so on in software which accesses images should be derived from the standard names **line**, **column**, **band**, and **pixel**. The use of the term *row* in place of *line* is discouraged, despite the historical use of *row* at KNPO. The *line, column, band* nomenclature is a defacto international standard, not only in the image processing literature, but at most astronomical data reduction centers as well.

Examples of standard identifiers include *nlines*, *ncols*, *npix*, and *ndim*, referring respectively to the number of lines, columns, pixels, or dimensions to be operated upon.

2.1.3.2. definition of a pixel

Given an image of dimension N, a *pixel* is defined as the datum whose coordinates within the image are given by the subscript [x1,x2,...,xN], where the first index in each dimension has the value one, and where **i** is the *column* index, **j** the *line* index, **k** the band index, and so on. The dimensionality of the image is given by the number of subscripts. The value of a pixel is *not* a dimension.

If an array of pixels is to be interpolated, the question of the extent or size of a pixel arises. In the IRAF system a pixel is defined as a mathematical point, and has no extent. This is in contrast to some other systems, which have adopted the "physical" definition of a pixel, i.e., pixel *i* is assumed to extend from [i–0.5] to [i+0.5].

Thus, given an array of N pixels, an IRAF interpolant will return an indefinite value at the points [1–eps] and [N+eps], where *eps* is a very small number. An array of N pixels contains N–1 subintervals. If an array of N pixels is expanded by interpolating every 0.5 pixels, an array of 2N–1 pixels will result. Mapping an array of N pixels into an array of 2N pixels requires a stepsize of (N–1)/(2N–1) pixel units.

2.1.4. Datafiles

The **datafile** provides a *database management* capability for the IRAF system. The datafile is used to store **records**. A record consists of an ordered set of **fields**, each of which has a name, a datatype, and a value. The structure of a datafile is defined by the applications program, and a description of that structure is saved in the datafile itself. It is this self describing nature of datafiles which makes database management possible.

The datafile has many advantages over the old technique of writing an array of binary records in a headerless file, via FIO **write** calls. Datafiles are self documenting, can be manipulated by the standard database management tools, and the structure of the records in a datafile can be modified as a program evolves, without losing the capability to access old datafiles.

2.1.5. List Files

The **list file** is a text file, each line of which comprises one element of the list. Lists are used to drive tasks in batch or semibatch mode. A typical list defines a set of files, images, records, coordinates of objects, etc. to be processed by a task.

Lists should be maintained as text files to take advantage of the ability of the CL to process text files. Lists maintained in text form can be created by i/o redirection, and are easily edited, sorted, filtered, inspected, and so on. Lists can be input to tasks using list structured parameters, redirection of the standard input, and templates.

2.1.6. FITS

The FITS standard of the AAS and IAU [1] is the standard format for image data entering and leaving the IRAF system. The FITS format will be used both for image data transmitted by magnetic tape between machines, and for image data transmitted between machines by other means (i.e., via a network).

Proposed extensions to the FITS standard may provide a means for transmitting tabular data (such as a list), as well as an efficient means for transporting text files. These extensions will be implemented in the IRAF system when a draft standard is received from the FITS standards committee of the AAS.

2.2. Virtual File Names

A file name may be specified in a machine independent fashion, or as an OS dependent pathname. A machine independent filename is called a **virtual file name** (VFN). The ability of the system to deal with OS dependent filenames is intended primarily as a convenience feature for the user. Applications programs and CL script tasks should be written in terms of virtual file names for maximum transportability.

A virtual file name has the following form:

ldir$root.extn

where

field	*usage*
ldir	logical directory or device name
root	root or base file name
extn	extension denoting the type of file

The *ldir* and *extn* fields are optional. The logical directory field, if present, must be delimited by the character $. The backslash character can be used to escape characters such as $, if required in OS dependent filenames.

The root and extension fields may contain up to 20 characters selected from the set [a-zA-Z0-9_+−#.]. A file name may not contain any whitespace. The extension field should not exceed three characters. The extension field is separated from the root field by the character "." (dot). If the root field contains one or more occurrences of the dot character, the final dot delimited field is understood to be the extension, and the remaining fields are considered to be part of the root.

Purely numeric filenames are legal virtual file names. If the first character of a file name is a digit, the character "I" will be prepended to generate the OS pathname. Thus, the filenames "I23" and "23" refer to the same file. Numeric filenames are reserved for use by the user as a convenient way to name imagefiles, and should not be used in programs or script tasks.

2.3. Standard Filename Extensions

A number of standard filename extensions are defined to identify those types of files which are most commonly used in IRAF programs and by users of the IRAF system. These extensions reflect the selection of UNIX as the IRAF software development system, but transportability is not compromised since the extension field is part of a VFN (and is therefore mapped in a machine dependent way).

Standard Filename Extensions

Extension	Usage
.a	archive or library file
.c	C language source
.cl	Command Language script file
.com	global common declaration
.df	IRAF datafile
.f	Fortran 77 source
.h	SPP header file (contains global *defines*)
.hlp	*Lroff* format help text
.ms	*Troff* format text
.o	object module
.par	CL parameter file
.pix	pixel storage file (part of an imagefile)
.s	assembler language source
.x	SPP language source

Note that no extension is assigned for executable files (executable files are not directly accessed by IRAF programs or utilities). Certain of these extensions may have to be mapped into a different form in the process of converting a VFN to an OSFN (i.e., on most operating systems, ".a", ".f", ".o", and ".s" will be mapped into some other extension at file access time by the system interface routine *zmapfn*).

2.4. One Indexing

The IRAF system is one-indexed. This convention is applied without exception in the system software, and should be applied equally rigorously in applications code. Past systems (i.e., the KPNO IPPS system and the original KPNO Forth Camera system) have shown that mixing zero and one indexing in the same system is confusing, and is the source of many errors.

Note that the one-indexing convention applies to both numbering systems and offsets. Thus, the coordinates of the first pixel in a two dimensional image are [1,1], and the offset of the first character in a file is also one. Scaling an offset involves subtracting the constant one, a multiply or divide to perform the actual scaling, followed by the addition of the constant one.

The awkwardness of one-indexing for calculating offsets (in comparison with zero-indexing) is balanced by the logical simplicity of one-indexed numbering schemes. The one-indexing convention was selected for IRAF because numbering schemes are more visible to the user than is offset arithmetic, and because IRAF is a Fortran based system.

2.5. The Procedure Naming Convention for the System Libraries

With the exception of certain "language" level identifiers (**open**, **close**, **read**, **write**, **map**, **error**, etc.), all procedures in the packages comprising the IRAF program and system interfaces are named according to a simple convention.

The purpose of the procedure naming convention is to make procedure name selection logical and predictable, and to minimize collisions with the names of the procedures (and other external identifiers) used in applications programs. This latter problem is a serious matter in a large system which is Fortran based, due to the global nature of all procedure and global common names, and the restriction to six character identifiers.

The procedure naming convention should *not* be used to generate names for procedures in applications code. The procedure naming convention purposely results in rather obscure identifiers. This is necessary for system library routines, to minimize the possibility of collisions, but at the highest level (in applications code and in CL packages), readability is the most important consideration.

The names of system library procedures are generated by concatenating the following fields:

package_prefix // *opcode* // *type_suffix*

The package prefix identifies the package to which the procedure belongs, and is one to three characters in length. The opcode is a concise representation of the function performed by the procedure. The type suffix identifies the datatype of the function value or primary operand.

An example of the use of the procedure naming convention is the generic function **clgpar**, in the CLIO package. In this case, the package prefix is "cl", the opcode is "g" (get), and the (abstract) type suffix is "par". The generic function **clgpar** is implemented with the following set of typed procedures:

clgpar → clgetb, clgetc, clgets, clgeti, clgetl, clgetr, clgetd, clgetx

or, more concisely,

clgpar → clget[bcsilrdx]

2.5.1. Orthogonality

The procedure naming convention is an example of a three dimensional "orthogonal" naming convention. The VAX instruction set and associated mnemonics are another example. As we have seen, often two dimensions are sufficient (no type suffix) to encode the names of the procedures in a package. Occasionally it is necessary to have more than three dimensions, as in the following example from the image i/o package:

getpix, **putpix** → im[gp][pls][123][silrdx]

where the fields have the following significance:

im[get/put][pixel/line/section][dimension][datatype]

The five dimensional expression on the right side represents a total of 108 possible procedure names (*imgp1s*, etc.). A **getpix** or **putpix** statement is easily converted into a call to the appropriate low level Fortran subprogram by analyzing the subscript and applying the above generating function.

2.5.2. Standard package prefixes

A table of the package prefixes for the packages comprising the IRAF system libraries is shown below.

Standard Package Prefixes		
package	*prefix*	
CLIO	cl	command language i/o
FIO	f	file i/o
MEMIO	m (or mem)	memory i/o
VSIO	v	virtual structure i/o
IMIO	im	image i/o
MTIO	mt	magtape i/o
GIO	g	graphics i/o
VOPS (1-dim)	a	vector operators
VOPS (2-dim)	m	matrix operators
byte primitives	byt	
char utilities	chr	
error handling	err (or xer)	
pattern matching	pat	
string utilities	str	
process control	t	
exception handling	x	
OS interface	z	

2.5.3. Standard type suffixes

The type suffix is optional, and is used when the operator is implemented for several different types of data. The type suffix is a single character for the primary data types, but may be up to three characters for the abstract data types ("file", "string", etc.). The standard type suffixes are as follows:

Standard Type Suffixes		
datatype	suffix	
bool	b	(primary types)
char	c	
short	s	
int	i	
long	l	
real	r	
double	d	
complex	x	
file	fil	(abstract types)
string	str	
cursor	cur	
CL parameter	par	
character constant	cc	

2.6. Mapping of External Identifiers

The SPP language maps identifiers longer than the six characters permitted by the Fortran standard into identifiers of six or fewer characters. Both local and external identifiers are mapped. The mapping convention applies to all procedures in the system libraries.

A simple, fixed mapping is used to facilitate the use of symbolic debuggers without having to resort to a compiler listing. A simple mapping convention also makes it easier for the

programmer to foresee possible redefinitions.

The mapping function used is known as the "5+1" rule. The six character Fortran identifier is formed by concatenating the first five characters and the last character of the long identifier from the SPP source code. Underscore characters are ignored.

Identifiers in SPP source code should be chosen to maximize readability, without concern for the length of an identifier. The compiler will flag spelling errors and identifiers which map to the same six character Fortran identifier (if both identifiers are referenced in the same file).

Examples:

XPP identifier	*Fortran identifier*	
strmatch	STRMAH	(library procedure)
read_template	READTE	(procedure)
get_keyword	GETKED	(procedure)
ival_already_used	IVALAD	(boolean variable)
days_per_year	DAYSPR	(integer variable)

2.7. Conventions for Ordering Argument Lists

The convention for ordering argument lists applies to both CL tasks and compiled procedures. This convention should serve only as a guideline: in practice, other considerations (such as symmetry) may produce a more natural ordering.

Argument lists may contain operands and their dimensions, objects used for working storage, control parameters, and status return values (organized in that order). The types of operands may be further broken down into those which are input and those which are output, ordered with the input parameters at the left and the output parameters at the right.

More precisely, the ordering of operands and parameters in the argument lists of procedures and tasks is as follows:

(1) The principal operand or operands (data objects) dealt with by the procedure, ordered with input at the left and output at the right. Examples of primary operands include file names, file descriptors, image header pointers, vectors, and so on.

(2) Dimension parameters, offsets, position vectors, or other objects which can be considered part of the specification of an operand. If the operands in (1) are individually dimensioned, the dimension argument(s) should immediately follow the associated operand. If several operands share the same dimension arguments, these arguments should follow the last operand in the group.

(3) Objects used for working storage, and their dimensions.

(4) Any control parameters, flags, options, etc., used to direct the operation of the procedure. Unless there is another ordering which is clearly more logical, these should be arranged in alphabetical order.

(5) Status return parameter or parameters, if any.

Argument lists should be kept as short as possible if they are to be easily remembered by the programmer (ideally, no more than three arguments). Short argument lists decrease the coupling between modules, increasing modularity and making programs easier to modify. Any procedure which requires more than five arguments should be carefully examined to see if it should be broken into several smaller procedures.

3. Coding Standards

Programs are read far more often than they are written. The readability of a program is a function of the **style** in which it is written. The effectiveness of a particular style in enhancing the readability of a program is increased when that style is applied consistently throughout the entire program. The readability of the code within a *system* is maximized when a single, well designed style is applied consistently throughout the system. Since large systems are written by many people (though often read by a single person), it is necessary to document the standard programming style for the system, as clearly as can be done.

The standard programming style for a system is a major part of the **coding standard** for that system (though not the whole story). The benefits and difficulties of coding standards are well summarized by the following excerpt from a paper describing the evolution of the *Ingres* data base management system [2]:

> *"The initial reaction was exceedingly negative. Programmers used to having an address space of their own felt an encroachment on their personal freedom. In spite of this reaction, we enforced standards that in the end became surprisingly popular. Basically, our programmers had to recognize the importance of making code easier to transfer to new people, and that coding standards were a low price to pay for this advantage..."*
>
> *"Coding standards should be drawn up by a single person to ensure unity of design; however, input should be solicited from all programmers. Once legislated, the standards should be rigidly adhered to."*

The standard language for IRAF system and applications code is the Subset Preprocessor Language (SPP), which was patterned after the C language of Kernighan and Ritchie [3]. Much of the text in the following pages was taken almost verbatim from reference [4], which defines the coding standard adopted at Bell Labs for the C language. Since such a well defined (and widely used) standard already exists, we have adopted the C coding standard as the core of the standard for the SPP language.

3.1. General Guidelines

In this section we discuss the philosophy governing the decomposition of the IRAF system into packages and tasks. The same principles are seen to apply to the decomposition of tasks or programs into separately compiled procedures.

Our intent here is to summarize the structural characteristics expected of a finished applications package. Once a package has been coded and tested, however, it is too late to change its structure. The functional decomposition of a package or program into a set of modules, the selection of names for the modules, and the definition of the parameters of each module, is the purpose of the detailed design process. A discussion of the techniques and tools used to perform a detailed design is beyond the scope of this document.

3.1.1. Packages and Tasks

The IRAF system and applications code is organized into **packages**, each of which operates upon a particular kind of data. These packages are independent, or are loosely coupled by the datafiles, imagefiles, or lists on which they operate.

Close coupling between packages (for example, by means of specialized data structures) should be avoided. Leave the coupling of modules from different packages to the user, or write high level script tasks ("canned" procedures) to streamline commonly performed operations, *after* the packages involved have been designed and coded.

A package consists of a set of **tasks**, each of which should perform a *single function*, and

all of which operate on the package data structures. The name of each task should be carefully chosen to identify the function performed by the task (a novice user should be able to guess what function the task performs without having to read the documentation). Command names should not be abbreviated to the point where they have meaning only to the package designer.

The tasks in a package should be *data coupled*, meaning that their operation is defined entirely in terms of the package data structures. Avoid *control coupling*, which occurs when one task controls the functioning of another by passing a control parameter or switch. A task should not modify another tasks parameters, nor should it modify its own input parameters.

A CL callable task may reference its own local parameters, plus two levels of **global parameters** (the package parameters and the CL parameters). Global parameters should be used with care to avoid tasks which are highly coupled. For example, if a task were to use the the CL "scratch" parameters **i** and **j** for loop control variables, that task would be strongly coupled to any other task in the system, now and in the future, which also references the global parameters **i** and **j** (with disastrous results). The CL scratch parameters are provided for the convenience of the user: they should not be used by tasks.

Global parameters can actually reduce the coupling between tasks when the alternative would be to add a parameter to the set of local parameters for each task in the package. Such parameters are normally set only by the user (or by a user script task), and are *read only* to all tasks in the package. Examples of such parameters might be the names of the package datafiles, or parameters which describe the general characteristics of the data to be operated upon. If in doubt, use a local parameter instead of a global parameter.

A task may be implemented as a **script task**, written in the CL, or as a compiled procedure or **program**, written in the SPP language. Any number of related or unrelated programs may be linked together to form a single executable **process**. The decision to implement a task in the CL or in the SPP language is irrelevant to the package designer, as is the grouping of programs to form physical processes.

3.1.2. Procedures

The guidelines for implementing a program as a set of separately compiled **procedures** are similar to those for decomposing a package into a set of tasks. *Each procedure should perform a single function, should be well named, should be data coupled, and should have as few parameters as possible*.

Procedures which perform a single function are less complex than multiple function procedures, tend to be less strongly coupled to their callers, and are more likely to be useful elsewhere in the program, and in future programs. A program structured as a hierarchy of single function, minimally coupled procedures is highly modular, and generally much easier to modify, than a program consisting of multiple function (monolithic), strongly coupled procedures. Reducing the coupling between procedures makes it less likely that a change to one procedure will affect the functioning of another procedure somewhere else in the system.

It has long been argued that a monolithic procedure is more efficient than one which calls external procedures to perform subfunctions. While there is some truth to this claim, efficiency is only one of the measures of the quality of software. Other factors such as reliability, robustness, flexibility, transportability, simplicity, and modifiability are often more important. Furthermore, it is almost always true that five or ten percent of the code accounts for ninety percent of the execution time, and it will prove easier to optimize that five or ten percent of the code if it is in the form of isolated, single function procedures (a small, simple procedure is easily replaced by an equivalent routine written in assembler, for example).

A section of code which is common to two or more modules, which is **functional** (performs a single, well defined function), and which is not strongly coupled to the rest of the code in the parent module, should be extracted to form a separate module. Not only does this reduce the amount of code which must be tested and debugged, it also makes the program easier to modify, since only a single section of code must be changed to modify the function in question.

Less obviously, a section of code should be extracted to form a new module even if the new module is only called from one other module, if the new module is functional, and is likely to be useful in future programs. A new module should also be created if doing so removes a sizable section of code from the parent module, significantly reducing the complexity of the parent module (provided the new module is functional and not strongly coupled). If the control flow of a procedure is so deeply nested that statements will no longer fit on a line, that is an indication that code should be extracted to form a new module.

The name of a procedure, like that of a task, should be carefully selected to identify the function performed by the procedure. *The function of each subprocedure referenced by a procedure should be evident to the reader, without having to go look up the source for the individual subprocedures.* For similar reasons, the function of each of the **arguments** of a subprocedure should be evident without having to look up the source or documentation for the procedure. The **define** feature of the SPP language is particularly useful for parameterizing argument lists.

Reducing the number of arguments to a procedure reduces the coupling of the procedure to its callers, making the procedure easier to modify and use, reducing the possibility of a calling error, and usually increasing the functionality (usefulness) of the procedure. Most procedures should have no more than three arguments: procedures with more than five arguments should be examined to see if they should be decomposed into several smaller procedures.

Psychologists have shown that one $8^1/_2$ by 11 inch sheet of paper (i.e., one page of a computer listing) contains about the amount of information that most people can comfortably handle at one time. Procedures larger than one or two pages should be examined to see if they should be broken down further. Conversely, procedures which contain fewer than five lines of code should be examined to see if they should be merged into their callers. If a procedure contains more than ten declarations for local variables or arrays, that is another indication that the procedure probably needs to be decomposed into smaller functional units.

A program is more resistant to changes in the external environment (and therefore more transportable) if that part of the program which interfaces to the outside world is isolated from the part which processes the data. This tends to happen automatically if the "single function" guideline is followed, but nonetheless one should be consciously aware of the need to *isolate those portions of a program which get parameters, access external data structures, and format the output results.*

Numerical routines, transformations, and so on should almost always be implemented as separate procedures. These are precisely those parts of a program which are most likely to be useful in future programs, and they are also among the most likely to be modified, or replaced by functionally equivalent modules, as the program evolves.

3.2. Languages

The standard language for IRAF systems and applications code is the SPP language [5], which is mechanically translated into Fortran during compilation. Fortran itself may be used for purely numerical routines (no i/o) which are called from programs written in the SPP language.

IRAF programs must be written in the SPP language, rather than Fortran, because the routines in the IRAF i/o libraries are callable only from the SPP language. The IRAF i/o libraries are interfaced to the SPP language because they are *written* in the SPP language.

3.2.1. The SPP Language

The IRAF Subset Preprocessor language (SPP) implements a subset of the full language scheduled for development in 1984. The SPP language is defined by the SPP Reference Manual [5]. Be warned that present compilers for the SPP language accept constructs that are not permitted by the language standard. As better compilers become available, programs using such constructs (i.e., parenthesis instead of brackets for array subscripts), will no longer compile. If

you are not sure what the language standard permits, have your code checked periodically by someone who is familiar with the standard.

3.2.2. The Fortran Language

The Fortran language is defined by the ANSI standards document ANSI X3.9-1978 [6]. Be warned that most Fortran compilers accept constructs that are not permitted by the language standard. When a Fortran module developed on one machine is ported to another, programs using such constructs (i.e., the DO WHILE and TYPE constructs provided by the DEC Fortran compilers), will no longer compile, or will run incorrectly.

Fortran is used in IRAF applications only for numerical subroutines and functions, such as mathematical library routines. The following Fortran statements should not be used in Fortran subprograms that are to be called from an IRAF program (use of one of these statements would probably result in a loader error):

all statements which involve i/o
CHARACTER
BLOCK DATA
(blank) COMMON
PAUSE
PROGRAM
STOP

The SPP datatypes **int**, **real**, **double**, and **complex** are equivalent to the Fortran datatypes INTEGER, REAL, DOUBLE PRECISION, and COMPLEX. These are the only datatypes which should be used in IRAF callable Fortran modules.

There is no single widely accepted coding standard for the Fortran language. Fortran code being ported into the IRAF system should remain in the form in which it was originally written, except for the removal of the statements listed above. If extensive modifications are required, the modules should be recoded in the SPP language. All new software should be written in the SPP language.

3.3. Standard Interfaces

The programmer should be familiar with the routines in the packages comprising the IRAF program interface, and should use these routines where applicable. This practice reduces the amount of code which must be written and debugged, and simplifies the task of the newcomer who must read and understand the code for the package. Furthermore, optimizations are often possible in system library routines which would be inappropriate or difficult to perform in applications modules.

Only procedures which appear in the documentation for a package (the **external specifications** of the package) should be called from programs external to the package. The external specifications of a package define the **interface** to the package. The major interfaces of a large system are normally documented and frozen early in the lifetime of the system. Freezing an interface means that its external specifications stop changing; *the internal specifications of the code beneath the interface can and will continue to change as the system evolves.*

Calling one of the internal, undocumented procedures in a package, or directly accessing the internal package data structures, is known as **bypassing** or **violating the interface**. Violating an interface is a serious matter because it results in code which works when it is coded and tested, but which mysteriously fails some months later when the programmer responsible for maintaining the called package releases a new version which has been modified internally, even though its external specifications have not changed.

Interfaces are often violated, albeit unintentionally, when a programmer copies the source for one of the documented procedures in a package, changes the name, and modifies it to do his bidding. This may result in the programmer getting his or her job done a bit faster, but must be avoided at all costs because sooner or later the resultant software system is going to fail.

Worse yet, there is no guarantee that when the failure occurs, it will occur in that part of the system written by the programmer who violated the interface. Activation of the offending module may corrupt the internals of the called package, resulting at some indefinite point later in an apparently unrelated error, which may be difficult to trace back to the module which originally violated the interface. Typically, the error will appear only infrequently, when the system is exercised in a certain way.

Violating interfaces results in an *unreliable system.* If such a problem as that described above happens very often, the systems programmer charged with maintaining the system will become afraid to change systems code, and the result will be a system which is hard to modify, and which will eventually have to be frozen internally as well as externally. At that point the system will no longer be able to evolve and grow, and eventually it will die.

Other common ways in which interfaces are violated include communicating directly with the host operating system (bypassing the system interface), communicating directly with the CL, or sending explicit escape sequences to a terminal. If one were to access an external image format by calling C routines interfaced directly to UNIX, for example, one would be bypassing the system interface, and the transportability of the applications program which did so would be seriously compromised.

The CL interface may be violated by sending an explicit command to the CL, by reading from CLIN or writing to CLOUT, or by directly accessing the contents of a parameter file. Sending a command to the CL violates the CL interface because a task must know quite a bit about the syntax of an acceptable CL command, as well as the capabilities of the CL, to send such a command.

From the point of view of a task, the CL is simply a data structure, the fields of which (parameters) are accessed via **clget** and **clput** procedures. Programs which do not expect the CL to be anything more than a data structure will be immune to changes in the CL as it evolves. In the future we might well have several different command languages, each with a different syntax and capabilities. An IRAF task which does not attempt to bypass the CL interface will be executable from any of these command languages, without modification or even recompilation.

3.4. Package Organization

Each package should be maintained in its own directory or set of directories. The name of the **package directory** should be the name of the package, or a suitable abbreviation.

A package consists of source files (".x", ".f", ".cl", ".h", ".com"), documentation (".hlp" and ".ms" files), parameter files (".par"), and executable modules. If the package is small it will be most convenient to maintain the package in a single directory. The package directory should contain a file named "Readme" or "README", describing the function of the package, and refering the reader to more detailed package documentation.

If a package is too large to be maintained in a single directory, two subdirectories named **bin** and **doc** should be created. The package directory should contain the sources, the Readme file, and a file named "Makefile" if *Make* is used to maintain the package. The **bin** directory should contain the executable files and the default parameter files (the CL requires that these be placed in the same directory). The **doc** directory should contain the design documentation, reference manuals, user's guides, and manual pages.

The programmer should develop and maintain a package in directories located within the programmer's own directory system. When the package is released, an identical set of directories will be created within the IRAF directory system. Subsequent releases of new versions of the package will be a simple matter of copying the files comprising the new package into the

IRAF directories, and documenting the differences between the old and new versions of the package.

This procedure makes a clear distinction between the current release of the package and the experimental version, buffering the user from constant changes in the software, yet giving the programmer freedom to experiment and develop the software at will.

3.5. Tasks and Processes

The **task** statement of the SPP language is used to group one or more compiled tasks (programs) together to form an executable process. As noted earlier (§3.1.1), the grouping together of programs to form a physical process is a detail which is irrelevant to the structure of the package.

The grouping of several programs together to form a single process can, however, result in significant savings in disk space by replacing a number of executable files by a single (slightly larger) file. The same technique can also have a significant impact on the efficiency of a CL script, by eliminating the overhead of process initiation required when each task called by the CL resides in a different executable file. In the case of a simple task which executes in a few tens of milliseconds, the overhead of process initiation could easily exceed the time required to actually execute the task by one or two orders of magnitude.

The user of a package may well wish to change the way in which programs are grouped together to form processes, in order to minimize the overhead of process initiation when the programs are executed in a sequence peculiar to the user's application. To make it easier to modify the grouping of tasks to form processes, the **task** statement should be placed in a file by itself, rather than including it in the file containing the source for a program.

In other words, *the task statement should be decoupled from the source for the programs which it references.* If this is done, then regrouping is a simple matter of editing the file containing the task statement, editing the package script task (which associates tasks with executable files), and compiling the new task statement.

3.6. File Organization

Each program or task in a package should be placed in a separate file. The name of the file should be the same as the name of the top level module in the file. This practice makes it easy to locate the source for a module, and speeds compilations. The *Make* and *Mklib* utilities are particularly useful for automatically maintaining programs and libraries consisting of many small files.

A file consists of various sections that should be separated by several blank lines. The sections should be organized as follows:

(1) Any header file includes should be the first thing in the file.

(2) A prologue describing the contents of the file should immediately follow the includes. If the prologue exceeds four lines of text, it should be enclosed in **.help** ... **.endhelp** delimiters, rather than making each line of text a comment line. Large blocks of texts are easier to edit if maintained as help blocks, and placing such program documentation in a help block makes it accessible to the online **help** utilities.

(3) Any parameter or macro definitions that apply to the file as a whole are next.

(4) The procedures come last. They should be in a meaningful order. Top-down is generally better than bottom up, and a "breadth-first" approach (functions on a similar level of abstraction together) is preferred over depth-first (functions defined as soon as possible after their calls). Considerable judgment is called for here. If defining large numbers of essentially independent utility procedures, consider alphabetical

order.

3.7. Header Files

Header files are files that are included in other files prior to compilation of the main file. A header file contains a number of **define** statements, defining symbolically the constants, structures, and macros used by a subsystem. Some header files are defined at the system level, like *<imhdr.h>* which must be included in any file which accesses the image header structure. Other header files are defined and used within a single package.

Absolute pathnames should not be used to reference header files. Use the *<name>* construction to reference system header files. Non-system header files should be in the same directory as the source files which reference them. Header files should be functionally organized, i.e., declarations for separate subsystems should be in separate header files. The name of the header file should be the same as the name of the associated subsystem, and the extension should be ".h". For example, if the name of a package were "imio", the package header file would be named "imio.h".

Header files should not be nested. Nesting header files can cause the contents of a header file to be seen by the compiler more than once. Furthermore, the dependence of a source file on a header file should be made clear and explicit. The pattern matching utilities (**match** or **grep**) are often used to search for the name of a particular header file, to determine which source files are dependent upon it.

3.8. Comments

Well structured code with self explanatory procedure and variable names does not need to be extensively commented. At a minimum, the contents of the file should be described in the file prologue, and each procedure in the file should be preceded by a comment block giving the name of the procedure and describing what the procedure does.

Comments within the body of a procedure should not obscure the code. Large procedures should be broken up into logical sections (groups of statements which perform some function that can be understood in the abstract), with one or more blank lines and (optionally) a comment preceding each section. The comment should be indented to the same level as the code to which it refers.

The amount of commenting required depends on the complexity of the code. Generally speaking, if a comment appears every five lines or less, the code is either overcommented or too complex. If a one page procedure contains no comments, it is probably undercommented.

Short comments may appear on the same line as the code they describe, but they should be tabbed over far enough to separate them from the statements. If more than one short comment appears in a block of code, they should all be tabbed to the same column.

Example 1: Compute the mean and standard deviation of a sample.

```
# Accumulate the sum and sum of squares of those pixels
# whose value is within range and not indefinite.
do i = 1, npix
    if (sample[i] != INDEF) {
        value = sample[i]
        if (value >= lcutoff && value <= hcutoff) {
            ngpix = ngpix + 1
            sum = sum + value
            sumsq = sumsq + value ** 2
        }
    }

# Compute the mean and standard deviation (sigma).
switch (ngpix) {
case 0:                             # no good pixels
    mean = INDEF
    sigma = INDEF
case 1:                             # exactly one good pixel
    mean = sum
    sigma = INDEF
default:
    mean = sum / ngpix
    temp = sumsq / (ngpix-1) - sum**2 / (ngpix * (ngpix-1))
    if (temp < 0)                   # possible with roundoff error
        sigma = 0.0
    else
        sigma = sqrt (temp)
}
```

3.9. Procedure Declarations

Each procedure should be preceded by several blank lines and a block comment that gives the name of the procedure and a short description of what the procedure does. If extensive comments about the arguments or algorithm employed are required, they should be placed in the prologue rather than in the procedure itself.

The prologue should be followed by one or two blank lines, then the **procedure** statement, which should be left justified in column one. A blank line should follow, followed by the declarations section, then another blank line, and lastly the body of the procedure, enclosed in left justified **begin** ... **end** statements. The declarations should start in column one, and the list of objects in each declaration should begin at the first tab stop. The body of the procedure should be indented one full tab stop.

If the function of an argument, variable, or external function is not obvious or is not documented in the prologue, it should be declared alone on a line with an explanatory comment on the same line. In general, well chosen identifiers are preferable to explanatory comments, which tend to produce clutter, and which are more likely to be misleading or wrong. Arguments should be declared first, followed by local variables and arrays, followed by function declarations, with the **errchk** declaration, common block includes, string declarations, and **data** initialization statements last.

Example 2

```
# ADVANCE_TO_HELP_BLOCK -- Search a file for a help block
# (block of text preceded by ".help" left justified on a
# line).  Upon exit, the line buffer will contain the text
# for the help statement, if one is found.  EOF is returned
# for an unsuccessful search.

int procedure advance_to_help_block (fd, line_buffer)

int     fd                                  # file to be searched
char    line_buffer[SZ_LINE]
int     getline(), strmatch()
errchk  getline

begin
        while (getline (fd, line_buffer) != EOF)
            if (strmatch (line_buffer, "^.help") > 0)
                return (OK)

        return (EOF)
end
```

3.10. Statements

The format of both simple and compound statements is the same, except that the body of a compound statement is enclosed in braces. The body or executable part of a statement should begin on the second line of the statement, and should be indented one more level than the first line. Each successive level should be indented four spaces more than the preceding level (every other level is aligned on a tab stop). The opening left brace should be at the end of the first line, and the closing right brace should be alone on a line (except in the case of **else** and **until**), indented to the same level as the initial keyword.

3.10.1. Statement Templates

Templates are shown only for the compound form of each statement. To get the template for the non-compound form, omit the braces and truncate the statement list to a single statement. The **iferr** statement is syntactically equivalent to the **if** statement, and may be used wherever an **if** could be used.

If a compound statement extends for many lines, the readability of the construct is often enhanced by inserting one or more blank lines into the body of the compound statement. In the case of a large **if else**, for example, a blank line (and possibly a comment) might be added before the **else** clause. Similarly, blank lines could be inserted before an **else if**, a **then**, or a **case**.

```
if (expr) {
    <statement>
    <statement>
}
```

```
iferr (statement) {
    <statement>
    <statement>
}

iferr {
    <statement>
    <statement>
} then {
    <statement>
    <statement>
}

if (expr) {
    <statement>
    <statement>
} else {
    <statement>
    <statement>
}
```

The **else if** construct should be used for general multiway branching, when the logical conditions for selecting a particular branch are too complex to permit use of the **switch case** construct.

```
if (expr) {
    <statement>
} else if (expr) {
    <statement>
} else if (expr) {
    <statement>
}
```

The **for** statement is the most general looping construct. The **do** construct should be used only to index arrays (i.e., for vector operations). The value of the index of the **do** loop is undefined outside the body of the loop. The **for** statement should be used instead of the **do** if the loop index is needed after termination of the loop. The **repeat** construct, without the optional **until**, should be used for "infinite" loops (terminated by **break**, **return**, etc.).

```
for (i=1;  i <= MAX;  i=i+1) {
    <statement>
    <statement>
}

do i = 1, npix {
    <statement>
    <statement>
}
```

```
while (expr) {
    <statement>
    <statement>
}

repeat {
    <statement>
    <statement>
} until (expr)
```

The **switch case** construct is preferred to **else if** for a multiway branch, but the cases must be integer constants. The cases should not be explicit or "magic" integer values; use symbolically defined constants. Explicit character constants are permissible, but often it is best to define character constants symbolically too. A number of common character constants are defined in the system include file *<chars.h>*.

```
switch (expr) {
case ABC:
    <statement>
case DEF, GHI, JKL:
    <statement>
default:
    <statement>
}
```

The **printf** statement is a compound statement, since the *parg* calls are logically bound to the **printf**. Although braces are not used, the body of the statement should be indented one level to make the connection clear. Printf statements must not be nested.

```
call printf (format_string)
    <parg_statement>
    <parg_statement>
```

The **null statement** should be used whenever a statement is required by the syntax of the language, but the problem does not require that a statement be executed. Null cases are often added to switch statements to reserve cases, even though the code to be executed for the case has not yet been implemented.

Example 3

```
# Skip leading whitespace.
for (ip=1;  IS_WHITE(str[ip]);  ip=ip+1)
    ;
```

3.11. Expressions

Whitespace should be distributed within an expression in a way which emphasizes the major logical components of the expression. For simple expressions, this means that all binary operators should be separated from their operands by blanks. In an argument list, a blank should follow each comma. Keywords and important structural punctuation like the brace

should be separated from the neighboring left or right parenthesis by a blank. Complex expressions are generally clearer if whitespace is omitted from the "inner" expressions.

Example 4:

```
alpha = beta + zeta
a = (a + b) / (c * d)
p = ((p-1) * SZ_DOUBLE) / SZ_INT + 1
IM_PIXFILE(im) = open (filename, READ_ONLY, BINARY_FILE)
a[i,j] = max(minval, min(maxval, a[i-1,j]))
```

By convention, whitespace is omitted from all but the most complex array subscript expressions, and the left square bracket is not separated from the array name by a blank. A unary operator should not be separated from its operand by a blank.

The system include file <*ctype.h*> defines a set of macros which should be used in expressions involving characters. For example, IS_WHITE tests whether a character is a whitespace character (see Example 3), IS_DIGIT tests whether a character is a digit, and IS_ALNUM tests whether a character is alphanumeric.

3.12. Constants

Numerical constants should not be coded directly. The **define** feature of the SPP language should be used to assign a meaningful name. This practice does much to enhance the readability of code, and also makes large programs considerably easier to modify, since one need only change the *define*. Defined constants which are referenced by more than one file should be placed in an ".h" include file.

A number of numerical constants are predefined in the SPP language. A full list is given in reference [5]. Some of the more commonly used of these global constants are shown below. To save space, those constants pertaining to i/o (READ_ONLY, TEXT_FILE, STDIN, STDOUT, etc.) are omitted, as are the type codes (TY_INT, TY_REAL, etc.), and the type sizes (SZ_INT, SZ_REAL, etc.).

Selected Predefined Constants		
constant	*datatype*	*meaning*
ARB	i	arbitrary dimension, i.e., "char lbuf[ARB]"
BOF, BOFL	i,l	beginning of file (use BOFL for seeks)
EOF, EOFL	i,l	end of file (use EOFL for seeks)
EOS	i	end of string
EPSILON	r	single precision machine epsilon
EPSILOND	d	double precision machine epsilon
ERR	i	error return code
INDEF	r	indefinite valued pixel
MAX_EXPONENT	i	largest exponent
MAX_INT	i	largest positive integer
MAX_REAL	r	largest real number
NO	i	opposite of YES
NULL	i	invalid pointer, etc.
OK	i	opposite of ERR
SZB_CHAR	i	size of a char, in machine bytes
SZ_FNAME	i	maximum size of a file name string
SZ_LINE	i	maximum size of a line of text
SZ_PATHNAME	i	maximum size of an OS pathname
YES	i	opposite of NO

3.13. Naming Conventions

Keywords, variable names, and procedure and function names should be in lower case. The names of macros and defined parameters should be in upper case. The prefix SZ, meaning **sizeof**, should be used only to name objects which measure the *size of an object in chars*. Other prefixes like LEN, N, or MAX should be used to name objects which describe the number of elements in an array or set.

For example, the system wide predefined constant SZ_LINE defines the maximum size of a line of text, in units of chars, while SZ_FNAME defines the maximum size of a file name string, also in chars. Since space in structures is allocated in struct units rather than chars, the constant defining the size of the FIO file descriptor structure is named LEN_FIODES, *not* SZ_FIODES.

4. Portability Considerations

IRAF programs tend to be highly transportable, due to the machine and device independent nature of the SPP language and the program interface libraries. Nonetheless, it is possible (unintentionally or otherwise) to produce machine or device dependent programs. A detailed discussion of the most probable trouble areas follows. The programmer should be aware of these pitfalls, but highly transportable programs can be produced merely by applying the following simple guidelines: *(1) choose the simplest, not the cleverest solution, (2) write modular, well structured programs, and (3) use the standard interfaces.*

4.1. keep it simple

Simple, modular programs, structured according to the guidelines in §3.1, are easy to understand and modify. Even the best programs are unlikely to be completely portable, because they will only have been tested and debugged on one or two systems by their author. Therefore the transportability of a program is significantly increased if it easy for someone who is unfamiliar with the code to quickly find and fix any machine dependencies. A package of **verification routines** are extremely useful when testing software on a new system, and ideally should be

supplied with each package, along with sample output.

4.2. use the standard interfaces

Much care has gone into making the standard interfaces as machine and device independent as possible. By using the standard interfaces in a straightforward, conventional fashion, one can concentrate on solving the immediate problem with confidence that a highly transportable and device independent program will automatically result.

The surest way to produce a machine or device dependent program is to bypass an interface. This fact is fairly obvious, but it is not always easy to tell when an interface is being bypassed (see §3.3 for examples). Furthermore, by bypassing an interface, one may be able to provide some feature that would be difficult or impossible to provide using the standard interfaces. In some cases this may be justified (provided transportability is not a requirement), but often the feature is cosmetic, and does not significantly increase the functionality of the program. The correct procedure is to request that the interface causing the problem be extended or refined.

4.3. avoid machine dependent filenames

Machine dependent filenames should not appear in source files. Files which are referenced at compile time, such as include files, should be placed either in the package directory or in the system library directory, to eliminate the need to use a pathname. Program files accessed at runtime must be referenced with a pathname, since the runtime current working directory is unpredictable. In this case a VFN should be used. The logical directory for the VFN should be defined in the package script task.

4.4. isolate those portions of a program which perform i/o

This fundamental principle is especially important when one attempts to transport an applications program from one reduction and analysis system to another, since the interfaces will almost certainly be quite different in the two systems. Encapsulating that part of the program which does i/o reduces the amount of code which must be understood and changed to bring up the package on the new system.

4.5. keep memory requirements to a reasonable level

Not all machines have large address spaces, nor do all machines have virtual memory. Virtual memory seems simple, but it is not; to use it effectively one must know quite a bit about how virtual memory is implemented by the local OS, and implementations of virtual memory by different operating systems differ considerably in their characteristics and capabilities. Using virtual memory effectively is not just a matter of accessing large arrays in storage order. If one can do that, then there is little justification for writing a program which is dependent on virtual memory.

It is possible to write down a set of guidelines for using virtual memory effectively and in a reasonably transportable manner, if one considers only large virtual memory machines. These guidelines are complex, however, and such a discussion is beyond the scope of this document. It must be recognized that any dependence on virtual memory seriously restricts the transportability of a program, and the use of virtual memory should only be considered if the problem warrants it.

The best approach for most applications is to restrict the memory requirements of a program to the amount of per-process *physical* memory which one can reasonably expect to be available on a modern supermini or supermicro. An upper limit of one quarter of a megabyte is recommended for most programs. Programs which need all the memory they can get, but which can dynamically adjust their buffer space to use whatever is available, should use the **begmem** system call to determine how much memory is available in a system independent way.

4.6. make sure argument and function datatypes match

Compilers for the SPP and Fortran languages do not verify that a function is declared correctly, or that a procedure or function is called with the correct number and type of arguments. This seriously compromises the transportability of programs, because *whether or not a type mismatch causes a program to fail depends on the machine architecture*. Thus, a program may work perfectly well on the software development machine, but that does not indicate that the program is correct.

The most dangerous example of this is a procedure which expects an argument of type short or char. If passed an actual argument of type integer, as happens when the actual argument is an integer constant (i.e., NULL, 1, ('a'+10), etc.), we have a type mismatch since the corresponding Fortran dummy argument is (usually) declared as INTEGER*2, while the actual argument is of type INTEGER. Whether or not the program will work on a particular machine depends on how the machine arranges the bytes in an integer. Thus, the mismatch will go undetected on a VAX but the program will fail on an IBM machine.

A similar problem occurs when a boolean dummy argument or function is declared as an integer in the calling program, and vice versa. In this case, whether or not the program works depends on what integer values the compiler uses to represent the boolean constants **true** and **false**. The danger is particularly great if the compiler happens to use the constants one and zero for true and false, since the integer constants YES and NO are equivalent in value and similar in function.

The technique used by the Fortran compiler to implement subroutine and function calls determines whether or not **calling a function as a subroutine**, or calling a subprogram with the **wrong number of arguments** will cause a program to fail. For example, if the arguments to a subroutine are placed on the hardware stack during a subroutine call, as is done by compilers which permit recursive calls, then most likely the stack will not be popped correctly upon exit from the subroutine, and the program will fail. On a machine which statically allocates storage for argument lists, the problem may go undetected.

4.7. do not use output arguments as local variables

This section is not directly relevant to the issue of portability, but is included nonetheless because the topic presented here is logically related to that discussed in the previous section.

The output or status arguments of a procedure should be regarded as *write-only*. Output arguments should not be used as local variables, i.e., should not appear in expressions. Likewise, the function value of a typed procedure should not be used as a local variable.

To see why this is important, consider a procedure *alpha* with input arguments A and B, and output arguments C and D:

```
procedure alpha (a, b, c, d)
```

The calling program may not be interested in the return values C and D, and may therefore call *alpha* as follows:

```
call alpha (a, b, junk, junk)
```

Since the SPP language passes arguments by reference, this call maps the two dummy arguments C and D to the same physical storage location. If C and D are used as distinct local variables within *alpha* (presumably in an effort to save storage), a subtle computation error will almost certainly result, which may be quite difficult to diagnose.

4.8. avoid assumptions about the machine precision

The variation of numeric precision amongst machines by different manufacturers is a well known problem affecting the portability of software. This problem is especially important in numeric software, where the accumulation of errors may be critically important. The SPP language addresses the problem of machine precision by providing both single and double

precision integer and floating point data types, and by defining a minimum precision for each.

To produce a transportable program, one must select datatypes based on the minimum precisions given in the table below. The actual precision provided by the software development machine may greatly exceed these values, but a program must not take advantage of such excess precision if it is to be transportable. In particular, a long integer should be used whenever a high precision integer is required, and care should be taken to avoid large floating point exponents.

Minimum Precision of Selected SPP Datatypes

datatype	precision
char	+/- 127 (8 bit signed)
short	+/- 32767 (16 bit signed)
int	+/- 32767 (16 bit signed)
long	+/- 2147483647 (32 bit signed)
real	6 decimal digits, exponent +/- 38
double	14 decimal digits, exponent +/- 38

4.9. do not compare floating point numbers for equality

In general, it is very difficult to reliably compare floating point numbers for equality. The result of such a comparison is not only machine dependent, it is context dependent as well. The only possible exception is when numbers are compared which have only been copied in an assignment statement, without any form of type coercion or other transformations.

```
real    x

begin
        x = 1.0D10
        if (x == 1.0D10)
            ...
end
```

The code fragment shown above, simple though it is, is machine dependent because the double precision constant has been coerced to type real and back to double by the time the comparison takes place. Comparisons of just this sort are possible in IRAF programs which flag bad pixels with the magic value **INDEF**. Avoid type coercion of indefinites; use INDEF or INDEFR only for type real pixels, INDEFD for type double pixels, and so on.

Occasionally it is necessary to determine if two floating point numbers are equivalent to within the machine precision. The predefined machine dependent constants **EPSILON** and **EPSILOND** are provided in the SPP language to facilitate such comparisons. The two single precision floating point numbers *x* and *y* are said to be equivalent to within the machine precision, *provided the quantities* x *and* y *are normalized to the range one to ten prior to comparison*, if the following relation holds:

abs (x – y) < EPSILON

4.10. use the standard predefined machine constants

A number of obviously machine dependent constants are predefined in the SPP language. These include such commonly used values as EPSILON, INDEF, SZB_CHAR, and so on. Other less commonly used machine constants, such as the maximum number of open files (LAST_FD), are defined in the system include file <*config.h*>. Device dependent parameters

such as the block or sector size for a disk device are not necessarily unique within a system, and are therefore not predefined constants. A run time call is required to obtain the value of such device dependent parameters.

A complete list of the standard predefined machine dependent constants is shown below. Some of these are difficult to use in a transportable fashion. The transportability of a program is greatest when no machine dependent parameters are used, be they formally parameterized or not.

Machine Dependent Constants

name	*datatype*	*meaning*
BYTE_SWAP	i	swap magtape bytes?
EPSILON	r	single precision machine epsilon
EPSILOND	d	double precision machine epsilon
INDEF	r	indefinite pixel of type real
INDEF*t*	*t*	indefinite valued pixels
MAX_DIGITS	i	max digits in a number
MAX_EXPONENT	i	largest floating point exponent
MAX_INT	i	largest positive integer
MAX_LONG	l	largest positive long integer
MAX_REAL	r	largest floating point number
MAX_SHORT	i	largest short integer
NBITS_INT	i	number of bits in an integer
NBITS_SHORT	i	number of bits in a short integer
NDIGITS_DP	i	number of digits of precision (double)
NDIGITS_RP	i	number of digits of real precision
SZB_ADDR	i	machine bytes per address increment
SZB_CHAR	i	machine bytes per char
SZ_FNAME	i	max chars in a file name
SZ_LINE	i	max chars in a line
SZ_PATHNAME	i	max chars in OS dependent file names
SZ_VMPAGE	i	page size, chars (1 if no virtual mem.)
SZ_*type*	i	sizes of the primitive types
WORD_SWAP	i	swap magtape words?

4.11. explicitly initialize variables

Storage is statically allocated for all local and global variables in the SPP language. Unless explicitly initialized, the initial value of a variable is *undefined*. Although many compilers implicitly initialize variables with the value zero, this fact is quite machine dependent and should not be depended upon. Local variables should be explicitly initialized in an assignment or **data** statement before use.

Global variables (in common blocks) cannot be initialized with the **data** statement. Some compilers permit such initialization, but this feature is again quite machine dependent, and should not be depended upon. Global variables must be initialized by a run time initialization procedure.

4.12. beware of functions with side effects

The order of evaluation of an expression is not defined. In particular, the compiler may evaluate the components of a boolean expression in any order, and parts of a boolean expression may not be evaluated at all if the value of the expression can be determined by what has already been evaluated. This fact can cause subtle, potentially machine dependent problems when a boolean expression calls a function with side effects. To see why this is a problem, consider the

following example:

```
if (flag || getc (fd, ch) == EOF)
    ...
```

The function *getc* used in the example above has two side effects: it sets the value of the external variable *ch*, and it advances the i/o pointer for file *fd* by one character. If the value of *flag* in the **if** statement is true, the value of the boolean expression is necessarily true, and the compiler is permitted to generate code which would skip the call to *getc*. Whether or not *getc* gets called during the evaluation of this expression depends on how clever the compiler is (which cannot be predicted), and on the run-time value of the variable *flag*.

4.13. use of intrinsic functions

The intrinsic functions are generic functions, meaning that the same function name may be used regardless of the datatype of the arguments. Unlike ordinary external functions and local variables, *intrinsic functions should not be declared.* Not all compilers ignore intrinsic function declarations.

Only the intrinsic functions shown in the table below should be used in SPP programs. Although current compilers for the SPP language will accept many Fortran intrinsic functions other than those shown, the use of such functions is nonstandard, and will not be supported by future compilers.

Standard SPP Intrinsic Functions						
abs	atan	conjg	exp	long	nint	sinh
acos	atan2	cos	int	max	real	sqrt
aimag	char	cosh	log	min	short	tan
asin	complex	double	log10	mod	sin	tanh

Note that the names of the type coercion functions (**char**, **short**, **int**, **real**, etc.) are the same as the names of the datatypes in declarations. The functions **log10**, **tan**, and the hyperbolic functions, may not be called with complex arguments.

4.14. explicitly align objects in global common

Care should be taken to align objects in common blocks on word boundaries. Since the size of a word is machine dependent, this is not always easy to do. Common blocks which contain only objects of type integer and real are the most portable. Avoid booleans in common blocks; use integer variables with the values YES and NO instead. Objects of type char and short should be grouped together, preferably at the end of the common block, with the total size of the group being an even number. Remember that the SPP compiler allocates one extra character of storage for character arrays; character arrays should therefore be odd-dimensioned.

5. Software Documentation

Even the best software system is of no value unless people use it. Given several software packages of roughly similar capabilities, people are most likely to use the package which is easiest to understand, i.e., which has the simplest interface, and which is best documented. Documentation is perhaps the single most important part of the user interface to a system, and to a large extent the quality of the documentation for a system will determine what judgment people make of the quality of the system itself.

The documentation associated with a large software system (or applications package) can be classed as either user documentation or system documentation. User documentation describes the function of the modules making up the system, without reference to the details of

how the modules are implemented. System documentation includes design documentation, documentation describing the details of how the software is implemented, and documentation describing how to install and test the system.

5.1. User Documentation

The first contact a user has with a system is usually provided by the user documentation for the system. Good user documentation should provide an accurate and concise introduction to the system; it should not emphasize the glamorous system features or otherwise try to "sell" the system. It should not be necessary for the user to read all the documentation to be able to make simple use of the system. The documentation should be structured in such a way that the user may read it to the level of detail appropriate to his or her needs. Good user documentation is characterized by its conciseness and clarity, not by the sheer volume of documentation provided.

In what follows, we use the terms "system", "subsystem", and "package" interchangeably. The term "function" refers both to CL callable tasks and to library procedures. The term "user" refers both to end users and to programmers, depending on the nature of the system or package to be documented. The term "document" need not refer to separately bound documents; whether separate documents or multiple sections within a single document are produced depends upon the size of the system and upon the number of authors.

The user documentation for a large system or package should consist of at least the following documents:

(1) The **User's Guide**, which introduces the user to the system, and provides a good overall summary of the facilities provided by the system. This document should provide just enough information to tell the first time user how to exercise the most commonly used functions. Great care should be taken to produce a highly readable document, minimizing technical jargon without sacrificing clarity and conciseness. Plenty of figures, tables, and examples should be included to enhance readability.

(2) The **Reference Manual**, which describes in detail the facilities available to the user, and how to use these facilities. The reference manual is the definitive document for the system. It should be complete and accurate; technical terms and formal notations may be used for maximum precision and clarity. The reference manual defines the *user interface* to the system; implementation details do not belong here.

The minimum reference manual consists of a set of so-called **manual pages**, each of which describes in detail one of the functions provided by the system. The manual pages should be available both on-line and in printed form. The printed reference manual should contain any additional information which pertains to more than one function, and which therefore does not belong in a manual page, but which is too technical or detailed for the user's guide.

Other user documentation might include a report of the results of any tests of the system, as when an a scientific analysis package is tested with artificial data. An objective evaluation of strengths and shortcomings of the algorithms used by the package might be useful. It is important that both the user and the implementor understand the limitations of the software, and its intended range of application.

5.2. System Documentation

System documentation is required to produce, maintain, test, and install software systems. The main requirement for system documentation is that it be accurate; it need not be especially well written, is usually quite technical, and need not be carefully typeset nor printed. The system documentation for a package should be maintained in files in the source directories for the package which it describes.

The system documentation for a large system or package should include the following documents:

(1) The requirements for the system.

(2) The detailed technical specifications for the system.

(3) For each program in the system, a description of how that program is decomposed into modules (i.e., a structure chart), and the function of each module.

(4) Implementation details, including descriptions of the major data structures and details of their usage, descriptions of complicated algorithms, important strategies and design decisions, and notes on any code that might be hard for another programmer to understand. This need not extend to describing program actions which are already documented using comments in the code.

(5) A test plan, describing what verification software is available, how to use it, and how to interpret the results. The amount of documentation required should be minimized by automating the verification software as much as possible.

(6) Instructions on how to install the system when it is ported to a new computer. List any include files which may need to be edited, directories required by the system which may have to be created, libraries or other program modules external to the package which are required, and any file or device names which may have to be changed. A description of how to compile each program should be included; a UNIX *Makefile* for the package would be ideal.

(7) A revision history for the software, giving the names of the original authors, the dates of the first release and of all subsequent revisions, and a summary of the changes made in each release of the system. Any bugs, restrictions, or planned improvements should be noted.

These documents are listed more or less in the order in which they would be produced. The requirements and specifications of a system are written during the preliminary design phase. Documentation describing the decomposition of programs into modules, and detailing the data structures and algorithms used by the package is written during the detailed design stage. After the code has been written and tested, additional notes on the details of the implementation should be made, and the original design documentation should be brought up to date. The remaining documentation should be produced after implementation, before the package is first released.

5.3. Documentation Standards

All documentation should be maintained in computer readable form on the software development machine. The standard text processing software for IRAF user documentation is the UNIX *Troff* text formatter, used with the *ms* macros, the *Tbl* table preprocessor, the *Eqn* preprocessor for mathematical graphics, and so on. Associated utilities such as *Spell* and *Diction* are useful for detecting spelling errors and bad grammatical constructs. User documentation will be typeset and reproduced in quantity by the KPNO print shop.

The standard text processing software for all on-line manual pages and system documentation is the *Lroff* text formatter, a portable IRAF system utility. The UNIX utilities cannot be used for on-line documentation, and should not be used for system documentation because it is difficult to justify the expense of typesetting system documentation, and because system documentation is not maintained in printed form, and many users will not have access to the UNIX text processing tools. The Lroff text processor is more than adequate for most system documentation.

The format of user documentation should be similar to that used in this document, i.e.:

(1) The title page should come first, consisting of the title, the names of the authors and of their home institutions, an abstract summarizing the contents of the document, and the date of the first release of the document, and of the current revision.

(2) A table of contents for the document should be given next, except possibly in the case of very small documents.

(3) Next should come the introduction, followed by the body of the document, organized into sections numbered as in this document.

(4) Any references, appendices, large examples, or the index or glossary if any, should be given last.

Lroff and Troff format source files should have the extensions ".hlp" and ".ms", respectively. *All documentation for a package should be maintained in the source directories for the package*, to ensure that the documentation gets distributed with the package, does not get lost, can easily be found, and to make it easier for the programmer to keep the documentation up to date.

5.4. Technical Writing

Technical writing is a craft comparable in difficulty to computer programming. Writing good documentation is not easy, nor is it a single stage process. Documents must be designed, written, read, criticized, and then rewritten until a satisfactory document is produced. The process has much in common with programming; first one should establish the requirements or scope of the document, then one should prepare an initial outline (design), which is successively refined until it is detailed enough to fully define the contents of the final document. Writing should not begin until one has structured the document into a hierarchy of sections, each of which is well named, and each of which documents a single topic.

English is not a formal language, like a computer language, and it is accordingly very difficult to define a standard style for technical prose. A discussion of writing style in general is given in the excellent little book by Strunk and White [14]. Technical writing differs from other writing in that the material should be clearly and logically organized into sections, and graphics, i.e., lists, tables, figures, examples, etc., should be liberally used to present the material. Large, monolithic paragraphs, or entire pages containing only paragraphs of text, appear forbidding to the reader and should be avoided.

The following guidelines for writing style in technical documents are reproduced from reference [8], *Software Engineering* by I. Sommerville:

(1) Use active rather than passive tenses when writing instruction manuals.

(2) Do not use long sentences which present a number of different facts. It is much better to use a number of shorter sentences.

(3) Do not refer to previously presented information by some reference number on its own. Instead, give the reference number and remind the reader what the reference covered.

(4) Itemize facts wherever possible rather than present them in the form of a sentence.

(5) If a description is complex, repeat yourself, presenting two or more differently phrased descriptions of the same thing. If the reader fails to completely understand one description, he may benefit from having the same thing said in a different way.

(6) Don’t be verbose. If you can say something in 5 words do so, rather than use ten words so that the description might seem more profound. There is no merit in quantity of documentation — quality is much more important.

(7) Be precise and, if necessary, define the terms which you use. Computing terminology is very fluid and many terms have more than one meaning. Therefore, if such terms (such as module or process) are used, make sure that your definition is clear.

(8) Keep paragraphs short. As a general rule, no paragraph should be made up of more than seven sentences. This is because of short term memory limitations. [Another general rule is that few paragraphs should be longer than seven or eight *lines* on an $8^1/_2$ by 11 inch page.]

(9) Make use of headings and subheadings. Always ensure that a consistent numbering convention is used for these.

(10) Use grammatically correct constructs and spell words correctly. Avoid constructs such as split infinitives.

Technical writing should not be regarded as a chore. The process is difficult and challenging, and can be quite rewarding. Often the act of writing results in new insight for the writer. Writing is a form of judgment; if an idea or design cannot be explained clearly, there is probably something wrong with it. Writing forces one to consider an issue in detail, and often is the source of new ideas. A software system cannot be widely used until it is documented, and the quality of the documentation will do much to ensure the success of the system itself.

References

1. D. C. Wells and E. W. Greisen, *FITS — A Flexible Image Transport System,* Proceedings of the International Workshop on Image Processing in Astronomy, Ed. G.Sedmak, M.Capaccioli, R.J.Allen, Osservatorio Astronomico di Trieste, 1979.
2. E. Allman and M. Stonebreaker, "Observations on the Evolution of a Software System", *Computer*, June 1982.
3. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice - Hall, Inc., Englewood Cliffs, New Jersey, 1978.
4. H. Spencer et al., *Indian Hill C Style and Coding Standards as amended for U of T Zoology UNIX*. An annotated version of the original Indian Hill (Bell Labs) style manual for the C language.
5. D. Tody, *A Reference Manual for the IRAF Subset Preprocessor Language,* KNPO, January 1983.
6. American National Standards Institute, Inc., *American National Standard Programming Language Fortran,* document number ANSI X3.9-1978, April 1978.
7. J. Larmouth, *Fortran 77 Portability*, Software — Practice and Experience, Vol. 11, 1071-1117 (1981).
8. I. Sommerville, *Software Engineering*, Addison-Wesley, 1982.
9. W. P. Stevens, *Using Structured Design*, John Wiley & Sons, Inc., 1981.
10. J. D. Aron, *The Program Development Process; Part II, The Programming Team*, Addison-Wesley, 1983.
11. W. S. Davis, *Tools and Techniques for Structured Systems Analysis and Design*, Addison-Wesley, 1983.
12. B. Meyer, "Principles of Package Design", *Communications of the ACM*, July 1982, Vol. 25, No. 7.
13. G. D. Bergland, "A Guided Tour of Program Design Methodologies," *Computer*, October 1981.
14. W. Strunk Jr. and E. B. White, *The Elements of Style*, Mcmillan Publishing Co., Inc., 1979 (third edition).

Standard Nomenclature

AAS

The American Astronomical Society.

C

C is a powerful modern language for both systems and general programming. C provides data structuring, recursion, automatic storage, a fairly standard set of control constructs, a rich set of operators, and considerable conciseness of expression. Developed by Ken Thompson and Dennis Ritchie at Bell Labs in the early 1970's, C is the language used to implement the kernel of the UNIX operating system, as well as the standard UNIX utilities.

CL

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

The CL is itself both a task and a process. The CL process runs concurrently with the subtasks which it executes. The CL process and the process containing the subtask being executed communicate dynamically via interprocess communication, providing both a highly interactive mode of execution, as well as a batch mode.

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

CLIO

CL I/O. A package of SPP callable library routines, used to communicate with the CL.

FIO

File I/O. A package of SPP callable library routines, used to access files.

FITS

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

FMTIO

Formatted I/O. A package of SPP callable library routines, used to perform formatted i/o (decoding and encoding of character strings).

Fortran

As the most widely used language for scientific computing for the past twenty years, Fortran needs no introduction. Fortran is used in the IRAF system as a sort of "super assembler" language. Programs and procedures written in the IRAF SPP language are mechanically translated into a highly portable subset of Fortran, and the Fortran modules are in

turn translated into object modules by the host resident Fortran compiler. Existing numerical and other modules, already coded in the Fortran language, are easily linked with modules written in the SPP language to produce executable programs. The IRAF system and applications software does not use any Fortran i/o; all i/o facilities are provided by the IRAF **program interface** and **virtual operating system**.

GIO

Graphics I/O. A package of SPP callable library routines, used to interface to graphics and grayscale devices.

IAU

The International Astronomical Union.

IMIO

Image I/O. A package of SPP callable library routines, used to access imagefiles (bulk data arrays).

IRAF

The IRAF or "Image Reduction and Analysis Facility", consists of a virtual operating system, a command language, a general purpose programming language (which was developed especially for IRAF), a large i/o library, a numerical library, and numerous support utilities and scientific applications programs. The system is designed to be transportable to any modern superminicomputer. When completed, the system will provide extensive facilities for general image processing, astronomical data reduction and analysis, scientific programming, and general software development.

Lroff

The Lroff text formatter is part of the portable IRAF system. The online **help** facilities use Lroff, and hence manual pages and other online documentation must be maintained in Lroff form. The Lroff text formatter is patterned after the UNIX *Troff* text formatter.

MTIO

Magnetic Tape I/O. A package of SPP callable library routines, used to read and write magnetic tapes.

Make

Make is a UNIX utility program, used to compile and link (make) programs. Make takes as input a human readable *Makefile* which describes the interdependencies of the modules in the package, as well as giving exact instructions for making each module. When making a target module, Make recompiles only those modules which have been changed since the target was last made. A simple command like "make all" usually suffices to make all of the modules in a package.

Make is most useful on UNIX systems. Even on non-UNIX systems, however, the makefile for the package is useful documentation, for it describes precisely how to make each module in the package.

Mklib

Mklib is a UNIX dependent utility developed for IRAF. Mklib is analogous to Make, except that Mklib is used to maintain libraries. Mklib checks each module in a library to see if it is up to date, and if not, recompiles the module and installs the new object module in the library. Mklib is used by the IRAF Sysgen utility to automatically update the IRAF system (consisting of four libraries containing several hundred modules).

OS

(1) An acronym for the term "operating system". (2) The OS interface package, which contains the machine dependent routines required to interface the portable IRAF i/o packages to the local operating system.

OSFN

An acronym for the term "OS dependent file name".

SPP

The IRAF Subset Preprocessor Language (SPP), implements a subset of the full language scheduled for development in 1984. The SPP language is a general purpose language, patterned after Ratfor and C. The language provides advanced capabilities, modern control constructs, enhanced portability, and support for the IRAF runtime library (CL interface, etc.).

Troff

Troff is the UNIX text formatter. In IRAF documentation, *Troff* is always used in conjunction with the "ms" macro package.

UNIX

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

VFN

An acronym for the term "virtual file name". A virtual file name is a machine independent filename of the form "ldir$root.extn".

VMS

The native operating system for Digital Equipment Corporation's VAX series of supermini computers.

VOPS

The "vector operators" package, a package of SPP callable library routines providing a wide class of vector pseudo-instructions. The VOPS routines are written in the SPP language, but may be optimized in assembler or interfaced directly to an array processor, depending upon the implementation.

band

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

binary file

A binary file is an array or sequence of **chars**, where the term char defines a unit of storage, and implies nothing about the contents of the file. Data is transferred between a binary file and a buffer in the calling program by a simple copy operation, without any form of conversion. Binary files are created, deleted, and accessed via the routines in the FIO interface. Barring device restrictions, binary files may be accessed at random, and extended indefinitely. Almost any device may be accessed as a binary file via FIO.

binary operator

An operator which combines two operands to produce a single result (i.e., the addition operator in "x + y").

brace

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

bracket

The left and right square brackets are the characters "[" and "]". Brackets are used in the SPP language to form array subscripts.

byte

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

char

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

column

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

compiler

A compiler for a language X is a program which translates a **source module** written in the language X into an **object module**. A **linker** subsequently combines a number of object modules to produce an executable **process**.

coupling

Coupling measures the strength of relationships between modules. The independence of modules is maximized when coupling in minimized. A change in one module is least likely to require a change in another module when the two modules are minimally coupled.

data structure

A data structure is an aggregate of two or more data elements. Examples include arrays, descriptors, files, records, linked lists, trees, graphs, and so on.

database management

Database management is a branch of computing science concerned with techniques for implementing, maintaining, and accessing databases. Databases may be used to store arbitrarily complex data objects. A database is self describing and self contained. Access to a database typically occurs only through well defined interfaces, which ideally provide a high degree of **data independence** (the external world knows no more than needed about the contents of the database, or how data is stored in the database).

Applications programs communicate with one another via records passed through the database, as well as save final results in the database. A general purpose query language can be used to inspect and manipulate the contents of a database.

datafile

A datafile is a database storage file. Datafiles are used to store program generated **records** or descriptors, containing the results of the analysis performed by a program. Datafile records may be the final output of a program, or may be used as input to a program.

field

A field is an element of a structure or record. Each field has a name, a datatype, and a value.

function

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

header file

A header file is a file (extension ".h") containing only defined constants, structure definitions, macro definitions, or comments. Header files are included in other files by referencing them in **include** statements, and are not directly compiled.

identifier

An identifier is a sequence of characters used to name a procedure, variable, etc. in a compiled language. In the SPP language, an identifier is an upper or lower case letter, followed by any number (including zero) of upper or lower case letters, digits, or instances of the underscore character.

image

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

imagefile

The form in which images are implemented in the IRAF system. The IRAF currently supports images of up to seven dimensions, in any of eight different datatypes. Only **line storage mode** is currently available. The "imagefile" structure is actually implemented as two separate files, the **image header file** and the **pixel storage file**.

include file

An "**include** <*include_file_name*>" statement in the SPP language is replaced during compilation by the contents of the named include file (the contents of the include file are inserted into the input stream).

interface

The interface to a module is *defined* by the **external specifications** of the module. The *actual* interface to a module is everything that is known about the module by other modules in the system. The interface to a subroutine library, for example, is defined by the manual pages, reference manuals, and other formal documentation for the library.

line

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

list file

A list file is a text file, each line of which is a record containing one or more fields. Each record in the list has the same format, though not all fields need be present (fields can only be omitted from right to left).

macro

A macro, or **inline function**, is a function with zero or more arguments, which is expanded by text substitution during the preprocessing phase of compilation.

newline

The newline character ('\n') delimits each line of text read by the FIO input procedures. If a text file is read character by character, a single newline character marks the end of each line, and the special character EOF marks the end of the file. Newline is logically equivalent to a carriage return followed by a line feed.

operand

An operand is a data object which is operated upon by an operator, procedure, or task. Operands may be either input or output, or both.

package

A package is a set of modules which operate on a specific **abstract datatype**. The modules in a package may be either procedures or tasks. Examples of abstract datatypes include the CL, the file, the imagefile, and so on. Some packages are merely collections of modules which are logically related (i.e., the class of system utilities).

parameter

An externaly supplied argument to a module which directly controls the functioning of the module.

pathname

An absolute OS dependent filename specification, i.e, a filename which is not an offset from the current directory.

pixel

The fundamental unit of storage in an image; a picture element. An image is an array of pixels.

pointer

A pointer is a datum which defines the coordinates of an object in some logical coordinate system. To use a pointer, one must know what type of object the pointer points to, and what coordinate system the pointer references.

portable

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

preprocessor

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

procedure

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

process

An executable partition of memory in the host computer. The host OS initiates a process by copying or mapping an executable file into main memory. In a multitasking, multiuser system, a number of processes will in general be simultaneously resident in main memory, and the processor will execute each in turn, performing many **context switches** each second with the result that all processes appear to be executing simultaneously.

program

A program is a compiled procedure which is called by the CL, via the CL interface. The procedure must be referenced in a **task** statement before it can be accessed by the CL, and must not have any formal arguments. A program communicates with the CL via CLIO. An arbitrary number of programs may be linked to form a single **process**.

program interface

The interface between an applications program and the outside world. The program interface is subdivided into a number of **packages**, each of which has a well defined interface of its own. The specifications of the program interface are summarized in the program interface **crib sheet**.

record

A record is data structure consisting of an arbitrary set of fields, used to pass information between program modules, or to permanently record the results of an analysis program in a **database**. Often, records are organized into arrays, where each record contains the results of the analysis of a particular object.

script task

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

specifications

A detailed description of a software system or subsystem, concentrating on the external attributes of the software rather than the on the implementation. **Requirements** are similar to specifications, but are usually more formal and less detailed. The specifications for a subsystem define the interface to the subsystem, and when written in an informal style may resemble a reference manual.

system interface

The interface between the portable IRAF software and the host operating system. The system interface is a **virtual operating system**. The system interface routines, maintained in the "OS" package, are in principle the only part of a system that needs to be changed when porting the system to a new computer.

task

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

task statement

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

text file

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

unary operator

An operator which operates on a single operand, i.e., the minus sign in the expression "–x", or the boolean complement operator in the expression "!x".

virtual memory

If the address space of a process exceeds the amount of physical memory which the process can directly address, the process is using virtual memory. The virtual address space is organized into a series of fixed size **pages**. The amount of physical memory available to a process is known as the **working set** of a process. Pages which are not **memory resident**, i.e., not in the working set, reside on some form of backing store, usually a disk file. When a page is referenced which is not in the working set, a **page fault** occurs, causing the page to be read into the working set. If the pattern of memory accesses is such that a page fault occurs on nearly every access, the process is said to be **thrashing**, and will run exceedingly slowly.

virtual operating system

A package of system calls, providing a set of primitive functions comparable to those provided by an actual operating system, which can be interfaced to a number of actual operating systems. The IRAF virtual operating system provides routines (the so-called **z-routines**) for file access, process initiation and control, interprocess communication, memory management, magtape i/o, exception handling, logical names, and time and date.

whitespace

A sequence of one or more occurrences of the characters blank or tab.

z-routines

Machine dependent routines, used to interface to the host operating system. The IRAF z-routines are maintained in the package "OS".