SPACE
TELESCOPE
SCIENCE
INSTITUTE

SPP Reference Manual

STSDAS Group
Science Software Branch
Zoltan G. Levay, Editor

Second Edition
October 1992

Operated by the Association of Universities for Research in Astronomy, Inc., for the National Aeronautics and Space Administration

The STSDAS Group

Bob Hanisch	Chief, Science Software Branch
Betty Stobie	STSDAS Programming Supervisor
Dick Shaw	STSDAS Project Scientist
Ray Williamson	System Administrator and Distribution
Dave Bazell	Synthetic Photometry, FOC Calibration and Analysis
Jonathan Eisenhamer	Graphics, Image Display, WCS Support
Phil Hodge	Table System, Fourier Analysis, FOC Calibration and Analysis
J.C. Hsu	HSP, FGS, and WF/PC Calibration and Analysis
Zolt Levay	Graphics, Image Display, Filmwriter, Documentation
Bernie Simon	Calibration Data Base, Table Editor, IRAF System Support
Nelson Zarate	FITS, IRAF System Support, Ports, Benchmarking
Mark Stevens	Technical Writing
Fred Romelfanger	X-Windows-Based User Interface
Jinger Mo	Software Testing, Image Restoration

Send comments or corrections to:
Zoltan G. Levay, SCARS
Space Telescope Science Institute
3700 San Martin Drive
Baltimore, Maryland 21218
E-mail: levay@stsci.edu

Table of Contents

List of Selected Tables

List of Selected Examples

Preface

The Subset Preprocessor Language (SPP) is a programming language designed to develop applications in the IRAF programming environment. This is a reference manual intended to explain the language sufficiently to allow a programmer to develop useful applications. As such, it comprises two fundamental parts. The first is a detailed reference describing the language's features, syntax, and structure. The other is a fairly complete description of the interfaces to the IRAF environment. Separate chapters are devoted to error handling and making IRAF tasks. Four appendixes cover the system defined include files, detailed examples and other helpful hints, and utilities for debugging applications code. Appendix E describest the STSDAS **tables** utilities. Simple examples of specific concepts are scattered throughout this text. These are usually fragments of code intended to illustrate the concept under discussion. However, Appendix B contains a few complete examples.

This is *not* a programming textbook. It is assumed that the reader is conversant with some programming language. Because of the similarity of SPP to Fortran and C, experience with those languages is certainly an asset. It is also assumed that the reader is familiar with IRAF to some extent. That is, that there is some experience with the concepts behind the structure of programs and rationale for the system. In addition, some knowledge of the IRAF command language (cl) is assumed.

Some comments on the syntax in this text may be useful.

- Literal text and reserved keywords to be used in code as-is are set in `typewriter` style to distinguish them from names of objects and real English words. For example, `procedure`, `pointer`, or `maxch` are keywords that may be used in SPP code. Another example is a directory or file name, which, as literal text, would be set in typewriter style: `gio$doc/gio.hlp` or `help cursor`

- When a reserved word ends in an italicized capital *T*, the *T* is a placeholder intended to be replaced by a data type character (see for example, “Arithmetic Operators” on page 104). These data type specifiers include:
  - `x` - Complex
  - `d` - Double
  - `l` - Long
  - `s` - String
  - `c` - Char
- Package names are set in bold face, for example, **cl** or **imio**.
- Generic names for entities replaced by some specific keyword are set in *italic* style, such as a template syntax: `for (`*init*`;` *test*`;` *control*`)` demonstrating the `for` syntax.
- Square brackets used in a template (*[...]*) surround optional text.
- Function names are usually referred to in the text without arguments but with empty parentheses to distinguish them from other identifiers.

SPP is a part of the IRAF application environment. IRAF was developed by the National Optical Astronomy Observatories (NOAO), primarily for the analysis of astronomical data. Doug Tody is primarily responsible for the design and management of the IRAF core system, including SPP. Additional examples of how to develop IRAF applications code can be found in *An Introductory User’s Guide to IRAF SPP Programming* by R. Seaman [Seaman92].

Chapter 1 of this manual is based largely on Doug Tody’s *A Reference Manual for the IRAF Subset Preprocessor* [Tody83]. Chapter 2 draws from the design documents for the various interfaces, and Appendix E is based on earlier document by the STSDAS Group.

CHAPTER 1:

Language Syntax

The SPP language is based on the Ratfor language. Ratfor, in turn, is based on Fortran, with extensions for structured control flow, etc. The lexical form, operators, and control flow constructs are identical to those provided by Ratfor. The major differences are the data types, the form of a procedure, the addition of inline strings and character constants, the use of square brackets for arrays, and the `task` statement. In addition, the SPP I/O facilities provided are quite different and are tailored to the IRAF environment. The syntax of the SPP language is fairly straightforward and fundamentally similar to most other high-level languages. While it is based on the Ratfor language, there are elements of C as well as elements of Fortran. SPP is a preprocessed language. That is, there is no SPP compiler per se, but it is translated into another compilable language. In fact, SPP is first translated into Ratfor, which is processed into Fortran. The xc compiler performs all preprocessing, compilation, and linkage. This chapter describes the language in detail. Chapter 2 describes the procedure libraries available to connect a program to the outside world, Chapter 4 describes how to compile an application as well as how it fits into the IRAF environment. Appendix B presents some basic examples and hints for writing real software.

Lexical Form

An SPP program consists of a sequence of lines of text. The length of a line is arbitrary, but SPP is guaranteed to be able to handle only lines of up to 160 characters long. The end of each line is marked by a "newline" character.

Character set

SPP uses the extended ASCII character set which includes the characters listed in Table 1.1

Characters	***Type***
`a-z`	All lower case letters
`A-Z`	All upper case letters
`0-9`	All digits
`# _` &, etc.	Special characters
[tab], [space]	White space

Table 1.1: SPP Character Set.

Some of these may be used in identifier names and numeric constants. The remaining ones have specific meaning within the language. SPP does *not* distinguish between lower case and upper case except for literal strings (inside double quotes). Any character may be used in a literal string. The specific meaning of special characters is described in the appropriate section.

White Space

White space is defined as one or more tabs or spaces. A newline normally marks the end of a statement, and is not considered to be white space. White space always delimits tokens, the smallest recognized elements of the language. Keywords and operators will not be recognized as such if they contain embedded white space. However, the absolute amount of white space is not relevant and there is no enforced structure of text on the line. Indentation and judicious use of white space greatly improves readability. Note, however, that spaces, including trailing blanks, are significant in literal quoted strings such as text to be written to standard output.

Comments

Comments begin with the # character and end at the end of the line. That is, anything after a # is ignored by the preprocessor until the next end of line. Thus, in-line comments may follow SPP statements.

Continuation

Statements may span several lines. A line that ends with an operator (excluding `/`) or punctuation character (comma or semicolon) is automatically understood to be continued on the following line.

Constants

SPP supports several types of constants. These are described below. (Predefined constants are described in Appendix A.)

Integer Constants

A integer constant is a sequence of one or more of the digits in the range `0` through `9`. An octal constant is a sequence of one or more of the digits in the range `0` through `7`, followed by the letter `b` or `B`. A hexadecimal constant is one of the digits in the range `0` through `9`, followed by zero or more of the digits `0` through `9`, the letters in the range `a` through `f`, or the letters `A` through `F`, followed by the letter `x` or `X`. Note that a hexadecimal constant must begin with a decimal digit (zero through nine) to distinguish it from an identifier. The notation shown in Table 1.2 more concisely summarizes these definitions.

Integer Type	*Definition*	*Examples*
Decimal	`[+|-][0-9]+`	`42, -999, 0`
Octal	`[+|-][0-7]+[b|B]`	`42b, 777B`
Hexadecimal	`[+|-][0-9][0-9a-fA-F]*[x|X]`	`0ffx, 0123ABCx`

Table 1.2: Integer Constant Notation.

In the notation used above, `+` means one or more, `*` means zero or more, `–` implies a range, and `|` means “or”. Brackets (`[...]`) define a class of characters. Thus, “`[0–9]+`” reads “one or more of the characters in the range 0 through 9.” An integer constant has the same range as the range of the underlaying Fortran constant. Since this changes from machine to machine, SPP has the predefined constant `MAX_INT` as the maximum allowable integer (see Appendix A).

Floating Point Constants

A floating point constant (type `real` or `double`) consists of a decimal integer, optionally preceded by a sign (`+` or `-`), followed by a decimal point, optionally followed by a decimal fraction, followed by one of the characters: `e`, `E`, `d`, `D`, followed by a decimal integer, which may be negative. Either the decimal integer or the decimal fraction part must be present. The number must contain either the decimal point or the exponent (or both). Embedded white space is not permitted. The following are all legal floating point numbers: `.01`, `100.`, `100.01`, `1E5`, `1e-5`, `-1.00D5`, `1.0d0`. A `complex` constant consists of two floating point constants separated by a comma and enclosed in parentheses representing the real and imaginary parts, `(1.0,0.0)` for example. A floating constant may also be given in sexagesimal, i.e., in hours and minutes, or in hours, minutes, and seconds, or any other units in which places of the number vary by a factor of sixty. Numerical fields are separated by colon characters (`:`) and there must be either two or three fields. The number of decimal digits in the second field and in the integer part of the third field is limited to exactly two. The decimal point and any fraction is optional. The low level procedures that parse input recognize this syntax as well, making it convenient for users to enter values in a natural format (time or equatorial coordinates).

Coordinate	***Floating Point***
`00:01`	0.017
`00:00:01`	0.00028
`01:00:00`	1.0
`01:00:00.00`	1.0
`01:30.7`	1.5116

Table 1.3: Coordinate and Floating Point Equivalents.

The last example has only two fields with the last including a fraction. These two fields are then the largest and next largest fields, such as hours and minutes of time or degrees and minutes of arc. Note that there may be some problems in rounding, however. The predefined constants

MAX_REAL and MAX_DOUBLE contain the host-dependent maximum permissible values for real and double constants, respectively.

Character Constants

A character constant consists of from one to four digits delimited at front and rear by the single quote (′), as opposed to the double quotes used to delimit *string* constants). A character constant is numerically equivalent to the corresponding decimal integer, and may be used wherever an integer constant would be used. On most systems, characters are represented in ASCII, therefore the character values are the ASCII values.

Character Constant	***Decimal Value***	***Interpretation***
‘\007′	7	The integer 7, CTRL G, (BEL)
‘a′	97	The character a
‘\n′	10	The newline character
‘\\′	92	The character \

Table 1.4: Character Constants.

The backslash character (\) is used to form *escape sequences*, which are special non-printed characters. SPP recognizes the following escape sequences:

Escape	***Interpretation***	***Decimal Value***	***Control Sequence***	***ASCII Mnemonic***
\b	Backspace	8	CTRL H	BS
\f	Form feed	12	CTRL L	FF
\n	Newline	10	CTRL J	LF
\r	Carriage return	13	CTRL M	CR
\t	Horizontal tab	9	CTRL I	HT

Table 1.5: Character Constant Escape Sequences.

String Constants

A string constant is a sequence of characters enclosed in double quotes ("), `"image"` for example. The double quote itself may be included in the string by escaping it with a backslash (`"abc\"xyz"`). All of the escape sequences given above are recognized. The backslash character itself must be escaped to be included in the string. A string constant may not span lines of text. For example,

```
call strcpy ("This is a long character string
with an embedded newline.", outstr, SZ_LINE)
```

Would result in the error “Newline while processing string.” However, you may include a newline in a string explicitly with the newline character, for example:

```
call strcpy ("A string\nwith a newline.", outstr, SZ_LINE)
```

Identifiers

An identifier is the name used to refer to a variable or a procedure. Identifiers are constructed of an upper or lower case letter, followed by zero or more upper or lower case letters, digits, or the underscore character. Identifiers may be as long as desired, but only the first five characters and the last character are significant. Identifiers are used for variable names and procedure names, including built-in, intrinsic functions, as well as other language constructs. SPP maps all identifiers to a Fortran identifier that conforms to Fortran 66 standards. That is, they must be six character or fewer and may not include underscores. SPP performs the mapping by first removing underscores and taking up to the first five characters and the last character. If there is a conflict between two SPP identifiers that map to the same Fortran identifier, the last character of the mapped name is replaced with a digit in one of the names. It may be instructive to see the mappings. The mapped SPP and Fortran identifiers are listed as comments in the Fortran output by xc (using the `-f` option) at the end of the translated source. The definition of an identifier may be summarized using the following rules:

```
[a–zA–Z][a–zA–Z_0–9]*
```

See “Constants” on page 3 for an explanation of the syntax of this shorthand. The following example illustrates valid and invalid SPP identifiers:

Valid Identifiers	***Invalid Identifiers***	
`For2next`	`1awhile`	*Starts with numeral, not letter*
`MAX_numbers`	`up&to`	*Contains &, an invalid special character*
`upts`		
`MAX_VALUES`		
`MAX_VARIABLES`		

Figure 1.1: Identifier Syntax.

Note that the last two map to the same Fortran variable. Therefore, if they were in the same source file, SPP would change the mapping of one to make them unique.

The identifiers in Figure 1.2 are reserved. That is, do not use them as variable or procedure names. Note that not all of them are actually used at present.

auto	clgetpar	double	getpix	long	real	struct	virtual
begin	clputpar	else	goto	map	repeat	switch	vstruct
bool	common	end	if	next	return	task	while
break	complex	entry	iferr	plot	scan	true	
call	data	extern	imstruct	printf	short	union	
case	define	false	include	procedure	sizeof	unmap	
char	do	for	int	putpix	static	until	

Figure 1.2: Reserved Identifiers.

Fortran statements

Fortran statements may be used in SPP source by preceding the statement with a percent character, `%`. The `xc` compiler then passes this statement through unchanged. Remember that Fortran *does* require specific positioning of the text on the line, unlike SPP. So you must include the necessary spaces between the `%` escape character and the beginning of the Fortran statement. For example:

```
# Fortran follows, note
# 6 spaces after %
%      INTEGER INTF
```

Also keep in mind that while most SPP data types are the same as Fortran, character strings are not. See “Calling Fortran Subprograms” on page 38 and “Fortran Strings” on page 125 for more details.

Data Types

The subset preprocessor language supports a fairly wide range of data types. The actual mapping of an SPP data type into a Fortran data type depends on what the target compiler has to offer. SPP supports the usual fundamental data types: integer, floating point, complex, boolean, and character. Some of these have more than one subtype, varying by the size of each value. The actual size in bytes of a particular data type depends on the host system. IRAF maintains a structure containing these definitions, available to the applications programmer.

Declaration	***Data Type***	***Fortran Equivalent***
`bool`	Boolean	`LOGICAL`
`char`	Character	short `INTEGER`
`short`	Short integer	short `INTEGER`
`int`	Integer	`INTEGER`
`long`	Long integer	long `INTEGER`
`real`	Single precision floating	`REAL`
`double`	Double precision floating	`DOUBLE PRECISION`
`complex`	Single precision complex	`COMPLEX`
`char[]`	String (character array)	short `INTEGER` array
`pointer`	Pointer to memory	`INTEGER`
`extern`	External function	`EXTERNAL`

Table 1.6: Data Types.

Note that the size of the variable depends on its hardware implementation which in turn depends on the combination of the Fortran compiler and the host operating system. For example, in VAX Fortran, short integers are implemented as `INTEGER*2`, including `char` and strings (`char` arrays), and long integers are implemented as `INTEGER*4`, which is the same size (four bytes) as `INTEGER`, by default. In addition to the seven primitive data types, the SPP language provides the abstract type pointer. The SPP language makes no distinction between pointers to different types of objects, unlike more strongly typed languages such as C. The `extern` type is also available to declare a function as a variable, as in the Fortran `EXTERNAL` statement.

Integer

SPP has three signed integer data types. There is no byte or unsigned integer data type.

- ***short*** - The smallest integer type, usually two bytes.
- ***int*** - A signed integer having the size of the fundamental host system word size, usually 32 bits or four bytes. This is equivalent to the Fortran `INTEGER` declaration.
- ***long*** - The largest integer type, usually the same as `int`.

Character

The `char` data type belongs to the family of integer data types, i.e., a `char` variable or array behaves like an integer variable or array. The `char` and `short` data types are signed integers (i.e., they may take on negative values).

String

A string is an array of type `char` terminated by an end of string character (`EOS`). Strings may contain only character data (values 0 through 127 decimal), and must be delimited by `EOS`. A character string may be declared in either of two ways, depending on whether initialization is desired:

```
char    input_file[SZ_FNAME]
string  legal_codes "efgdox"
char    x[15]
```

The preprocessor automatically adds one to the declared array size, to allow space for the `EOS` marker. However, the space used by the `EOS` marker is not considered part of the string. Thus, the `char` array `x[15]` will contain 16 elements, space for up to 15 characters, plus the `EOS` marker.

It is probably a good idea to use an *odd* number for the string size declaration so that the resulting array contains an even number of elements. This permits alignment of strings on *long word* boundaries[1]. Since `char` is implemented as Fortran `INTEGER`, whose size is usually four bytes, sometimes referred to as a long word. Access to memory is usually more efficient if the variables are placed matching the addressable pieces

Note that the string value need not fill the declared size. The `EOS` character signals the end of the string. This is in contrast to Fortran strings, which do not include a terminator character and thus have an implicit size equal to the declared size and are padded with trailing blanks to the string length. Rather, SPP strings are practically identical to the concept of strings in C. Therefore, it is not possible to call a Fortran subroutine directly that expects a string in the calling sequence. However, there are procedures that convert between SPP and Fortran strings. (See "Calling Fortran Subprograms" on page 38). Note that in most procedures that take a string argument, there is also an argument that specifies the maximum string size. See Chapter 2 for specific library procedures.

Floating point

Floating point variables may be single precision (`real`), double precision (`double`), or complex (`complex`) and behave as the equivalent Fortran floating point variables.

- ***real*** - A single precision value equivalent to the Fortran `REAL` data type.
- ***double*** - A double precision floating point value, equivalent to the Fortran `DOUBLE PRECISION` data type.
- ***complex*** - A pair of single precision floating point values equivalent to the Fortran `COMPLEX` data type.

Boolean

The only permissible values for a boolean variable are `true` and `false`. They are used as flag variables or used in test expressions of constructs such as `if` and `while`. Note the distinction between *boolean* variables and the *integer* constant parameters `YES` and `NO`; the latter are sometimes used as flags.

---

1. A glossary of terms appears on page 237.

Pointer

Pointers are used to reference dynamically allocated memory. See "Memory Allocation — memio" on page 53 for a more complete discussion of dynamically allocated memory. More abstractly, pointers may be used to reference "structures," allocated memory with a particular arrangement of variables of differing data types and having a specific structure in memory.

Declarations

All SPP variables must be declared. This includes scalars and arrays, as well as functions. All declarations must precede the body of the procedure. That is, they must be between the `procedure` statement and the `begin` statement. Although the language does not *require* that procedure arguments be declared *before* local variables and functions, it is customary and a good practice. The syntax of a type declaration is the same for parameters, variables, and procedures.

```
type_spec object [, object [,... ]]
```

Here, *type_spec* may be any of the seven fundamental data types, a derived type such as `pointer`, or `extern`. A list of one or more data objects follows. An *object* may be a variable, array, or procedure. The declaration for each type of object has a unique syntax, as follows:

```
procedure identifier()
variable identifier
array identifier[dimension_list]
```

Note that all declaration statements *must* begin at the first character of the line. That is, there may be no white space between the beginning of the line and the beginning of the declaration.

Scalar Variables

Scalar variables are declared with the data type statements and the name of the variable. For example:

```
int     rows            # Number of rows
int     cols            # Number of columns
real    x, y            # Coordinates
bool    verbose         # Print verbose output?
```

Customarily, most variables are described by an in-line comment.

Arrays

Arrays are declared similarly to scalars, with the array size appended to the variable name and enclosed in square brackets (`[` and `]`). The sizes of each dimension are separated by commas within the brackets.

type_spec object[*dim[,dim,...]*]

Note that here the outer square brackets are required, the inner ones represent optional multiple dimensions. Arrays may be up to seven dimensions and are one-indexed by default. That is, the first element is numbered one. Multiply dimensioned arrays are ordered such that the leftmost dimensions vary the fastest, as they are Fortran arrays. Arrays are referenced using the variable name with the element number(s) in *square brackets* (`[]`). As many dimensions must be used in the reference as in the declaration. It is not permitted to address an array outside its declared scope, but is not detected by the compiler. The following examples illustrate how to declare subscripted variables in SPP:

```
int     ivec[100]           # An integer vector with 100 elements
char    line[SZ_LINE]       # A line of text
real    image[100,100]      # Image buffer
```

Example 1.1: Declaring Subscripted Variables.

The last example declares `image` to be 100 by 100 elements in size. The first element would be specified as `image[1,1]`, followed by `image[2,1]`, `image[3,1]`, ... `image[1,2]`, `image[2,2]`, ... `image[100,100]`. The size of each dimension of an array may be specified by any compile time constant expression, or by an integer parameter or parameters, if the array is a formal parameter to the procedure. If the array is declared as a formal procedure argument and the size of the highest (rightmost, or most slowly varying) dimension is unknown, the size of that dimension should be given as `ARB` (for arbitrary). The declared dimensionality of an array passed as a formal parameter to a procedure may be less than or equal to the actual dimensionality of the array. For example, the following example declares several arrays and uses some of them as arguments to functions.

```
define  SZ_DATA 1024
define  ISIZE   100
.
.
real    data[SZ_DATA]         # 1-D array, SZ_DATA defined above
integer intarr[ISIZE,ISIZE]   # 2-D
short   3darray[10,20,30]     # 3-D
.
.
        call myfunc (data, intarr, 3darray, 10, 20)
.
.
procedure myfunc (data, intarr, 3darray, i, j)
real    data[ARB]             # Length of array is unknown
integer intarr[ARB]           # Referenced as 1-D
short   3darray[i,j,ARB]      # 3-D, dimensions passed as arguments
int     i, j                  # Array dimensions
```

Example 1.2: Declaring Arrays and Using as Arguments to Functions.

Note that the `integer` array `intarr` is declared as two-dimensional but referenced in the procedure as one-dimensional. The `short` array `3darray` is declared as three-dimensional in both the calling and called procedure. However, in the called procedure, the last dimension is declared as ARB, while the others are declared with passed arguments. The lower dimensions must be declared explicitly in order for the function to compute the index of the elements. It is highly recommended to use `define`d (macro) constants instead of absolute constants to declare array sizes. This makes maintenance much easier in that the value is declared only once. If the constant is defined outside of a procedure, then any procedure in the same file may access the same constant, eliminating the need to pass a dimension to the functions. In addition, if the constants are defined in an `include` file they are available to procedures in more than one file.

Functions

External functions, whether supplied by the programmer or part of a library package must be declared in a manner similar to variables. This does *not* include intrinsic functions such as `sin()`, `abs()`, etc. (see “Intrinsic Functions” on page 36). Functions may be declared to be any valid SPP data type. For example, if the program includes a `real` valued function named `myfunc`, its declaration and invocation might appear as in Example 1.3.

```
real   rval, x, y, z
real   myfunc()          # Local function
       .
       .
       rval = myfunc (x, y, z)
```

Example 1.3: Invoking External Functions.

External Functions

The `extern` data type declares a variable as a function. The name of the function may then be passed as an actual argument in a procedure call. In the formal procedure (dummy) arguments, the same argument must also be declared `extern`.

```
extern tick()        # Declare tick() as an external function
begin
       # Call axistick using function tick()
       call axistick (igs, ..., tick)

       # Call axistick using function ticklabel()
       call axistick (igs, ..., ticklabel)
end

procedure axistick (igs, ..., func)
pointer igs
.
.
extern func()      # Declare the passed function external
begin
     .
     .
end
```

Example 1.4: Declaring and Using the `extern` Data Type.

Common

Global common provides a means for sharing data between separately compiled procedures. The `common` statement is a declaration, and must be used only in the declarations section of a procedure. Each procedure referencing the same common must declare that common in the same way.

`common` */identifier/ object [, object [, ...]]*

For example,

```
common  /vfnxtn/ nextn, iraf, os, map
```

To avoid the possibility of two procedures declaring the same common area differently in separate procedures, the `common` declaration should be placed in an `include` file (see “Include Files” on page 39). This permits considerably more reliable and easy maintenance, avoiding changes in one procedure without changing another.

Initialization

The `data` Statement

Local variables, arrays, and character strings may be initialized at compile time with the `data` statement. Data in a global common may *not* be initialized at compile time. If initialization of data in a global common is required, it must be done at run time by an initialization procedure. The syntax of the `data` statement is defined identically to the standard Fortran 77 `DATA` statement. Some simple examples follow.

```
real     x, y[2]
char     ch[2]
data     x/0/, y/1.0,2.0/, ch/’a’,’b’,EOS/
```

Any `data` statements must follow all declarations. Note that variables initialized by `data` are *not* guaranteed to have that value except the first time the task is executed from the cl. IRAF tasks executed from the cl may be *cached* or stored in the process cache. That is, they are not restarted from the main procedure except the first time they are executed and after the process cache is flushed (using the cl task `flprcache`). Therefore, a variable modified in a task procedure will not have the initialized value the next time the task is executed, but will have the modified value. It is always safer to initialize variables with macro symbolic constant `define` statements or explicit assignment statements.

The `string` Statement

Character strings may be declared and initialized with the `string` statement. This consists of the keyword `string` followed by the identifier name, followed by the initialization value enclosed in double quotes. Not that there is no explicit string size. A `char` array is implicitly declared the size of the initialization string.

```
string  errmsg "Could not open input"
```

Macro Definitions

An SPP *macro* assigns a symbol or identifier to arbitrary text, implementing *string substitution*. This enables any piece of code to be hidden by using its defined symbol rather than the text itself. Upon precompilation, the macro symbol is replaced by its assigned text. The primary uses of macros are to define *symbolic constants* such as mathematical constants, whose value will not change at run time, implementing in-line or *statement functions*, and for creating *data structures*. Macro definitions allow hiding certain information and can do much to enhance the ease of modifying and maintaining a program. By convention, the names of macros are upper case, to distinguish the names from variables, functions, and other identifiers and to make it clear that a macro is being used. Macros are created by using the `define` command. If the macro is defined after the `procedure` statement, it must be defined before the `begin` statement, and only that procedure may use it. That is, its scope is within a single procedure. If a macro is defined before the `procedure` statement, it is available to any procedure in the source file. Macros that are shared by several procedures should be defined in an include file, particularly if the source is in different files (see “Include Files” on page 39).

Macros may or may not have arguments. An argument is declared in a macro definition by using a dollar character (`$`) and a numeral indicating the argument number. In the macro invocation, arguments are passed in parentheses, `()`. Multiple arguments are separated by commas. Macros without arguments are used primarily to turn explicit constants into symbolic parameters. Examples are shown throughout this text. Macros with arguments are used as statement functions and data structure elements.

Macros incorporating expressions should be enclosed in parentheses to ensure that the expression is executed with the intended precedence. Macro definitions may not include string constants. You may use the `string` statement to declare string constants. All other types of constants, constant expressions, array and procedure references, are allowed, however. The domain of definition of a macro extends from the line following the macro, to the end of the file (except for include files). Macros may be recursive and may be redefined, resulting in no mention by the compiler.

Macro definitions are frequently shared among procedures in several source files by putting them in an *include file*. This is another source file, but has the extension `.h` and is included in any source by using the `include` statement (see “Include Files” on page 39). There are many examples of macro definitions and structures using them in the IRAF sources, both the system code as well as the applications. Look in the `lib$` and `hlib$` directories for the include files for the IRAF system. In addition, each applications package usually contains one or more header include files containing numerous examples.

Symbolic Constants

Constants may be declared as variables, initialized with an assignment statement or by using a `data` statement. Alternately, a symbolic constant may be declared as a macro, using a `define` statement. Each time the macro is used in the code, its name is replaced by the text specified in the `define` statement when the code is compiled. There is no data storage allocated nor an assignment executed at run time. It becomes easy to change the values of constants by changing it once in the `define` statement rather than throughout the code. The meaning of the code frequently becomes clearer by referring to constants by name (`PI`) rather than by value (`3.14159`). There are many constants defined automatically as well as several `include` files available defining many frequently used constants. See Appendix A for a description of these. The following example illustrates the use of macros as symbolic constants:

```
# Use predefined math constants
include <math.h>
define DATA_SIZE   1024
define R_ZERO      0.415
.
.
procedure myproc()
real  ref
real  data[DATA_SIZE]   # Locally defined constant
char  errtxt[SZ_LINE]   # Predefined constant
.
.
begin
     .
     .
     # Uses PI, defined in <math.h>
     ref = PI * R_ZERO
     Assign string, size uses predefined constant
     call strcpy (“End of File”, errtxt, SZ_LINE)
     .
     .
end
```

Example 1.5: Using Symbolic Constants.

Data Structures

A data structure allows a set of variables to be treated as a group. These may include variables of different data types, arrays, strings, pointers, etc. See “Data Structures” on page 58 for more details and additional examples.

```
# Symbolic constant
define LINEAR 1
.
.
# Define the structure (array)
define I_TYPE $1[1]
define I_NPIX $1[2]
.
.
define I_COEFF $1[10]

procedure do_coeff (val1, ...)

# DO_COEFF -- This procedure uses the elements of the coeff array,
# referencing them by their symbolic names, via macros defined
# above.
int   val1
int   other_val
.
begin
     .
     .
     if (I_TYPE(coeff) == LINEAR) {
         I_NPIX(coeff) = val1
         I_COEFF(coeff) = 2
     } else {
         I_NPIX(coeff) = other_val
         I_COEFF(coeff) = 3
    }
    .
    .
end
```

Example 1.6: Using Data Structures.

In this example the macros define a simple structure that permits a different way of using an array. Instead of accessing the array by numeric element numbers, it permits a different name to be defined for each array element that may contain inherently different entities. The array `coeff[]` is redefined as a simple structure containing the fields `I_TYPE`, `I_NPIX`, ..., and `I_COEFF`. Defining a structure enhances the readability of a program by permitting reference to the fields of the structure by name, rather than the array element (`coeff[2]`), and furthermore makes it easier to modify the structure. The same code could be written without using macros, referencing `coeff` as elements of the array or declaring the equivalent elements as separate variables. Note that parentheses are used to refer to elements of the structure, as opposed to square brackets, which refer to array elements. The equivalent implementation without using macros would use an array and reference the elements of the array by their number. This simple example is straightforward. However, for a complicated example, it is usually much clearer to refer to disparate entities by name rather than by an array element.

```
procedure do_coeff (val1, ...)
int    val1
int    other_val
.
.
int    coeff[10]     #Array to contain a “structure”
begin
     .
     .
     if (coeff[1] == 1) {
         coeff[2] = val1
         coeff[10] = 2
     } else {
         coeff[2] = other_val
         coeff[10] = 2
     }
     .
     .
end
```

Example 1.7: Implementing Example 1.6 with Array Elements.

The same result may be accomplished by using a common block, as is shown in the next example.

```
# Symbolic constant
define LINEAR 1
procedure do_coeff (val1, ...)
real    val1
int     other_val
int     i_type
int     i_npix
int     i_coeff
common /coeffs/ i_type, i_npix, ..., i_coeff
begin
     .
     .
     if (i_type == LINEAR) {
         i_npix = val1
         i_coeff = 2
     } else {
         i_npix = other_val
         i_coeff = 3
     }
     .
     .
end
```

Example 1.8: Implementing Example 1.6 with Common Blocks.

Of course, any other procedure using the variables in the common block would have to declare it identically. If you do use `common`, put it and the associated variable declarations in an *include* file so there is only one place the declarations needs to be modified. It is possible to define a structure containing any data type. The types `int`, `real`, `bool`, and `pointer` are guaranteed to be the same length, a single word in memory. A common method of declaring a structure is to use dynamically allocated memory, referring to the structure elements using the `Mem[]` syntax (see "Memory Allocation — memio" on page 53). In this case, you need not explicitly specify a different offset for each data type. For types which may differ in size, however, you must be able to refer to the correct offset and size of a particular structure element. This applies to `short`, `long`, `double`, `complex`, and particularly to `char` and elements treated as arrays. Note that these should be aligned on *long word boundaries*[2]. The convention is to declare the variables in the order of longest first to shortest last, with character strings declared last. There are system defined macros for aiding in the conversion of pointers to these data types:

Macro	***Converts to Type***
`P2X`	`complex`
`P2D`	`double`
`P2L`	`long`
`P2S`	`short`
`P2C`	`char`

Table 1.7: System Macros for Converting Pointers.

The `P2`*T* macros permit you to address the next *structure element* without worrying too much about the word size. These are defined in `hlib$iraf.h` since they depend on the host architecture. The following example declares a structure containing several different data types and some constants. The difference between this and the previous example is that the memory containing the structure is allocated dynamically instead of using a statically allocated array. This additionally permits multiple instances of the structure to be defined. This is the way many packages

2. The size of a long word is machine dependent, but by correctly using structures in SPP you will avoid these difficulties.

handle internal parameters. For example, each time an image is opened using `immap()`, a structure is allocated containing parameters pertaining to the image. Multiple images may be opened, each having associated parameters organized using the same structure.

```
define LEN_MYSTR 128                        # Size of structure
define XVAL      Memx[P2X($1)]              # complex
define DVAL      Memd[P2D($1+2)]            # double
define LVAL      Meml[P2L($1+4)]            # long
define RVAL      Memr[($1+6)]               # real (no P2R)
define IVAL      Memi[($1+7)]               # int (no P2I)
define PVAL      Memi[$1+8)]                # pointer (same as int)
define LENARR    10
define IARRAY    Memi[($1+8+$2)]            # 10 element int array

# Offset the next field by the size of the array
define SVAL      Mems[P2S($1+8+LENARR+1)] # short
define CVAL      Memc[P2C($1+8+LENARR+2)] # Single char
define LEN_CS    64
define CSVAL     Memc[P2C($1+8+LENARR+3)] # Character string

# The next field must be offset by the size of the string
```

Example 1.9: Structure Elements Defined in `myincl.h`.

Note that even though the `P2`*T* macros take care of the offsets into the `Mem[]` arrays, you still need to keep in mind the size of each structure element to find the offset to the next one. Thus, `DVAL` is offset by two from `XVAL` since a `complex` is two words. However, adjacent fields have consecutive offsets (`$1`, `$1+1`, ...) if they occupy a single word. Note also the use of a second argument in `IARRAY` to specify the array element, the position within the chunk of the allocated memory. The above structure definition would be used by first allocating memory for the structure and accessing each field using the returned structure pointer, as shown in Example 1.10.

```
include myincl.h

complex xconst
double  dconst
real    rconst
pointer mstr
int     i
.
begin
     # Allocate memory for the structure
     call malloc (mstr, LEN_MYSTR, TY_STRUCT)

     # Initialize the structure values
     XVAL(mstr) = xconst
     RVAL(mstr) = rconst

     do i = 1, LENARR
         # Array elements
         IARRAY(mstr,i) = ...

     # Character string
     call strcopy (“Hello World”, CSVAL(mstr), 11)
     .
     .
```

Example 1.10: Allocating and Using Structures by Pointer.

Another way to define arrays or character strings in a macro structure is to store only a pointer to dynamically allocated memory in a field of the structure. In this case, the memory for the array has to be allocated explicitly in the code in addition to the memory for the structure.

```
define LEN_MYSTR 2                  # Size of the structure
define R_ARR_P   Memi[($1)]         # Pointer to a real array
define R_ARRAY   Memr[R_ARR_P($1)]  # The array
define CH_STR_P  Memi[($1+1)]       # Pointer to a char string
define CH_STR    Memc[CH_STR_P($1)] # The string

# The structure would be used as follows
define SZ_RARR   1024
pointer mstr
.
.
     # Allocate memory for the structure
     # Note the use of TY_STRUCT for the data type
     call malloc (mstr, LEN_MYSTR, TY_STRUCTY)

     # Allocate memory for the
     # real array in the structure
     call malloc (R_ARR_P(mstr), SZ_RARR, TY_REAL)

     # Fill in the array (with the constant 100)
     call amovkr (100.0, R_ARRAY(mstr), SZ_RARR)

     # Allocate memory for the
     # chracter string in the structure
     call malloc (CH_STR_P, SZ_LINE, TY_CHAR)

     # Initialize the string
     call strcpy (“Hello World”, CH_STR(mstr), SZ_LINE)
     .
     .
```

Example 1.11: Defining Arrays in a Structure with Dynamically Allocated Memory..

Macro Functions

Macros with arguments may also be used to define in-line functions. For example, here are a couple of definitions of character classes from the system include `lib$ctype.h`:

```
define IS_UPPER ($1>=’A’&&$1<=’Z’)      <------------   Character Functions from
define IS_LOWER ($1>=’a’&&$1<=’z’)                      lib$ctype.h
define IS_DIGIT ($1>=’0’&&$1<=’9’)

define RADIAN    57.295779513082320877  <-----------    Math Functions from
define RADTODEG  (($1)*RADIAN)                          hlib$math.h
define DEGTORAD  (($1)/RADIAN)
```

Example 1.12: Macro Definitions.

These are used in the following:

```
include <char.h>
include <math.h>
procedure myproc ()
char    string[SZ_LINE]
real    deg_ang
real    rad_ang
.
.
begin
     # Check if character is a digit
     if (IS_DIGIT(string[i])) {
          .
          .
     }

     # Convert degrees to radians
     deg_ang = DEGTORAD(rad_ang)
end
```

Example 1.13: Using Macro Functions.

Control Flow

SPP provides a full set of control flow constructs found in most modern languages such as conditional execution and repetition. Some of these have already appeared in examples. An SPP control flow construct executes a *statement* either conditionally or repetitively. The statement to be executed may be a simple one line statement, a *compound statement* enclosed in curly brackets or braces, or the *null statement* (`;` on a line by itself). An assortment of repetitive constructs are provided for convenience. The simplest constructs are `while`, which tests at the top of the loop, and `repeat until`, which tests at the bottom. The `do` construct is convenient for simple sequential operations on arrays. The most general repetitive construct is the `for` statement.

- Conditional Constructs
  - `if`
  - `if...else`
  - `switch`
  - `case`

- Repetitive constructs
  - `do`
  - `for`
  - `repeat...until`
  - `while`
- Branching
  - `break`
  - `next`
  - `goto`
  - `return`

Two statements are provided to interrupt the flow of control through one of the repetitive constructs. The `break` statement causes an immediate exit from the loop, by jumping to the statement following the loop. The `next` statement shifts control to the next iteration of a loop. If `break` and `next` are embedded in a conditional construct which is in turn embedded in a repetitive construct, it is the outer repetitive construct which will determine the point to which control is shifted. Note that formatting in the form of indentation and white space is not mandatory, but makes the code more readable and therefore easier to maintain.

if...else

The `if` and `if else` constructs are shown below. The *expr* part may be any boolean expression (see “Expressions” on page 31). The *statement* part may be a simple statement, compound statement enclosed in braces, or the null statement. The statement(s) will be executed if the expression resolves to `true`. Otherwise, it will fall through to the next block consisting of an `else` or `else if`.

```
 if (expr)
        statement
[else if (expr)
        statement]
[else (expr)
        statement]
```

The control flow constructs may be nested indefinitely. There may be an `if` clause without an `else` or `else if`. There is no `end if`. A simple example of an `if ... else ... else if` is:

```
if (counter >= MAX) {
    x = sqrt (a)
    call xpoc (x, y, z)
} else if (counter < MIN) {
    .
    .
}
```

Example 1.14: Using `if..else`.

switch...case

The `switch case` construct evaluates an integer expression once, then branches to the matching case. Each case must be a unique integer constant. The maximum number of cases is limited only by table space within the compiler. A case may consist of a single integer constant, or a list of integer constants, separated by commas and terminated by the colon character`:`. The special case `default`, if included, is selected if the switch value does not match any of the other cases. If the switch value does not match any case, and there is no default case, control passes to the statement following the body of the `switch` statement. In *every* case, control passes to the statement following the switch. A `break` statement is not needed after each case (in contrast to the `switch ... case` statement in C). Each case of the `switch` statement may consist of an arbitrary number of statements, which do not have to be enclosed in braces. The body of the `switch` statement, however, must be enclosed in braces as shown below.

```
switch (expr) {
case list:
          statements
[case list:
          statements]
.
.
[default:
          statements]
}
```

For example:

```
switch (operator) {
case '+':
     c = a + b
case '-':
     c = a - b
default:
     call error (1, "unknown operator")
}

# or

switch (key) {
case 'a', 'A':
     .
     .
case 'b', 'B':
     .
     .
}
```

Example 1.15: Using `switch` and `case`.

The `switch` construct will execute most efficiently if the cases form a monotonically increasing sequence without large gaps between the cases (i.e., `case 1`, `case 2`, `case 3`, etc.). Ideally, the cases should be defined parameters or character constants, rather than explicit numbers.

while

The `while` statement repetitively executes a statement or a block of statements as long as the specified condition expression is *true*. The condition is tested at the *beginning* of the loop, so it is possible for the statement not to be executed at all.

```
while (expr)
          statement
```

repeat...until

The `repeat` construct repetitively executes a statement or a block of statements. The simpler form simply repeats forever. The statement block might include a `break` statement to terminate the loop.

The `repeat...until` form executes the statement as long as the logical expression in the `until` statement is *false*. The condition is tested at the *end* of the loop, so the statement will always be executed at least once.

```
repeat
    statement
    until (expr)
```

```
repeat
    statement
```

for

The `for` construct consists of an initialization part, a test part, a loop control part, and a statement to be executed. The initialization part consists of a statement which is executed once before entering the loop. The test part is a boolean expression, which is tested before each iteration of the loop. The loop control statement is executed *after* the last statement in the body of the `for`, before branching to the test at the beginning of the loop. When used in a `for` statement, `next` causes a branch to the loop control statement. The `for` construct is very general, because of the lack of restrictions on the type of initialization and loop control statements chosen. Any or all of the three parts of the `for` may be omitted, but the semicolon delimiters must be present. Only one statement is permitted for each control section, unlike C.

```
for (init; test; control)
    statement
```

For example:

```
for (ip=strlen(str); ip > 0 && str[ip] != 'z'; ip=ip-1)
    ;
```

Example 1.16: Using `for`.

This `for` statement searches the string `str` backwards until the character `'z'` is encountered, or until the beginning of the string is reached. Note the use of the null statement (`;`) in the body of the `for`, since everything has already been done in the `for` itself. The `strlen` procedure is shown in a later example. Note that the above example may result in an error if the string is null, in which case `ip = 0` and the test `str[ip] != 'z'` will try and access a character before the beginning of the string.

do

The `do` construct is a special case of the `for` construct. It is ideal for simple array operations, and since it is implemented with the Fortran `DO` statement, its use should result in particularly efficient code.

```
do lcp = initial, final [, step]
          statement
```

General expressions are permitted as loop control in the `do` statement but their result must be integers. The loop may run forward or backward, with any step size. Note that to operate backward, the step must be negative, and the initial value should be larger than the final value. The body of the `do` will *not* be executed if the initial value of the loop control parameter satisfies the termination condition. For example:

```
do i = 1, NPIX
    a[i] = abs (a[i])
```

Example 1.17: Using do.

break

The `break` statement causes an immediate exit from a loop by jumping to the statement following the loop.

next

The `next` statement immediately shifts control to the next iteration of a loop.

return

The `return` statement assigns a value to a function or returns control to the calling procedure. This value is passed back to the calling procedure as the function value. The returned value is an expression which resolves to the declared data type of the function. For example:

```
real function func (i, x)
real    i
real    x
real    retval
begin
        retval = i * x
        return retval
end
```

Example 1.18: Using the `return` Statement.

goto

The `goto` statement unconditionally branches to another point in a procedure. The target statement is specified by a label, which is an integer constant on the beginning of a line, preceding an executable (unnumbered) statement. For example:

```
     call smark (sp)
     .
     .
     goto 10
     .
     .
10
    call sfree (sp)
```

Example 1.19: Using the `goto` Statement.

Alternately, the label may be assigned a symbolic value using the `define` statement. This permits more mnemonic labels.

```
.
.
define termin_ 10
begin
       call smark (sp)
       .
       .
       goto termin_
       .
       .
termin_
       call sfree (sp)
       .
       .
```

Example 1.20: Using Symbolic Values with `goto` Statements.

The underscore at the end of the label (`termin_` in the example above) is not required. but is a recommended convention to permit the labels to stand out as distinct from other identifiers.

Expressions

An *expression* may be a numeric constant, a string constant, an array reference, a call to a typed (function) procedure, or any combination of the above elements, in combination with one or more unary or binary operators. Every expression is characterized by a data type and a value. The data type is fixed at compile time, but the value may be either fixed at compile time, or calculated at run time. Parentheses may be used to force the compiler to evaluate the parts of an expression in a certain order. In the absence of parenthesis, the *precedence* of an operator determines the order of evaluation of an expression. The highest precedence operators are evaluated first. The precedence of the SPP operators is defined by the order in which the operators appear in the table under heading “Data Types” on page 8. Procedure call has the highest precedence. The argument list in a procedure or array reference consists of a list of general expressions separated by commas. If an expression contains calls to two or more procedures, the order in which the procedures are evaluated is undefined.

Operators

SPP supports the usual arithmetic operators which take operands of any numeric data type. In addition there are the usual comparison operators which take operands of any data type with the data type of the result always boolean. Finally, there are boolean operators taking boolean operands and also resulting in a boolean.

Operator	*Operands*	*Result*	*Operation*
+	Numeric	Numeric	Add
-	Numeric	Numeric	Subtract, negate
*	Numeric	Numeric	Multiply
/	Numeric	Numeric	Divide
**	Numeric	Numeric	Power
<	Numeric	Boolean	Less than
<=	Numeric	Boolean	Less than or equal to
>	Numeric	Boolean	Greater than
>=	Numeric	Boolean	Greater than or equal to
==	Numeric	Boolean	Equal to
!=	Numeric	Boolean	Not equal to
!	Boolean	Boolean	Not
||	Boolean	Boolean	Or
&&	Boolean	Boolean	And
|	*Reserved operator*		
&	*Reserved operator*		

Table 1.8: Arithmetic and Boolean Operators.

Minus (–) may be a binary operator (have two arguments) or unary operator (have one argument) operator. As a binary operator it represents subtraction and as a unary operator it represents negation. The boolean not (!) is always a unary operator.

Mixed Mode Expressions

Binary operators combine two expressions into a single expression. If the two input expressions are of different data types, the expression is said to be a *mixed mode* expression. The data type of a mixed mode expression is defined by the order in which the types of the two input expressions appear in the table under “Data Types” on page 8. The data types are listed in the table in order of increasing precedence. Thus, the data type which appears furthest down in this table will be the data type of the combined expression. For example, an `int` plus a `real` produces a `real`. Mixed mode expressions involving `bool` are illegal. While `char` expressions are permitted, there are no string operators or expressions since there is no fundamental string data type.

Type Coercion

Type coercion refers to the conversion of an object from one data type to another. Such conversions may involve loss of information, and hence are not always reversible. Type coercion occurs automatically in mixed mode expressions, and in assignment statements. Type coercion is not permitted between booleans and the other data types.

Data Type	***Contains***
`aimag`	Imaginary part of `complex`
`complex`	Complex
`double`	Double precision floating point
`int`	Integer
`real`	Single precision floating point

Table 1.9: Data Type Precedence.

The data type of an expression may be coerced by a call to an intrinsic function. The names of these intrinsic functions are the same as the names of the data types. Thus, `int(x)`, where `x` is of type `real`, coerces `x` to type `int`, while `double(x)` produces a double precision result.

The Assignment Statement

The *assignment statement* assigns the value of the general expression on the right side to the variable or array element given on the left side. Automatic type coercion will occur during the assignment if necessary (and legal). Multiple assignments may not be made in a single assignment statement. That is, an assignment statement may have only one equal sign. However, a line may contain more than one statement, separated by semicolons (`;`).

```
i = 5
z[i] = sqrt (x[i]**2 + y[i]**2)
x1 = 0.0; x2 = 1.0
```

Example 1.21: Assignment Expressions.

Procedures

Procedures are the basic units of SPP programs. They also include functions, procedures that return a value. The form of a `procedure` declaration is shown below.

```
[data_type] procedure proc_name ([p1 [, p2 [,... ]]])
[declarations for procedure arguments]
[declarations for local variables]
[declarations for functions]
[initialization]

begin
	[executable statements]
end
```

The *data_type* field must be included if the procedure returns a value. The `begin` keyword separates the declarations section from the executable body of the procedure, and is required. The `end` keyword must follow the last executable statement. Note that the `procedure` statement and the declaration statements *must* begin in the first character on the line.

All parameters, variables, and typed procedures must be declared. The SPP language does not permit implicit typing of parameters, variables, or procedures, unlike Fortran. By convention, declarations of procedure arguments precede local declarations. It is also good practice to use in-line comments to describe the declarations.

If a procedure has formal parameters, they should agree in both number and type in the procedure declaration and when the procedure is called. In particular, beware of `short` or `char` parameters in argument lists. An `int` may be passed as a parameter to a procedure expecting a `short` integer on some machines, but this usage is *not portable*, and is not detected by the compiler. The compiler does not verify that a procedure is declared and used consistently.

If a procedure returns a value it is known as a *function* and the calling program must declare the procedure in a type declaration, and must reference the procedure in an expression. The function procedure must contain a `return` which assigns the value to pass back to the caller as the function value. A function procedure may return a numerical value, but may not return an array or string.

If a procedure does not return a value, the calling program may reference the procedure *only* in a `call` statement. However, the `return` statement may be used to end the procedure at any point and return control to the calling procedure.

begin...end

The executable statements in a procedure must be surrounded by `begin` and `end` statements. All declarations must be placed between the `procedure` statement and the `begin`.

{...}

Braces (`{` and `}`) may be used to bracket explicitly groups of statements intended to be treated as a single statement, for example, in `if`, `for`, or `while` constructs.

Arguments

Formal or *dummy* arguments and *actual* arguments must match in number and type. That is, the declarations in the calling and called procedure must be the same for all of the arguments.

`entry` Statement

Procedures with multiple entry points are permitted in SPP because they provide an alternative to global common when several procedures must access the same data. The multiple entry point mechanism is similar to block structuring. The multiple entry point construct is only useful for small problems. If the problem grows too large, an enormous procedure with many entry points may result, which is difficult to maintain. The form of a procedure with multiple entry points is shown below. Either all entry points should be untyped, as in the example, or all entry points should return values of the same type. Control should only flow forward. Each entry point should be terminated by a `return` statement, or by a `goto` to a common section of code which all entry points share. The shared section of code should be terminated by a single `return` which all entry points share.

```
procedure push (datum)
int    datum            # value to be pushed or popped
int    stack[SZ_STACK]  # the stack
int    sp               # the stack pointer
data   sp/0/
begin
     # Push datum on the stack, check for overflow
     .
     .
     return
entry pop (datum)
     # Pop stack into "datum", check for underflow
     .
     .
     return
end
```

Example 1.22: Using the `entry` Statement.

Intrinsic Functions

Any function written as part of the task must be declared. However, SPP includes several intrinsic functions that need not be declared. The intrinsic functions are generic functions, meaning that the same function name may be used regardless of the data type of the arguments. The arguments to trigonometric functions are assumed to be in radians, as in Fortran.

Function	*Description*
`abs(a)`	Absolute value $\|x\|$
`acos(a)`	Arccosine, returns angle in radians $\cos^{-1} a$
`asin(a)`	Arcsine, returns angle in radians $\sin^{-1}$ a
`atan(a)`	Arctangent, returns angle in radians $\tan^{-1} a$
`atan2(a, b)`	Arctangent, returns angle in radians $\tan^{-1} a$
`char(a)`	Convert to character
`complex(a,b)`	Complex from real and imaginary parts
`conjg(a)`	Complex conjugate
`cos(a)`	Cosine, argument in radians
`cosh(a)`	Hyperbolic cosine, argument in radians
`double(a)`	Convert to double precision
`exp(a)`	Exponential e^a
`int(a)`	Convert to integer, truncate
`log(a)`	Natural logarithm
`log10(a)`	Common logarithm
`long(a)`	Convert to long integer
`max(a, b)`	Maximm
`min(a, b)`	Minimum
`mod(a, b)`	Modulus or remainder $a - [a/b]$
`nint(a)`	Nearest integer
`real(a)`	Convert to single precision
`short(a)`	Convert to short integer
`sin(a)`	Sine, argument in radians
`sinh(a)`	Hyperbolic sine, argument in radians
`sqrt(a)`	Square root
`tan(a)`	Tangent, argument in radians
`tanh(a)`	Hyperbolic tangent, argument in radians

Table 1.10: Intrinsic Functions

Note that the names of the type coercion functions (`char`, `short`, `int`, `real`, etc.) are the same as the names of the data types in declaration statements. The functions `log10`, `tan`, and the hyperbolic functions may *not* be called with complex arguments. As in Fortran, the arguments to trigonometric functions must be in radians.

Calling Fortran Subprograms

Since SPP is preprocessed into Fortran, in most cases, it is quite straightforward to call an existing Fortran subroutine from an SPP procedure. The most important caution is in using character strings. SPP strings are not the same as Fortran strings. SPP strings are implemented as arrays of integers. However, there are procedures available to transform between the two: `f77pak()` converts an SPP string to a Fortran string, and `f77upk()` converts a Fortran string to an SPP string. Note that you must declare the Fortran string in the SPP procedure with a Fortan statement. This is possible with the `%` escape as the first character on a line. This indicates to the xc compiler that the following statement should not be processed but copied directly to the Fortran code. See also “Expressions” on page 31 and “Fortran Strings” on page 125.

Program Structure

An SPP source file may contain any number of `procedure` declarations, zero or one `task` statements, any number of `define` or `include` statements, and any number of `help` text segments. By convention, global definitions and include file references should appear at the beginning of the file, followed by the task statement, if any, and the procedure declarations.

```
include <stype.h>         # Character type definitions
include "widgets.h"       # Package definitions file
include "../more.h"       # In the parent directory

# This file contains the source for the tasks making up the
# Widgets analysis package (describe the contents of the file.
define MAX_WIDGETS    50     # Local definitions
define NPIX           512
define LONGITUDE      7:32:23.42
task   alpha, beta, epsilon=eps

# ALPHA -- (describe the alpha task)
procedure alpha()
.
```

Example 1.23: Program Structure.

Include Files

Include files permit an external file to be inserted into SPP code. They are referenced at the beginning of a file to include global definitions that must be shared among separately compiled files, and within procedures to reference common block definitions. Two forms allow for system-defined includes or user-defined includes. The `include` statement is effectively replaced by the contents of the named file. Includes may be nested at least five deep. The most common uses for include files are macro definitions and structure declarations to be shared by several source files comprising a task. The name of the file to be included must be delimited by either angle brackets (<*file*>) or quotation marks ("*file*"). The first form is used to reference the IRAF system include files. This includes *external packages* such as STSDAS if these are installed. The second, more general, form may be used to include any file. The file name may include an absolute or relative directory path. However, the safest and most portable method of accessing include files in SPP source is to have the source and include files in the same directory. You then need only refer to the file itself in the `include` statement without any absolute or relative directory information.

```
include <imhdr.h>     # Include image header system definitions
include "mytask.h"    # Application task definitions
include "../more.h"   # In the directory above
```

Example 1.24: Using Include Files.

Help Text

Documentation may be embedded in an SPP source file either by commenting out the lines of text using the # character or by enclosing the lines of text within `.help` and `.endhelp` directives. If there are only a few lines of text, it is probably most convenient to comment them out. Large blocks of text should be enclosed by the help directives, making the text easier to edit, and accessible to the on-line documentation and text processing tools.

```
# Everything from the '#' to the end of line is a comment

.help [keyword [qualifier [package description]]]

help text

.endhelp
```

Figure 1.3: Commenting out Documentation Blocks.

The preprocessor ignores comments, and everything between `.help` and `.endhelp` directives. The directives must occur at the beginning of a line to be recognized. In both cases, the preprocessor ignores the remainder of the line. The arguments to `.help` are used by the `help` cl utility, but are ignored by SPP. Help text may be typed in as it is to appear on the terminal or printer, or it may contain text processing directives. See the cl `lroff` documentation for a description of the IRAF text processing directives. Manual pages (help text) for tasks may be stored either directly in the source file as help text segments, or in separate files. If separate source and help files are used, both files conventionally have the same root name, and the help text file should have the extension `.hlp`.

The `task` Statement

The `task` statement is used to make an IRAF task. A file need not contain a task statement, and may not contain more than a single task statement. Files without task statements are separately compiled to produce object modules, which may subsequently be linked together to make a task, or which may be installed in a library. An executable program requires a `task` statement, although it may be in a file by itself. This is then linked with the other procedures making up the task.

```
task    ltask1, ltask2, ltask3=proc3
```

If the task name is identical to the main procedure of the task, then only the task name needs to be in the `task` statement. The main procedure may have a different name, however. In this case, the procedure name must be specified in the `task` statement with an assignment.

```
task doit = t_doit

procedure t_doit ()
begin
    .
    .
end
```

Example 1.25: The `task` statement.

Generic Preprocessor

There are many cases in which the same algorithm may need to be implemented for several different data types. The *generic preprocessor*, in addition to SPP converts a generic procedure into a set of procedures specific to particular data types. We mention this briefly here and refer to a more detailed discussion in “Generic Preprocessor” on page 167 and `help generic` in the IRAF cl, which describe all of the preprocessor directives and the command used to process generic code. Many useful examples of generic procedures exist in IRAF, particularly in the **vops** package, a library of generic procedures dealing with vector operations implemented for the SPP data types. See “Vector (Array) Operators — vops” on page 103 for a description of this package. To indicate the flavor of this facility, here is an example of generic code from the **vops** package:

```
# AABS -- Compute the absolute value of a vector (generic).

procedure aabs$t (a, b, npix)

PIXEL a[ARB], b[ARB]
int npix, i

begin
do i = 1, npix
     b[i] = abs(a[i])
end
```

Example 1.26: Generic Code from **vops** Package.

The generic preprocessor will replace the `$t` suffix on the procedure name by the single character initial of the data type (`s`, `i`, etc.). The preprocessor directive `PIXEL` is replaced by the appropriate data type declaration (`short`, `int`, etc.).

CHAPTER 2:

Libraries and Packages: The VOS Interface

The IRAF Virtual Operating System (VOS) comprises several libraries of procedures that provide the interface to IRAF, permitting an SPP application to access images, cl parameters and so forth. It provides an environment for developing scientific analysis applications. The libraries described here are available to any SPP application without explicitly including the library when linking. Other libraries exist that may be included. In addition, an applications package may create its own library.

Several VOS packages have associated include files which may be used for predefined constants, structures, and other macros. These may be included in code with the <*file*> syntax (see "Include Files" on page 36). Note that here the term *package* refers to a set of procedures in a library, not a set of applications tasks available in the IRAF cl.

The VOS procedures are grouped into library packages of related procedures. Most of them deal with input and output of various forms.

- ***clio*** - Interaction with the cl
- ***memio*** - Dynamic memory allocation
- ***imio*** - Image access
- ***fmtio*** - Formatted I/O
- ***fio*** - Basic file I/O
- ***vops*** - Vector (array) operations
- ***gio*** - Vector graphics
- ***tty*** - Terminal I/O
- ***osb*** - Bit and byte operations

- ***plio*** - Pixel lists
- ***mwcs*** - World coordinate system
- ***etc*** - Miscellaneous

The procedures described here represent the normal interface between an SPP program and the IRAF environment. That is, they are the *only* procedures that should be called. While additional, lower-level, procedures exist in the library, these should not be used. The top-level interface is intended to be stable and well documented. The remainder of the library cannot be guaranteed to remain free of modifications such as changes to the calling sequence. Using lower level procedures in portable, maintainable code represents an *interface violation* and causes potential maintenance problems.

This chapter describes many of the VOS package library procedures. While every attempt has been made to provide comprehensive and up-to-date information on the VOS packages, there are quite a few libraries and the number of individual procedures is quite large. An exhaustive description of each procedure and its calling sequence is beyond the scope of this reference. In particular, it is not practical to describe each procedure in extensive detail. Nor is there room to fully describe every calling argument to every procedure. However, in many cases it should be clear what the data type and meaning are for most of them. In many cases, they are discussed in the text. Examples are used throughout to demonstrate the most commonly used procedures. Ideally, there would be a complete document for every library package describing each procedure and its calling arguments in detail. An example is **gio** with a quite complete reference. However, not every package has such complete documentation.

There is usually a table describing the important procedures in a given library package. If there is a variable and equals sign then the procedure is a function. If there is no variable assignment, the procedure is invoked by a `call` statement. It should be fairly clear what is the data type of the function by the variable name. In many cases, a given procedure is implemented separately for several different SPP data types. That is, there is a separate procedure for each data type. In that case, there is usually a single entry in the table for that family of procedures with the suffix *t* indicating to specify the data type with the initial of the data type name.

You should refer to the source code for the definitive description of any procedure. The best sources for such information is in the IRAF system itself. Each package resides in a separate directory below the IRAF `sys` directory, with the same name as the package. This directory contains the source code for the package library procedures. In addition, there is usually a `doc` directory below this source directory, containing help files or additional documentation. For example, the directory `sys$imio` contains the source and additional documentation for the **imio** library. Note also that the IRAF cl defines an environment variable for each library with the same name, `imio` or `fmtio`, for example. Therefore, the source to `immap()` is in `imio$immap.x`. It is quite instructive to look at the source files as well as the associated documentation. Note however, that these source directories contain *all* of the library procedures. This includes lower level code, not intended to be called by SPP applications tasks, but by the library procedures themselves.

2.1 Interaction with the cl — clio

The **clio** package allows an application to interact with the IRAF command language (cl). This includes mostly reading and writing cl parameters. In addition, there is a set of procedures for handling *filename templates*, lists of input files, as well as satisfying interactive graphics input (cursor position). Parameters in the cl may have a data type attribute as SPP parameters are typed. The SPP data type need not match the cl parameter's data type, however. The data type is silently converted by **clio**. The typed procedures returning cl parameter values refer to the data type of the SPP variables accepting the value of the cl parameter.

Ordinary Parameters

There is a separate read (get) and write (put) procedure for each SPP data type. All of the get procedures, *except strings*, are functions, returning the value of the cl parameter as the function value. Each function takes a single argument of type `char`, the cl parameter name. When the function is called, the cl will attempt to resolve the value of the parameter from a default in a parameter file or prompt for input from the standard input stream `STDIN` (see “Formatted I/O — fmtio” on page 78). If the program is not connected to the cl (i.e., if it is run stand-alone), a prompt will be written to `STDOUT` and the value of the parameter is read from `STDIN`. In the case of string parameters, there is a get and put procedure, returning the string value in a calling argument.

Function Call		***Purpose***
`value = clget`*`T`*	`(parname)`	Get the value of a cl parameter
`clgstr`	`(parname, string, maxch)`	Get a cl string parameter
`value = clput`*`T`*	`(parname, value)`	Put the value of a cl parameter
`clpstr`	`(parname, string)`	Put a cl string parameter
`clgwrd`	`(parname, keyword, maxchar, dictionary)`	Get an enumerated string

Table 2.1: Parameter I/O Functions.

The procedures to read and write numeric parameters are implemented for each SPP data type: `bool, char, short, int, long, real, double, and complex.` Use the appropriate procedure by replacing *T* with the first letter of the corresponding data type, `clgetr()` for type `real` or `clgeti()` for type `integer`, for example. Note that the data type of the returned value need not match the parameter's data type. Implicit type conversion is done by **clio**.

The `parname` parameter is a `char` variable containing the parameter name. This may be a literal string, a predefined string parameter constant, or a character variable containing the desired string (which may also have been read with `clgstr()`). In the case of `clgstr()`, the additional parameter `maxch` specifies the size of the string parameter. The following example illustrates **clio** by reading several parameters from the cl.

```
task readcl

procedure readcl ()

int    ival             # An integer
short  sval             # A short integer
real   rval             # A real
char   strval[SZ_LINE]  # A string of size SZ_LINE
                        # (a predefined constant)

int    clgeti()
real   clgetr()
short  clgets()

string ipar "intpar"    # The cl parameter name of an integer
string spar "shortpar"  # The cl parameter name of a short
string rpar "realpar"   # The cl parameter name of a real

begin
     # Use clget functions for numeric parameters
     # String variables contain the parameter names
     ival = clgeti (ipar) # Get an int
     rval = clgetr (rpar) # Get a real
     sval = clgets (spar) # Get a short

     # Get the string
     call clgstr ("strpar", strval, SZ_LINE)
end
```

Example 2.1: Reading Parameters From the cl.

Note the literal string constants for the parameter names and the predefined constant `SZ_LINE` specifying the size of the returned string. Also, note the distintion between the variable assigned a value in the code and the parameter as defined in the cl. There is no `short` data type in the cl, only integers. The procedure `clgets()` reads a cl parameter of any data type into a short variable. The cl parameter `shortpar` is declared as an integer but the variable `sval` is declared `short`.

Such a procedure implemented as part of a task may use a *parameter file* to specify attributes of parameters. This is a text file with a root name the same as the task name and an extension `.par`. The above example defines a task `readcl` whose parameter file would be called `readcl.par`, containing the lines shown in. See “Parameter Files” on page 171 for a more detailed description of `.par` files.

```
# Parameter file for task readcl
intpar,i,a,0,1,20,"Integer parameter"
realpar,r,a,-1.2,-10.9,99.8,"Floating point parameter"
strpar,s,a,"hello",,,"String parameter"
# There aren’t really shorts in the cl, only integers
shortpar,i,a,1,,,"Short parameter"
```

Example 2.2: Parameter File.

The `clgwrd()` procedure returns the value of an *enumerated string* parameter. This is a string parameter whose value may take on one of a list of possible values. The list of possibilities is specified in the parameter file in the minimum value field as a quoted string with values delimited by a vertical bar. For example the parameter color might permit the selection of several possible values. The definition in the parameter file might be:

```
color,s,h,"black","|black|white|red|green|blue|",,"color"
```

The cl uses minimum matching to determine the desired value from the smallest unique initial characters the user specifies for the string. You must specify the *dictionary* or the list of possible values to `clgwrd()` in the `dictionary` argument returns the full word in the `keyword` argument.

One pitfall is the potential mismatch between the enumeration string in the parameter file and the dictionary in the source. However, it is possible to read the enumeration string using `clgstr()` since it is possible to read the individual components of the parameter definition in addition to its value. The following would return the dictionary for the `color` parameter as defined above:

```
call clgstr ("color.p_min", colordict, SZ_LINE)
```

Where `colordict` is a string variable and would be used in the `clgwrd()` call:

```
call clgwrd ("color", color SZ_LINE, colordict,)
```

pset parameters

Any cl parameter may be included in a *pset*. A pset is a set of cl parameters referred to as a group via a single parameter of a task. The pset itself is defined as a task in the cl and is defined by a `.par` file. In the SPP code, however, pset parameters are accessed identically to any other task parameter. While you *may* prepend the pset name to the parameter name, this is not necessary and not recommended.

List Structured Parameters

List structured or *list-directed* parameters permit a number of values to be accessed by an application from a file specified by name. The following procedures get list structured parameters from the cl. The first two return a status value which is `EOF` on reading at the end of file on the input. The `clglp`*t*`()` procedures return the value of the appropriate data type as the function value.

Procedure Call	***Purpose***
`status = clglpT  (param, value)`	Get a numeric parameter
`len = clglstr (param, outstr, maxch)`	Get a string parameter

Table 2.2: List-Structured Parameter Functions.

The procedure represented by `clglp`*T*`()` reads a numeric list structured parameter and is implemented for the usual SPP data types: `bool`, `char`, `short`, `int`, `long`, `real`, `double`, and `complex`. It returns the value as the second procedure argument, whose data type should match the procedure. The function return value is an integer status that takes the value `EOF` upon reading after the last parameter in the list. The other procedure, `clglstr()` returns the length of the string read as the `int` function value, or `EOF` after reading the last string. For example, we may wish to read integer values from a list filename `int_file.txt` which contains the following:

```
1
22
333
4444
55555
666666
```

If we add the following statements to the program `readcl` in the previous section:

```
.
.
while (clglpi ("intval", ival) != EOF) {
    call printf ("integer value: %d\n")
       call pargi (ival)
}
.
.
```

then the parameter file should have the following line:

```
intval,*i,a,"int_file.txt",,,"> List of integer elements"
```

Notice the additional flexibility to input data to a program; changing the input list filename gives you another set of values.

Vector Parameters

It is possible to access a group of parameter values using a single root parameter name. This provides the capability of vectors or arrays in cl parameters. The array structure, default values, ranges, etc. may be specified in the `.par` file as with scalar parameters. However, the syntax is slightly different. For example, the following declares a singly dimensioned real array having three elements.

```
vecreal,ar,a,1,3,1,,,"real vector elements", 0.0,1.2,3
```

Note that the character `a` precedes the data type field, the next three fields specify the dimensionality, size, and starting index, and the default values are *after* the prompt string. The following code (Example 2.3) will read the above values.

```
real     arr[3]
.
.
     arr[1] = clgetr("vecreal[1]")
     arr[2] = clgetr("vecreal[2]")
     arr[3] = clgetr("vecreal[3]")
```

Example 2.3: Reading Vector Parameters.

Note that the element number of the cl parameter vector is enclosed in square brackets following the parameter name and is part of the string passed to the `clget`*T*`()` and `clput`*T*`()` procedures.

Interactive Graphics Cursor

The cl treats an interactive graphics input cursor read similarly to a list structured cl parameter query. When the user asks for a cursor position, either through a cl query or through a task, the cl issues a prompt which the user must satisfy with some action. In the case of a normal cl parameter, the user may type in the value of the parameter. For a cursor read (assuming a graphics terminal with cursor capability) the graphics enters *graphics input* (GIN) mode. The user may then move the cursor on the screen. To terminate graphics mode, the user types a key on the keyboard. This satisfies the query prompt and the cl returns the cursor position. The `clgcur()` procedure returns the next cursor value from a list structured cursor type parameter. The format of a cursor value is as follows:

```
x y wcs key sval
```

where

- *x, y* - are the *x* and *y* cursor coordinates
- *wcs* - is the world coordinate system in which cursor coordinates are given
- *key* - is the key (stroke) value associated with cursor read
- *sval* - is an optional string associated with the given key

All of the fields need not be given, and extra fields may be supplied and will be either ignored or returned in *sval*. The *x y*, and *wcs* fields may be omitted, in which case the input is *key sval*, causing `INDEF INDEF 0` *key sval* to be returned, exactly as if the `INDEF INDEF 0` had been typed in. The number of fields read is returned as the function value; `EOF` is returned when the end of the cursor list is reached. Since the cl treats a cursor query as a parameter, the **clio** procedure `clgcur()` is used to perform interactive graphics input from an SPP task. Its calling sequence is:

```
call clgcur (param, wx, wy, wcs, key, strval, maxch)
```

Field Types and Names	*Contents*
`char param`	cl parameter name
`real wx, wy`	World coordinates of cursor
`int wcs`	Index of WCS at cursor position
`int key`	Keystroke value used to return cursor
`char strval [maxch]`	String command if `key = ':'`
`int maxch`	Size of `strval`

Table 2.3: Graphics Cursor Parameters.

Note that the argument `key` is an `int` typed variable, not `char` as might be expected.

There are two flavors of cursor available through the cl: for vector graphics and image display. The cl data type of a cursor parameter may be either `*gcur` for a graphics cursor parameter or `*imcur` for an image display cursor parameter.

See "Vector Graphics — gio" on page 114 for a brief description of the graphics procedures. See the **gio** reference manual (*Graphics I/O Design* [Tody84b]) for a more complete description of cursor interaction.

cl **Command**

A quite general method is available to execute any cl command (task) from an SPP application. The procedures `clcmd()` and `clcmdw()` send a string as a command line to the cl. The single argument to both procedures is a string containing the command to execute. The only difference between the two procedures is that `clcmdw()` waits for the completion of the command before returning to the caller.

Procedure Call	*Purpose*
`clcmd (cmd)`	Send a command line to the cl
`clcmdw (cmd)`	Send a command to the cl and wait for completion

Table 2.4: CL Command Execution Procedures.

Sending an explicit command to the cl requires that the task have detailed knowledge of the capabilities of the cl and of the syntax of the command language. This means that the task is very dependent on the cl and may no longer work if the cl is modified, or if there is more than one version of the cl in use in a system. For this reason `clcmd()` should only be used where it is truly necessary, usually only in system utilities.

2.2 Memory Allocation — memio

Memory may be dynamically allocated within an SPP application. The memory is referenced by a *pointer*, an `int` value containing the memory location of the first element of the buffer. The allocated memory may then be accessed as if it were a statically allocated array. The advantages to allocating memory dynamically are to reduce the size of compiled code and to allocate arrays whose size is not known at compile time. The pointer is used in subsequent procedure calls to refer to the allocated memory. The `Mem`m[] construct is used to access the data. When passed to a procedure, the data are treated simply as an SPP array.

Pointers are indices into (one indexed) Fortran arrays. A pointer to an object of one data type will in general have a different value than a pointer to an object of a different data type, even if the objects are stored at the same physical address. Pointers have strict alignment requirements, and it is not always possible to coerce the type of a pointer. For this reason, the pointers returned by `malloc()` and `salloc()` are always aligned for all data types, regardless of the data type requested.

There are two types of dynamically allocated memory: stack and heap. They are treated identically in terms of dealing with the allocated data, but the mechanics of the allocation differ slightly.

`malloc` and relatives

Heap memory is used for arbitrarily large buffers and the resulting pointers may be stored and passed to calling and called procedures.

Procedure and Variables	***Purpose***
`malloc (memptr, size, datatype)`	Allocate heap memory
`calloc (memptr, size, datatype)`	Allocate cleared heap memory
`realloc (memptr, size, datatype)`	Reallocate memory
`mfree (memptr, datatype)`	Free heap memory

Table 2.5: Heap Memory Allocation Procedures.

Many VOS library procedures return a pointer allocated by `malloc()`, the imio procedures, for example. Be sure to free the memory by using the `mfree()` procedure. Note that the `mfree()` procedure in addition to the allocation procedures requires the data type of the allocated memory as an argument. These data types are passed as predefined parameter constants, defined by the system, for example, `TY_INT, TY_REAL`, etc.

Parameter	***Word Size***	***Data Type***
`TY_BOOL`	`SZ_BOOL`	Boolean
`TY_CHAR`	`SZ_CHAR`	Character
`TY_SHORT`	`SZ_SHORT`	Short integer
`TY_INT`	`SZ_INT`	Integer
`TY_LONG`	`SZ_LONG`	Long integer
`TY_REAL`	`SZ_REAL`	Single precision real
`TY_DOUBLE`	`SZ_DOUBLE`	Double precision real
`TY_COMPLEX`	`SZ_COMPLEX`	Complex
`TY_STRUCT`	`SZ_STRUCT`	Structure

Table 2.6: Memory Allocation Parameter Data Types.

Memory allocated explicitly with `malloc()` should be freed after use by `mfree()`. Pointers allocated implicitly, by `immap()`, etc., for example, should not be freed explicitly. They will be freed by the appropriate close procedure such as `imunmap()`. The `realloc()` procedure changes the size of a previously allocated buffer, copying the contents of the buffer if necessary. This is useful when allocating memory of unspecified size. For example, when reading from `STDIN`, you might allocate a data buffer initially with some default size. After reading all of the data you may wish to use `realloc()` to insure that the buffer is only as big as the amount of the data read. Note that `realloc()` will allocate new memory if the passed pointer is `NULL`, so it may be used in place of `malloc()`. This may be useful in a loop in which you need not use `malloc()` the first time you enter the loop. The only difference between `malloc()` and `calloc()` is that the latter sets all of the buffer values to zero, while the former retains the contents of the memory locations, which should be considered garbage. The following example illustrates allocating a block of memory using `malloc()` and calling a procedure to perform some operation on the values.

```
procedure mexamp (ncols, nrows)

# Dynamically allocate memory and perform some operation on array

int      ncols, nrows       # Number of columns and rows
pointer  buff               # The memory buffer pointer

begin
     # Allocate a real memory buffer with the passed size
     # and data type
     call malloc (buff, ncols*nrows, TY_REAL)

     # Operate on the buffer, dereferenced with Memr
     # Pass the size of the array
     call dostuff (Memr[buff], ncols, nrows)

     # Free the memory
     # Pass the data type
     call mfree (buff, TY_REAL)
end

procedure dostuff (buffer, ncols, nrows)

# Operate on a 2-D array

real buffer[ncols,nrows]
int  ncols, nrows
int  i, j

begin
     do j = 1, nrows {
        do i = 1, ncols {
          .
        .
        buffer[i,j] = ...
        .
        .
        }
     }
end
```

Example 2.4: Allocating and Using a Memory Block.

Note that the `dostuff()` procedure need not have nested loops if the operation is independent of column or row information. In fact, the vector operator (vops) procedures may be used for any dimensionality of arrays.

smark and salloc

Stack memory is useful for small buffers local to a procedure.

Procedure Call	***Purpose***
`smark (stkptr)`	Mark memory stack
`salloc (memptr, size, datatype)`	Allocate stack memory
`sfree (stkptr)`	Free memory stack

Table 2.7: Stack Memory Procedures.

The `salloc()` procedure allocates stack memory. This is a preallocated block of memory, a chunk of which may be used temporarily by a task. This differs from `malloc()` which allocates the memory at the time it is called. To use `salloc()`, a stack pointer must be referenced first using the `smark()` procedure. This marks the beginning of the block of memory to be referenced. It is not necessary (nor possible) to free the individual memory buffers allocated using `salloc()`. However, the stack pointer should be reset using `sfree()` at the end of the procedure. The memory pointer returned by `salloc()` should not be passed back to a calling procedure but may be passed down to a called procedure. Otherwise stack memory is used identically to heap memory allocated by `malloc()` or `calloc()`, see Example 2.5.

```
pointer  sp
pointer  cbuf
pointer  rbuf

begin
     # Mark the memory stack
     call smark (sp)

     # Allocate a character buffer
     call salloc (cbuf, SZ_LINE, TY_CHAR)

     # Allocate a real buffer
     call salloc (rbuf, npix, TY_REAL)

     # Pass the memory buffers to a procedure
     call myproc (Memc[cbuf], SZ_LINE, Memr[rbuf], npix)

     # Free the memory stack
     call sfree (sp)
end
```

Example 2.5: Using Stack Memory.

Data Structures

Dynamic memory is often used in creating and using data structures (see “Macro Definitions” on page 16 and “Data Structures” on page 18 for more details and additional examples). The structure is described by macro `define` statements declaring the components of the structure. These may be based on dynamically allocated memory, in which case the memory must be allocated before the structure is addressed, and the memory pointer passed as an argument to the structure element. Example 2.6 shows some code that may reside in an `include` file; it declares a structure consisting of integers and strings.

```
define LEN_IGS        20                 # Structure size
define CMD_STATE      Memi[($1)+1]       # Command state
define LAST_CMD_PNT   Memi[($1)+5]       # Last command buffer location
define WRITE_CMD      Memi[($1)+7]       # Write command to buffer?
define SP_OUT_P       Memi[($1)+8]       # Temp output file name pointer
define SPOOL_OUTPUT   Memc[SP_OUT_P($1)] # Temp output file name
define SYM_TABLE      Memi[($1)+11]      # Symbol table pointer
define TOKEN_VALUE    Memi[($1)+12]      # Token value structure
define INPUT_SOURCE   Memi[($1)+13]      # Input stream descriptor
define STATE_STACK    Memi[($1)+15]      # Command state stack
define PLOT_PARMS     Memi[($1)+18]      # Plot parameters structure
```

Example 2.6: Declaring a Data Structure.

The strings (`SPOOL_OUTPUT`, for example) are in turn declared using dynamically allocated memory, the pointer being saved in another element of the structure. The elements are addressed with the `Mem` constructs. To use this structure, the memory must first be allocated using `malloc()` or `calloc()` with a data type of `TY_STRUCT` (see Example 2.7). The first line of the macro provides the number of elements to allocate. Elements of the structure are referenced name, with the pointer to the dynamically allocated memory passed as an argument to the macro.

```
# Allocate the structure
call malloc (igs, LEN_IGS, TY_STRUCT)
.
.
# Allocate string structure elements
call malloc (SP_OUT_P(igs), SZ_LINE, TY_CHAR)
call malloc (SP_OUT_P(igs), SZ_LINE, TY_CHAR)
.
.
# Assign an integer structure element
INPUT_SOURCE(igs) = STDIN
.
.
# Fill the string structure element
call strcpy ("test", SPOOL_OUTPUT(igs), SZ_LINE)
```

Example 2.7: Using the Memory Structure.

There may also be substructures, pointed to by an element of the primary structure. Example 2.8 shows a substructure called from the **gio** structure defined in `lib$gset.h`.

```
define GP_PLAP      ($1+20)   # polyline attributes
.
.
define LEN_PL       4
define PL_STATE     Memi[$1]  # polyline attributes
define PL_LTYPE     Memi[$1+1]
define PL_WIDTH     Memi[$1+2]
define PL_COLOR     Memi[$1+3]
```

Example 2.8: Substructures of a Data Structure.

Example 2.8 defines a structure for storing polyline attributes. `GP_PLAP` is a member of the top-level **gio** structure and `PL_LTYPE` for example is a member of the polyline substructure. These would be used in code as shown in Example 2.9.

```
include <gio.h>
.
.
pointer plap, pmap
.
.
begin
     plap = GP_PLAP(gp)
     .
     .
     PL_LTYPE(plap) = linetype
     .
     .
```

Example 2.9: Using the Substructures.

A more complicated example (Example 2.10) illustrates a two-dimensional array in a substructure, again from **gio**. Note the use of two arguments to the macro, referred to as `$1` and `$2` in the definition.

Example 2.11 shows how the two-dimensional in the structure could be used. Note the two arguments to the macro `GP_WCSPTR`, one of which is itself a symbolic definition, `GP_WCS`, also part of the data structure. The structure defined in Example 2.10 is a fragment of the **gio** header file `gset.h`, included in the source example.

```
define LEN_WCS          11
define LEN_WCSARRAY     (LEN_WCS*MAX_WCS)
.
.
define GP_WCSPTR        (($2*LEN_WCS+$1+150)  # pointer to WCS substructure
.
.
# WCS substructure
define WCS_WX1          Memr[$1]        # window coordinates
define WCS_WX2          Memr[$1+1]
define WCS_WY1          Memr[$1+2]
.
.
define WCS_XTRAN        Memi[$1+8]      # type of scaling (linear,log)
define WCS_YTRAN        Memi[$1+9]
define WCS_CLIP         Memi[$1+10]     # clip at viewport boundary?
```

Example 2.10: Defining a 2-Dimensional Array in a Structure.

```
include <gio.h>

procedure gswind (gp, x1, x2, y1, y2)

pointer gp         # graphics descriptor
real    x1, x2     # range of world coords in X
real    y1, y2     # range of world coords in Y

pointer w

begin
     w = GP_WCSPTR (gp, GP_WCS(gp))
     if (!IS_INDEF(x1))
        WCS_WX1(w) = x1
     .
     .
     if (!IS_INDEF(y2))
        WCS_WY2(w) = y2
     .
     .
end
```

Example 2.11: Using the Structure.

2.3 Accessing Images — imio

Procedures in the sf **imio** library allow an SPP application to read and write IRAF images. IRAF supports several different image formats, including old IRAF, (OIF format), GEIS or STSDAS (STF format) and PROS (QPOE format). However, the same **imio** procedures are used regardless of the specific format of the image so the formats are transparent to the applications program. The details of decoding the image files are buried in the *kernels* beneath the applications level of **imio**. The *user* specifies the format type when specifying the image names as input or output to the task. The `imtype` cl environment variable also may be used to specify the default image type. A specific image name extension overrides the value of `imtype`. The imio interface supports images of up to seven dimensions. In a sense, all images are multidimensional, with the higher, unused axis lengths set to one. An *n* dimensional image may therefore be accessed by a program coded to operate upon an *m* dimensional image.

Open

To access an image, you must first open it using the `immap()` function.

Procedure Call	***Purpose***
`imp = immap (filename, mode, template)`	Open an image file
`imunmap (imp)`	Close an image

Table 2.8: Image I/O Functions.

This returns a pointer type variable that is the address of the image descriptor structure. The `immap()` function has three arguments. The first argument is the image filename, passed as a string, the second is a mode specifying how to access the image. It is an integer usually passed as a symbolic constant parameter. The access mode argument may be one of the following symbolic parameters:

Parameter	*Access Mode*
`READ_ONLY`	Read only
`READ_WRITE`	Read and write
`WRITE_ONLY`	Write only
`NEW_FILE`	New image
`NEW_COPY`	New image, header copied from open image
`NEW_IMAGE`	Alias for `NEW_FILE`

Table 2.9: Access Mode Parameters.

The third argument is the pointer to another image, already opened with another `immap()` call. It is used only if the access mode is `NEW_COPY` and specifies a *template* image. The header of the template image will be copied to the header of the new image, but not the pixel values. That is, the structure of the new output image will be similar to the existing image, but the pixels will be different.

`imunmap()` releases any dynamically allocated memory used for file and I/O buffers. Note that **imio** refers to images by the header filename, regardless of the format of the image. Therefore, if you do specify an extension on the image filename in a call to `immap()`, use the header file extension, not the pixel file.

Extension	*Image Format*
`.imh`	OIF, Old IRAF
`.hhh`	STF, STScI GEIS
`.qp`	QPOE, PROS

Table 2.10: Image Formats.

For example,

```
im = immap ("taurus.imh", READ_ONLY, 0)
```

You may omit the extension, in which case **imio** will interpret the filename as an image header. If there is only one image with the specified root name, then it will open that one, regardless of the image format. If there are two images with the same root but different extensions (different image formats), **imio** will open the one in OIF (IRAF) format.

Of course, it is usually up to the user to specify a filename. You need not append an extension unless you wish to force a particular format, or if you wish to use a non-standard extension. If the task creates an image from scratch (using `NEW_IMAGE`, not copying an existing image) there is an additional way to control the image format. The cl environment variable `imtype` specifies the image format if there is no extension to the output image filename.

Image data are passed from **imio** procedures to the application via pointers in dynamically allocated memory. These **imio** procedures comprise families of calls to read and write the pixel data. Each `pointer` typed function returns a pointer to dynamically allocated memory containing the specified part of the image.

Arbitrary Line I/O

These procedures read image data one line at a time. They allocate a block of memory containing the pixels and return the memory pointer as the function value.

Procedure Call	***Purpose***
`bpt = imgl1`*`T`*` (imp)`	Get a 1-D image
`bpt = imgl2`*`T`*` (imp, line)`	Get a line from a 2-D image
`bpt = imgl3`*`T`*` (imp, line, band)`	Get a line from a 3-D image
`bpt = impl1`*`T`*` (imp)`	Put a 1-D image
`bpt = impl2`*`T`*` (imp, line)`	Put a line to a 2-D image
`bpt = impl3`*`T`*` (imp, line, band)`	Put a line to a 3-D image

Table 2.11: Image Line I/O Functions.

All of the above procedures are implemented for the usual SPP numeric data types: `short`, `int`, `long`, `real`, `double`, and `complex`. That is, the procedure name represents the data type of the SPP buffer that holds the image pixels, not necessarily the data type of the image file. The returned pointer type function value is a pointer to memory allocated by **memio** for the line of pixels from the image. This differs from the image file descriptor (`imp` above), which is a pointer to a structure containing the attributes of the image as a whole. The pixel data may be passed to another procedure via the `Mem[]` construct.

You need not explicitly deallocate memory allocated by any imio procedure. However, you should call `imunmap()` for any images opened with `immap()`. This will flush I/O buffers and free allocated memory.

Note that the output (`imp...()`) procedures as well as the input (`img...()`) procedures return a pointer to dynamic memory. The pixels are written to the file when the output buffer is full; in some cases, not until the image is closed, or when flushed explicitly. When writing to an output image, your procedure fills the buffer associated with the pointer and then calls the `imp...()` procedure.

Example 2.12 is a simple example of copying one image into another using arbitrary line I/O.

```
# IMCOPY -- Copy a 2-D image. The header information is preserved.
# The output image has the same size and pixel type as the input image.
# An image section may be used to copy a subsection of the input image.

procedure imcopy (in_image, out_image)

char    in_image[ARB]
char    out_image[ARB]

int     npix, nlin
int     line
pointer in, out, l1, l2
pointer immap(), imgl2r(), impl2r

begin
    # Open the input image.
    in = immap (in_image, READ_ONLY, 0)

    # Open the output image as a copy of the input
    out = immap (out_image, NEW_COPY, in)

    # Fine the line size
    npix = IM_LEN(in,1)
    nline = IM_LEN(in,2)

    do line = 1, nlin
       # Copy the image line
       call amovr (Memr[(imgl2r (in, line)],
          Memr[impl2r (out, line)], npix)

    # Close the images
    call imunmap (in)
    call imunmap (out)
end
```

Example 2.12: Copying Images Using Arbitrary line I/O.

Line by Line I/O

Another family of procedures returns a pointer to a line of an image, progressing through adjacent lines with each successive call. These differ from the previous family in that those allow a particular line to be read in random order. These procedures return the next line in order.

Procedure Call	***Purpose***
`status = imgnl`*`T`*` (im, bufptr, v)`	Get next image line

Table 2.12: Line by Line I/O.

This family of procedures is implemented for the usual SPP numeric data types: `short`, `int`, `long`, `real`, `double`, and `complex`. The functions return the buffer pointer in an argument, `bufptr`, not in the function value as the previous procedures. These procedures return a completion status as the function value which may be tested for `EOF`. The argument `v` is a `long` array containing indexes of the line to read. This should be initialized to ones. After each call to `imgnl`*T*() it is updated to contain the index of the next line. See the example below.

This family of procedures is useful for operating on an image line by line, without regard for the absolute size or even the dimensionality of the image. Because of the buffering of image input and output and a certain amount of asynchronous I/O, substantially more efficient code can result. Example 2.13 demonstrates line by line image I/O by copying an image to a new image. Note that the procedure works the same regardless of the dimensionality and data type of the images. Another, more complete example, can be found in Appendix B.

```
# IMCOPY -- Copy an image. The header information is preserved.
# The output image has the same size, dimensionality, and pixel
# type as the input image.

procedure imcopy (in_image, out_image)

char    in_image[ARB]
char    out_image[ARB]

int     npix
long    one_l
long    v1[IM_MAXDIM], v2[IM_MAXDIM]
pointer in, out, l1, l2
pointer immap(), imgnlr(), impnlr()

begin
     # Open the input image.
     in = immap (in_image, READ_ONLY, 0)

     # Open the output image as a copy of the input
     out = immap (out_image, NEW_COPY, in)

     one_l = 1

     # Initialize position vectors to
     # line 1, column 1, band 1 ...
     call amovkl (one_l, v1, IM_MAXDIM)
     call amovkl (one_l, v2, IM_MAXDIM)

     # Find the line size
     npix = IM_LEN(in,1)

     while (imgnlr (in, l1, v1) != EOF &&
        impnlr (out, l2, v2) != EOF)
        # Copy the image.
        call amovr (Memr[l1], Memr[l2], npix)

     # Close the images
     call imunmap (in)
     call imunmap (out)
end
```

Example 2.13: Line by Line Image I/O.

General Sections

These procedures return a pointer to dynamically allocated memory containing the pixels from an arbitrary section of an image. Note the difference from line-by-line I/O, in which the returned memory always represents a single line of an image, regardless of the dimensionality. These procedures may return a multi-dimensional section.

Procedure Call	***Purpose***
`bpt = imgs1`*`T`* `(imp, x1, x2)`	Get a section of a 1-D image
`bpt = imgs2`*`T`* `(imp, x1, x2, y1, y2)`	Get a section of a 2-D image
`bpt = imgs3`*`T`* `(imp, x1, x2, y1, y2, z1, z2)`	Get a section of a 3-D image
`bpt = imps1`*`T`* `(imp, x1, x2)`	Put a section of a 1-D image
`bpt = imps2`*`T`* `(imp, x1, x2, y1, y2)`	Put a section of a 2-D image
`bpt = imps3`*`T`* `(imp, x1, x2, y1, y2, z1, z2)`	Put a section of a 3-D image
`bpt = imggs`*`T`* `(imp, vs, ve, ndim)`	Get a general section

Table 2.13: Image Section Memory I/O Functions.

All of the above procedures are implemented for the usual SPP numeric data types: `short`, `int`, `long`, `real`, `double`, and `complex`. `imggs`*T*() differs from the other procedures in that the same arguments may be used for images of any dimension. The vectors `vs` and `ve` describe the range of elements in the section.

Miscellaneous Procedures

There are a few additional procedures providing miscellaneous capabilities.

Function Call		*Purpose*
`imflush`	`(imp)`	Flush the output buffer
`imaccess`	`(image, acmode)`	Test availability of image
`imcopy`	`(input, output)`	Copy images (does not work on OIF files)
`imdelete`	`(image)`	Delete the image
`imrename`	`(oldname, newname)`	Rename the image
`imgsection`	`(imagef, section, maxch)`	Get the image section field
`imgimage`	`(imspec, image, maxch)`	Get the image name
`imgcluster`	`(imspec, cluster, maxch)`	Get the cluster name

Table 2.14: Miscellaneous Image I/O Functions.

The last three procedures parse a fully qualified image filename into its components. The terms *image*, *section*, and *cluster* refer to separate fragments of a fully qualified image name. The image section is a string enclosed by square brackets specifying some subraster of an image, for example, `[100:125,200:450]`. The image name is the filename and group member number (applicable to STF images) without the image section, and the cluster is the filename only. Example 2.14 should clarify this nomenclature. Image sections will be explained in greater detail (See "Image Sections" on page 74.)

```
include <imhdr.h>

pointer im
char    imspec[SZ_FNAME], image[SZ_FNAME], image_clus[SZ_FNAME]
pointer immap()

begin
     call strcpy ("w00xh902t.c0h[1/125[100:125,200:450]",
        imspec, SZ_FNAME

     # Extract the image name
     call imgimage (imspec, image, SZ_FNAME)

     # The image string will have "w00xh902t.c0h[1/125]"

     # Extract the cluster name
     call imgcluster (imspec, image_clust, SZ_FNAME)

     # The image_clus string will contain: "w00xh902t.c0h"

     im = immap (imspec, READ_ONLY, 0)

     # To get the values 1 and 125 in the string "[1/125]"
     # use the following macros (after opening the image
     # as above).

     cl_index = IM_CLINDEX(im)  # value 1
     cl_size  = IM_CLSIZE(im)   # value 125
```

Example 2.14: Using Image Section Syntax.

Note that `imacces()` tests only whether an image name is valid, not if the image exists. However, if the image includes an image section, then `imacces()` will test for its existence.

Header Parameters

Image headers describe the format of an image and permit arbitrary parameters to be carried with the pixel data. The image database interface is the **imio** interface to the database containing the image headers. The first, fixed format, part of the image header contains the standard fields in binary and is fixed in size. This is followed by the *user area*, a string buffer containing a sequence of variable length, newline delimited FITS format *keyword=value* header cards. When an image is opened a large user area is allocated to permit the addition of new parameters without filling up the buffer. When the header is subsequently updated on disk only as much disk space is used as is needed to store the actual header.

Images comprise keyword parameters in an image header in addition to the pixel values. These header keywords describe the fundamental properties of the image such as its size and data type. In addition, they represent other pertinent information such as the instrument, date, world coordinate transformation, or any other data thought useful by the originator of the data. See “Standard Fields” on page 72 for an explanation of the standard parameters available for every image.

Procedure Call			***Purpose***
`value =`	`imget`*T*	`(imp, keyword)`	Get a header parameter
	`imgstr`	`(imp, keyword, outstr, maxch)`	Get a string parameter
	`imput`*T*	`(imp, keyword, value)`	Put a header parameter
	`impstr`	`(imp, keyword, value)`	Put a string parameter
	`imadd`*T*	`(imp, keyword, default)`	Add a header parameter
	`imastr`	`(imp, keyword, default)`	Add a string parameter
	`imaddf`	`(imp, keyword, default)`	Add a keyword with no value
	`imdelf`	`(imp, keyword)`	Delete a parameter
`istat =`	`imaccf`	`(imp, keyword)`	Test if parameter exists
`itype =`	`imgftype`	`(imp, keyword)`	Return datatype of parameter

Table 2.15: Image Header Parameter Functions.

In each procedure, the name of the parameter is specified as a character string (keyword here), sometimes referred to as a *field*. The procedures `imget`*T*`()`, `imput`*T*`()`, and `imadd`*T*`()` are implemented for the SPP data types `bool`, `char`, `short`, `int`, `long`, `real`, and `double`. The argument imp is a pointer type reference to the image returned by `immap()`.

New parameters will typically be added to the image header with either one of the typed `imadd()` procedures or with the lower level `imaddf()` procedure. The former procedures permit the parameter to be created and the value initialized all in one call, while the latter only creates the parameter. In addition, the typed `imadd()` procedures may be used to update the values of existing parameters, i.e., it is not considered an error if the parameter already exists. The principal limitation of the typed procedures is that they may only be used to add or set parameters of a standard data type.

The value of any parameter may be fetched with one of the `imgetT()` functions. Be careful not to confuse `imgets()` with `imgstr()` (or `imputs()` with `impstr()`) when fetching or storing the string value of a field. Fully automatic type conversion is provided. Any field may be read or written as a string, and the usual type conversions are permitted for the numeric data types.

The `imaccf()` function may be used to determine whether a field exists. Fields are deleted with `imdelf()`. It is an error to attempt to delete a nonexistent field. The following example (Example 2.15) illustrates handling of image header parameters. The character string field can take the name of any existing keyword in the image header, e.g., `DATE_OBS` or `i_naxis1`.

```
# Get the value of datatype and value of existing keywords
# and append new keywords to the image header with those values.

switch (imgftype (im, field)) {
case TY_BOOL:
     if (imgetb (im, field))
        O_VALB(o) = true
     else
        O_VALB(o) = false
     call strcpy ("NEW_BKY", nfield, SZ_KEYWORD)
     call imaddb (im, nfield, O_VALB(o))

case TY_CHAR:
     call imgstr (im, field, O_VALC(o), SZ_LINE)
     call strcpy ("NEW_SKY", nfield, SZ_KEYWORD)
     call imastr (im, nfield, O_VALI(o))

case TY_ING:
     O_VALI(o) = imgeti (im, field)
     call strcpy ("NEW_IKY", nfield, SZ_KEYWORD)
     call imaddi (im, nfield, O_VALI(o))

case TY_REAL:
     O_VALR(o) = imgetr (im, field)
     call strcpy ("NEW_RKY", nfield, SZ_KEYWORD)
     call imaddr (im, nfield, O_VALR(o))

default:
     call error (1, "unknown expression datatype")
}
```

Example 2.15: Handling Image Header Parameters.

Procedure Call	*Purpose*
`list   = imofnls (imp, template)`	Open a sorted file template
`list   = imofnlu (imp, template)`	Open unsorted file template
`nchars = imgnfn (list, fieldname, maxch)`	Get next filename
`imcfnl (list)`	Close template

Table 2.16: Image File I/O Functions Handling Templates.

The field name list procedures `imofnl[su]()`, `imgnfn()`, and `imcfnl()` procedures are similar to the fio file template facilities, except that the `@file` notation is not supported. The template is expanded upon an image header rather than a directory. Unsorted lists are the most useful for image header fields. If sorting is enabled each comma delimited pattern in the template is sorted separately, rather than globally sorting the entire template after expansion. Minimum match is permitted when expanding the template, another difference from file templates. Only actual, full length field names are placed in the output list.

Standard Fields

The **imio** database interface, described above, may be used to access any field of the image header, including the *standard fields* shown in Table 2.17, existing for every image. In addition, there may be other parameters unique to the particular image.

Keyword	*Type*	*Declaration*
`i_ctime`	`long`	Time of image creation
`i_history`	`string`	History string buffer
`i_limtime`	`long`	Time when limits (minmax) were last updated
`i_maxpixval`	`real`	Maximum pixel value
`i_minpixval`	`real`	Minimum pixel value
`i_mtime`	`long`	Time of last modify
`i_naxis`	`int`	Number of axes (dimensionality)
`i_naxisN`	`long`	Length of axis n (`i_naxis1`, etc.)
`i_pixfile`	`string`	Pixel storage filename
`i_pixtype`	`int`	Pixel datatype (SPP integer code)
`i_title`	`string`	Title string

Table 2.17: Standard Header Keywords.

The names of the standard fields share an `i_` prefix to reduce the possibility of collisions with user field names, to identify the standard fields in sorted listings, to allow use of pattern matching to discriminate between the standard fields and user fields, and so on. The `i_` prefix may be omitted provided the resultant name does not match the name of a user parameter. It is however recommended that the full name be used in all applications software.

You will need to use the include file `<imhdr.h>` when dealing with image headers. This defines macros for standard image header parameters dealing with fundamental characteristics of the image such as the size, data type, etc. Several header parameters are available via the **imio** structure defined by `<imhdr.h>`. Others may be accessed through the **imio** database procedures. Parameters may be read or written. If a parameter does not exist, it must be created. Example 2.16 is a fragment of code that finds the size of the image, the number of pixels per line and the number of lines. Since the keyword values in Table 2.17 are accessible through the `<imhdr.h>` structure, they can be used to get keyword values from an image using the **hedit** task.

```
include <imhdr.h>
char    image[SZ_FNAME]
pointer im
int     npts, nrow
pointer immap()
begin
     # Open the image
     im = immap (image, READ_ONLY, 0)
     # Find the image size
     npts = IM_LEN(im,1)
     nrow = IM_LEN(im,2)
     .
     .
end
```

Example 2.16: Using Header Parameters.

Image Sections

A fundamental feature of **imio** is the capability to treat a subset of an image identically to an entire image. The image filename as passed to `immap()` may include an *image section* which specifies what part of the image to read. The image section facility greatly increases the flexibility of the imio interface. Image sections are specified as part of the image name input to `immap()`, and are not visible to the applications program, which sees a somewhat smaller image, or an image of lesser dimensionality. Some examples are shown below. In addition, see “World Coordinates — mwcs” on page 129 describing the **mwcs** world coordinate system library.

Section	*Refers to...*
`pix[]`	The whole image
`pix[i,j]`	The single pixel value (scalar) at [i,j]
`pix[*.*]`	The whole image, two dimensions
`pix[*,-*]`	Flip Y-axis
`pix[*,*,b]`	B and B of 3-D image
`pix[*,*:s]`	Subsample in Y by S
`pix[*,1]`	Line l of image
`pix[c,*]`	Column c of image
`pix[i1:i2,j1:j2]`	Subraster of image
`pix[i1:i2:sx,j1:j2:sy]`	Subraster with sampling

Table 2.18: Image Section Syntax.

Image Name Templates

The filename template package of procedures permits the use of wildcards or nested lists of image filenames. The functionality and calling sequences are similar to those of the **fio** filename template package (see “Filename Templates” on page 101).

An image template is expanded into a list of image names or image sections with `imtopen()`. The list is not globally sorted, however sublists generated by pattern matching are sorted before appending the sublist to the final list. The number of images or image sections in a list is given by `imtlen()`. Images are read sequentially from the list with `imtgetim()`, which returns `EOF` when the end of the list is reached. The list may be rewound with `imtrew()`. An image template list should be closed with `imtclose()` to return the buffers used to store the list and its descriptor.

Procedure Call	***Purpose***
`list     = imtopen (template)`	Open image template
`nimages = imtlen (list)`	Return number of images
`           imtrew (list)`	Rewind template list
`nchars  = imtgetim (list, fname, maxch)`	Get next image name
`           imtclose (list)`	Close template

Table 2.19: Image Template Functions.

Note that the `int` function `imgetim()` returns `EOF` upon attempting to read at the end of file. Otherwise, it returns the number of characters in the image name.

Example 2.17 is the top level procedure for the IRAF `images.imcopy` task in `images$imutil/t_imcopy.x`. It demonstrates handling image name templates. Some comments have been added to clarify the code.

```
include <imhdr.h>

# IMCOPY -- Copy image(s)
# The input images are given by an image template list.
# The output is either a matching list of images or a directory.
# The number of input images may be either one or match the number
# of output images. Image sections are allowed in the input
# images and are ignored in the output images. If the input and
# output image names are the same then the copy is performed to a
# temporary file which then replaces the input image.

procedure t_imcopy()

char  imtlist1[SZ_LINE]       # Input image list
char  imtlist2[SZ_LINE]       # Output image list
bool  verbose                 # Print operations?

char  image1[SZ_PATHNAME]     # Input image name
char  image2[SZ_PATHNAME]     # Output image name
char  dirname1[SZ_PATHNAME]   # Directory name
char  dirname2[SZ_PATHNAME]   # Directory name
int   list1, list2, root_len

int   imtopen(), imtgetim(), imtlen()
int   fnldir(), isdirectory()
bool  clgetb()
```

(Continued...)

Example 2.17: Handling Image Name Templates.

```
begin
         # Get input and output image template lists.

# Get the input image template from the task parameter "input",
# for example: input = "*.imh", input = "im1.imh,im2.imh,w0*.c0h"
     call clgstr ("input', imtlist1, SZ_LINE)

# Get the output image template from the task parameter "output",
# for example: output = $home/data/ or output = cimf.imh
     call clgstr ("output", imtlist2, SZ_LINE)
     verbose = clgetb ("verbose")

     # Check if output string is a directory
     if (isdirectory (imtlist2, dirname2, SZ_PATHNAME) > 0) {

        # When output = "$home/data/" then isdirectory is > 0
        # and dirname2 will have the output string.
        list1 = imtopen (imtlist1)
        while (imtgetim (list1, image1, SZ_PATHNAME) != EOF) {

           # imtopen will return a pointer to a list of files and
           # each occurrence of imgetim will put an image name in image1

           # Strip the image section first because fnldir
           # recognizes it as part of a directory. Place
           # the input image name without a directory or
           # image section in string dirname1
           call get_root (image1, image2, SZ_PATHNAME)
           root_len = fnldir (image2, dirname1, SZ_PATHNAME)
           call strcpy (image2[root_len+1], dirname1, SZ_PATHNAME)
           call strcpy (dirname2, image2, SZ_PATHNAME)
           call strcat (dirname1, image2, SZ_PATHNAME)
           call img_imcopy (image1, image2, verbose)
        }
        call imtclose (list1)
     } else {
        # Expand the input and output image lists
        list1 = imtopen (imtlist1)
        list2 = imtopen (imtlist2)
        if (imtlen (list1) != imtlen (list2)) {
           call imtclose (list1)
           call imtclose (list2)
           call error (0, "Number of ... images not the same")
        }
        # Do each set of input/output images
        while ((imtgetime (list1, image1, SZ_PATHNAME) != EOF) &&
           (imtgetim (list2, image2, SZ_PATHNAME) != EOF)) {
           call img_imcopy (image1, image2, verbose)
        }
        call imtclose (list1)
        call imtclose (list2)
     }
end
```

Example 2.17 (Continued): Handling Image Name Templates.

2.4 Formatted I/O — fmtio

SPP includes complete facilities for formatting numeric and text data for input, output, and internal use.

`printf` and its relatives

Text and binary numbers formatted as text may be directed to the standard output (`STDOUT`), the standard error stream (`STDERR`), a text file, or a string. Note that `STDOUT` may be redirected to a file or piped to another task in the IRAF cl. Binary values may be formatted via a format specification string. The values to format must be passed in separate procedure calls. The `printf()` family of procedures performs formatted output. These are similar to the C `<stdio>` library procedures except that the values to format are not included in the calling sequence because SPP (Fortran) does not handle variable numbers of calling arguments in a portable manner.

Procedure Call		***Purpose***
`printf`	`(format)`	Formatted print to `STDOUT`
`eprintf`	`(format)`	Formatted print to `STDERR`
`fprintf`	`(fd, format)`	Formatted print to any open file
`sprintf`	`(outstr, maxch, format)`	Formatted print to a string buffer
`clprintf`	`(param, format)`	Formatted print to a cl parameter
`parg`*`T`*	`(value)`	Pass a numeric argument to a `printf()`
`pargstr`	`(value)`	Pass string argument to a `printf()`

Table 2.20: Formatted Output Functions.

The values to format and print are passed via `parg`*`T`*`()` procedures. There is a separate procedure for each of the SPP data types: `bool`, `char`, `short`, `int`, `long`, `real`, `double`, and `complex`. For example, for numerical values, `pargr()` is used for floating point, `pargi()` for integer, while `pargstr()` would be used for strings. Note that the data type specified by the name of the procedure represents the data type of the parameter passed to the format, not the format itself. In general, any SPP data type variable may be formatted by any `printf()` format specification.

Format Codes

A format specification is a string that describes how values are to be represented in the output. The string may include any text, but fields may be included to format values. These fields have the form %*w.dCn*. Any text not preceded by a percent character will be written to the output unchanged. The percent character is a required part of the format field and the remainder of the word specifies the form of the output. *w* is the field width, *d* is the number of decimal places or the number of digits of precision, *C* is the format code, and *n* is radix character (for format code `r` only). The *w* and *d* fields are optional. The string may be a literal, a string variable, or a predefined parameter constant. Therefore, run-time formats are possible. The format codes *C* are shown in Table 2.21.

Code	***Format***
`b`	Boolean, true or false (`yes` or `no` only on output)
`c`	Single character, `c` or `\c` or `\0nnn`
`d`	Decimal integer
`e`	Exponential, d specifies the precision
`f`	Fixed format, d specifies the number of decimal places
`g`	General format, d specifies the precision
`h`	Sexagesimal, hh:mm:ss.ss, d is the number of decimal places
`m`	Minutes, seconds (or hours, minutes), mm:ss.ss
`o`	Octal integer
`r`*n*	Convert integer in any radix n
`s`	String, `d` field specifies max chars to print
`t`	Advance to column given as field w
`u`	Unsigned decimal integer
`w`	Output the number of spaces given by field w
`x`	Hexadecimal integer
`z`	Complex format `(r,r)`, d specifies the precision

Table 2.21: Output Format Codes.

The conventions for the *w* (field width) specification are as follows:

Code	***Effect***
`n`	Right justify in field of n characters, blank fill
`-n`	Left justify in field of n characters, blank fill
`0`*n*	Zero fill at left, only if right justified
absent	Use as much space as needed, d field sets precision
`0`	Use as much space as needed, d field sets precision

Table 2.22: Field Width Specifications.

Escape sequences (e.g., `\n` for newline) are replaced by the appropriate character value on output:

Escape	***Replacement Character***
`\b`	Backspace
`\f`	Form feed
`\n`	Newline (LF)
`\r`	Carriage return
`\t`	Tab
`\"`	String delimiter character
`\'`	Character constant delimiter character
`\\`	Backslash character
`\`*nnn*	Octal value of character
%%	Insert a percent character in the output

Table 2.23: Escape Sequences.

Note that a newline is not automatically written for every `printf()` call, as with a Fortran `WRITE`. Use `\n` in the format text to explicitly write a newline. (See Example 2.18).

```
mean = 4027.123
sigma = 33.98423

call printf ("mean: %06g sigma: %6.2f\n")
call pargr (mean)
call pargr (sigma)
```

Output Produced...

```
mean: 4027.12  sigma: 33.98
```

Example 2.18: Writing a Newline.

Additional Output Procedures

Substituting `eprintf()` for `printf()` would write to the standard error stream `STDERR` instead of standard output. These two streams are treated separately by the cl. To write to an arbitrary text file, use `fprintf()`, specifying a file descriptor for an open text file, see Example 2.19.

```
.
.
char   filename[SZ_FNAME]     # Output text file name
int    ival
real   rval
int    fd
int    open()

begin
     .
     .
     # Open the output text file
     fd = open (filename, NEW_FILE, TEXT_FILE)

     # Write formatted output
     call fprintf (fd, "ival = %d, rval = %f\n")
        call pargi (iv)
        call pargr (rval)
     .
     .
```

Example 2.19: Writing an Arbitrary Text File.

Similarly, formatted text may be written to a text string variable using `sprintf()`. This is particularly useful for error messages or runtime formats, i.e., generating a format string to use in another `printf()` call. Note that `sprintf()` includes an argument specifying the maximum size of the output character string.

```
char  filename[SZ_FNAME]
char  outstr[SZ_LINE]      # String taking formatted output
char  fmtstr[SZ_LINE]      # Format string
int   ival
real  rval
begin
     # Write formatted output
     call sprintf (outstr, SZ_LINE, "ival = %d, rval = %f\n")
        call pargi (ival)
        call pargr (rval)
     # Write the string to output
     call printf (outstr)
     # Write the output string
     call sprintf (outstr, SZ_LINE, "Couldn't open file %s\n")
        call pargstr (filename)
     call error (0, outstr)
     # Get a format string from the cl
     call clgstr ("format", fmtstr, SZ_LINE)
     call printf (fmtstr)
        call pargi (ival)
     .
     .
```

Example 2.20: Writing Output to a Text String Using `sprintf()`.

Formatted Input — `scan`, et. al.

Formatted input may be read from the standard input stream `STDIN`, a text file, a string variable, or a cl parameter using the `scan` family of procedures. Each scan procedure returns an integer status as the function value. This status will contain `EOF` upon reading end of file.

Procedure Call	***Purpose***
`scan ()`	Scan from `STDIN`
`stat = fscan (fd)`	Scan from file opened as `fd`
`stat = sscan (str)`	Scan from the string `str`
`stat = clscan (param)`	Scan from the cl parameter `param`
`scanc (ch)`	Get the next character from a scan
`reset_scan ()`	Rescan same input

Table 2.24: Formatted Input Functions.

Note that as with the output (`printf()` family) procedures, variables are not changed by the `scan()` procedures. Values read are placed in variables using the `gargT()` family of procedures.

Procedure Call	***Purpose***
`gargT (value)`	Get a typed argument
`gargstr (outstr, maxch)`	Get rest of line
`gargwrd (outstr, maxch)`	Get next "word"
`gargrad (lval, radix)`	Non-decimal `gargi()`
`gargtok (tok, outstr, maxch)`	Get next token

Table 2.25: Input Functions.

There is a separate `gargT()` procedure for each of the SPP data types: `bool`, `char`, `short`, `int`, `long`, `real`, `double`, and `complex`. A *word*, as recognized by `gargwrd()`, is any string separated by white space.

```
define DATA_SIZE    1024

procedure getfield (xcoord, ycoord, npts)

# Read 2 columns of numbers from STDIN to dynamically allocated
# arrays passed back via pointers

pointer xcoord, ycoord       # Coordinate arrays
int     npts                 # Number of points
int     row
char    inline[SZ_LINE]
int     getline()

begin
     # Default number of data values
     npts = DATA_SIZE
     # Allocate the arrays
     call malloc (xcoord, npts, TY_REAL)
     call malloc (ycoord, npts, TY_REAL)
     row = 0
     while (getline (STDIN, inline) != EOF) {
        row = row + 1       # Read an input file row
        if (row > npts) {
           npts = npts + DATA_SIZE    # No room -- Allocate more scratch space
           # Reallocate space to save allocated memory
           call realloc (xcoord, npts, TY_REAL)
           call realloc (ycoord, npts, TY_REAL)
        }
        # Put the values into the data arrays
        call gargr (Memr[xcoord+row-1])
        call gargr (Memr[ycoord+row-1])
        }
     }
     call sfree (sp)
     npts = row
     # Resize data buffers to match amount of data read
     call realloc (xcoord, npts, TY_REAL)
     call realloc (ycoord, npts, TY_REAL)
end
```

Example 2.21: Formatting Output.

Internal Formatting

These procedures convert a string representation of a number into its binary value. They perform the same function as the `garg...()` procedures, but do I/O internally. That is, they read from a character string variable, not an input stream or file. Each function may be called repeatedly to decode a string of values delimited by white space or embedded in non-numeric characters.

Procedure Call	***Purpose***
`nchar = cto`*`T`*` (str, ip, value)`	Convert string to binary (there is no `ctos()`
`nchar = cctoc (str, ip, char)`	`char` constant to char
`nchar = gctod (str, ip, dval)`	Convert any number to `double`
`nchar = gctox (str, ip, xval)`	Convert any number to `complex`
`nchar = gctol (str, ip, lval, radix)`	Variable radix
`nchar = ctowrd (str, ip, outstr, maxch)`	Word or string
`token = ctotok (str, ip, outstr, maxch)`	Extract token

Table 2.26: Internal Formatting Functions.

There is a separate `cto`*`T`*`()` procedure for each of the SPP numeric data types: `int`, `long`, `real`, `double`, and `complex`. All of the procedures except `ctotok()` return the number of non-white input characters converted as the integer function value.

`ctotok()` returns an integer code identifying the type of *token* returned. Tokens represent the smallest substrings recognized in the string. The values assigned to the token returned by `ctotok()` are defined in the include file `ctotok.h`.

While `ctowrd()` nominally recognizes *words* separated by white space, any string enclosed in quotes is treated as a single word.

The `dtoc()` *format* (see Table 2.27) is one of the characters `e`, `f`, `g`, `h`, or `m`. See “Format Codes” on page 79 for their meaning.

```
procedure parse (instr, first, second, third)
# Parse a string expected to contain three values, as:
# "[1,2,3]" or "4 5 6"

char  instr[ARB]            # Input string
int   first, second, third  # Values in string
int   ip
int   nchar
int   ctoi()

begin
     ip = 1     # Initialize the string pointer
     # Read the first field
     if (ctoi (instr, ip, first) == 0) {
        # Nothing there
        first = INDEFI
        return
     }
     # Read 2nd field, The string pointer ip is just after 1st number
     if (ctoi (instr, ip, second) == 0) {
        # Nothing there
        second = INDEFI
        return
     }
     # Read last field
     if (ctoi (instr, ip, third) == 0) {
        # Nothing there
        third = INDEFI
     }
end
```

Example 2.22: Using Internal Formatting Functions.

Procedure Call	***Purpose***
`nchar = itoc  (ival, outstr, maxch)`	`int` to `char`
`nchar = ltoc  (lval, outstr, maxch)`	`long` to `char`
`nchar = ctocc (char, outstr, maxch)`	`char` to `char` constant
`nchar = gltoc (lval, outstr, maxch, radix)`	Generic `long`
`nchar = xtoc  (xval, outstr, maxch, decpl, format, width)`	`complex` to `char`
`nchar = dtoc  (dval, outstr, maxch, decpl, format, width)`	`double` to `char`

Table 2.27: Conversion Functions.

Character and String Functions

SPP characters are implemented as integers. Character strings are implemented as fixed length arrays of characters (integers) with the element following the last character set to zero to indicate the end of the string. Therefore they cannot be treated simply as scalar variables in assignment statements. There is a family of procedures for assigning and otherwise manipulating strings. The `chr...()` family of functions convert a single character (type `char`) to upper or lower case. The converted character is returned as the function value.

Procedure Call	***Purpose***
`ch = chrupr (ch)`	Change character to upper case
`ch = chrlwr (ch)`	Change character to lower case

Table 2.28: Character Case Conversion Functions.

Note that there are macro definitions to accomplish the same purpose. The macro `TO_UPPER()` converts a single character to upper case and `TO_LOWER()` converts a character to lower case. However, these assume that the character is already the appropriate case. These macros are defined in `<ctype.h>`. The `str...()` family of procedures deal with character strings (`char` arrays).

Procedure Call	***Purpose***
`nchar = gstrcat (str, outstr, maxch)`	Returns length of output string
`strcat (str, outstr, maxch)`	Concatenate `str` to `outstr`
`nchar = gstrcpy (from, to, maxch)`	Returns length of output string
`strcpy (from, to, maxch)`	Copy `EOS` delim string
`nchar = strlen (str)`	Length of string (excluding `EOS`)
`strlwr (str)`	Convert string to lower case
`strupr (str)`	Convert string to upper case

Table 2.29: Basic String Functions.

Note that `strlen()` returns the number of characters actually occupying the string, not including the EOS character but including any blanks, not the declared size. This is different from the Fortran `len` function, which returns the declared size of a string, implicitly padded with blanks to the declared size.

Procedure Call	*Purpose*
`index = stridx (char, str)`	First index of character in string
`index = stridxs (set, str)`	Return the index of the first occurrence of any of a set of characters in a string
`index = strldx (char, str)`	Last index of character in string
`index = strldxs (set, str)`	Return the index of the last occurrence of any of a set of characters in a string

Table 2.30: String Index Functions.

Note that the argument `char` in `stridx()` and `strldx()` is not a string (a double quoted literal or char array) but an integer representing a single character. If it's a literal, it should be in *single* quotes. Otherwise, it should be a scalar `char` variable.

Procedure Call	*Purpose*
`index = strdic (instr, outstr, maxch, dict)`	Search a dictionary string for a match with an input string
`nchar = strmac (macro, argstr, outstr, maxch)`	Expand a macro by string substitution
`int   = strsrt (x, sb, nstr)`	Sort a list of strings
`strtbl (fd, buf, strp, nstr, first_col,last_col, maxch, ncol)`	Print a list of strings.

Table 2.31: Complex String Functions.

Note that macro expansion in `strmac()` is not recursive.

String Comparisons

Procedure Call	***Purpose***
`index  = strcmp (str1, str2)`	Compare two strings.
`bool   = str`*`OP`*` (s1, s2)`	Is s1 *OP* s2? (see below)
`-1,0,1 = strncmp (s1, s2, n)`	Counted comparison
`nextch = strsearch (str, patstr)`	Fast substring search
`nextch = strmatch (str, patstr)`	Match strings using metacharacters
`nextch = gstrmatch (str, patstr, first, last)`	Generalized pattern matching
`bool   = streq (str1, str2)`	$s_1 == s_2$
`bool   = strne (str1, str2)`	$s_1 != s_2$
`bool   = strlt (str1, str2)`	$s_1 < s_2$
`bool   = strgt (str1, str2)`	$s_1 > s_2$
`bool   = strle (str1, str2)`	$s_1 <= s_2$
`bool   = strge (str1, str2)`	$s_1 >= s_2$

Table 2.32: String Comparison Functions.

The `strcmp()` procedure returns -*n* if $s_1 < s_2$, 0 if $s_1 = s_2$, and +*n* if $s_1 > s_2$. The `bool` procedure `strop()` determines whether two strings satisfy a logical operation. The function is selected by replacing `op` with an operator from the list.

For example, to test whether strings are equal, use streq(). Pattern matching characters or *metacharacters* are defined in the include file <pattern.h>:

Procedure	***Metacharacter***	***Purpose***
CH_BOL	^	Beginning of line
CH_NOT	^	Not, in character classes
CH_EOL	$	End of line symbol
CH_ANY	?	Match any single character
CH_CLOSURE	*	Zero or more occurrences
CH_CCL	[	Begin character class
CH_CCLEND	]	End character class
CH_RANGE	-	Range, as in [a-z]
CH_ESCAPE	\\	Escape character
CH_WHITESPACE	#	Match optional white space
CH_IGNORECASE	{	Begin ignoring case
CH_MATCHCASE	}	Begin checking case

Table 2.33: Pattern Matching Metacharacters.

Evaluating Expressions — evexpr

The evexpr() procedure is a function which takes an algebraic expression as input, evaluates the expression, and returns the value of the expression as the function value.

Procedure Call	***Purpose***
opt = evexpr (expr, getop_epa, ufcn_epa)	Evaluate expression

Table 2.34: Evaluating Expressions.

The input expression is a character string. It is parsed and reduced to a single value. The operands to the expression may be either constants or identifiers (strings). If an identifier is encountered the user supplied get operand procedure is called to return the value of the operand. Operands are described by the operand structure, and operands are passed about by a pointer to such a structure. The value of the expression is returned as a pointer to an operand structure containing the function value. Operands of different data types may be mixed in an expression with the usual automatic type coercion rules. All SPP data types are supported including strings (`char` arrays). All SPP operators and intrinsic functions are recognized. (See "Intrinsic Functions" on page 35).

Output is a pointer to an operand structure containing the computed value of the expression. The output operand structure is dynamically allocated by `evexpr()` and must be freed explicitly by the user with `mfree()`.

Note that the second and third arguments are the `int` entry point addresses of procedures. The function `locpr()` is used to return the address of a function. If there is no function supplied, use `NULL` for the address. A generic example is:

```
op = evexpr (expr, locpr(getop), locpr(ufcn))
```

with the user-supplied procedures having the calling sequences shown in Table 2.35:

Procedure Call	*Purpose*
`getop (identifier, op)`	Return named operand's value
`ufcn (fcn, args, nargs, op)`	Return named function's value

Table 2.35: Calling User-Supplied Procedures.

If a syntax error occurs while parsing the expression `evexpr()` will take the error action *syntax error*. The `NULL` arguments could be replaced by the `locpr()` addresses of get operand and/or user function procedures if required by the application.

The lexical form of the input expression is the same as that of SPP and the cl for all numeric, character, and string constants and operators. Any other sequence of characters is considered an identifier and will be passed to the user supplied get operand function to be turned into an operand.

This procedure requires the include file `<evexpr.h>` that defines the operand structure. The operand structure is used to represent all operands in expressions and on the parser stack. Operands are passed to and from the outside world by means of a pointer to an operand structure. The caller is responsible for string storage of string operands passed to `evexpr()`. `evexpr()` manages string storage for temporary string operands created during expression evaluation, as well as storage for the final string value if the expression is string valued. In the latter case the value string should be used before `evexpr()` is called again.

Calling Procedure	*Returned Data Type*
`O_TYPE(op)`	Operand data type
`O_VALB(op)`	Boolean value
`O_VALI(op)`	Integer value (or string pointer)
`O_VALR(op)`	Real value
`O_VALC(op)`	String value

Table 2.36: Evaluating Procedure Data Types.

The following simple example (Example 2.23) evaluates a constant expression and prints the value on the standard output. An only slightly more complicated example (Example 2.24) uses the procedure `get_op()` to return an operand value.

```
include     <evexpr.h>
pointer     op, evexpr()

begin
     # Evaluate an expression
     op = evexpr ("sin(.5)**2 + cos(.5)**2)", NULL, NULL)

     # Print the result of the operation
     switch (O_TYPE(op)) {
     case TY_INT:
        call printf ("result = %d\n")
           call pargi (O_VALI(op))
     case TY_REAL:
        call printf ("result = %g\n")
           call pargr (O_VALR(op))
     case TY_CHAR:
        call printf ("result = %s\n")
           call pargstr (O_VALC(op))
     }

     # Free the operand structure memory
     call mfree (op, TY_STRUCT)
```

Example 2.23: Evaluating Data Types.

```
include <evexpr.h>

real procedure evalu8 (expr)

pointer igps
char    expr[ARB]
pointer op
int     npts
extern  get_op()
pointer evexpr()
int     locpr()

begin
     op = evexpr (expr, locpr(get_op), 0) # Evaluate expression
     switch (O_TYPE(op)) {
     case TY_REAL:
        return (LOP_VALR(op))
     case TY_INT:
        return (LOP_VALI(op))
     }
     call mfree (op, TY_STRUCT)
end


procedure get_op (operand, op)


# Assigns value to expression operand. Allowed operands are x and y.
# Values are taken from the common /evopcom/.


char     operand[ARB]     # operand name
pointer   op                # operand (output)
common    /evopcom/ x, y

begin
    # Set up operand structure (zero length ==> scalar)
    call xev_initop (op, O, TY_REAL)
    switch (operand[1]) {
    case 'x', 'X':          # Allow either case operand
       LOP_VALR(op) = x       # Assign a real-valued operand
    case 'y', 'Y':
       LOP_VALR(op) = y
    }
    # Free operand structure memory
    call mfree (op, TY_STRUCT)
end
```

Example 2.24: Returning Operand Value.

2.5 File I/O — fio

File I/O takes place using a *stream*, that is, an I/O channel available to the SPP program. The standard streams, referred to as `STDIN`, `STDOUT`, and `STDERR` (macros for integer values specifying a stream), are always open. That is, you need not call `open()` to access them. `STDIN` and `STDOUT` read from and write to the user terminal when working interactively but may be redirected or piped. `STDERR` is for warning or error messages. The fio library permits input from and output to binary or text files.

Before any I/O can be done on a file, the file must be opened. The `open()` procedure may be used to access ordinary files containing either text or binary data. To access a file on one of the special devices such as magnetic tape, a special open procedure must be used. To conserve resources (file descriptors, buffer space) a file should be closed when no longer needed. Any file buffers that may have been created and written into will be flushed before being deallocated. `close()` ignores any attempts to close `STDIN`. Attempts to close `STDOUT`, or `STDERR` cause the respective output byte stream to be flushed, but are otherwise ignored. An error results if one attempts to close a file that is not open. File I/O functions are listed in Table 2.37; if you are working with binary data, Table 2.42, “Binary File I/O Functions.,” on page 98 lists additional functions.

Procedure Call	***Purpose***
`fd = open (fname, mode, type)`	Open or create a text or binary file
`close (fd)`	Close a file
`flush (fd)`	Flush any buffered output to a file
`seek (fd, loffset)`	Set the file offset of the next char to be read or written
`long = note (fd)`	Note the position in file for later seek

Table 2.37: File I/O Functions.

The access modes (the mode argument to `open()`) are:

Access Mode	*Definition*
`READ_ONLY`	Read-only from an existing file
`WRITE_ONLY`	Write-only to an existing file
`READ_WRITE`	Read from or write to an existing file
`APPEND`	Write to the end of an existing file
`NEW_FILE`	Create a new file
`TEMP_FILE`	Temporary file; delete upon task completion

Table 2.38: File Access Modes.

The file types (the `type` argument to `open()`) are:

File Type	*Definition*
`TEXT_FILE`	File of lines of text
`BINARY_FILE`	Buffered binary byte stream
`SPOOL_FILE`	In-memory “file”

Table 2.39: File Types.

Procedure Call	*Purpose*
`fseti (fd, param, value)`	Set integer fio options
`value = fstati (fd, param)`	Get the value of an integer fio parameter
`value = fstatl (fd, param)`	Get value of a long integer fio parameter
`fstats (fd, param, outstr, maxch)`	Get a string valued fio parameter
`stat = finfo (fname, ostruct)`	Get directory information on a file
`stat = access (fname, mode, type)`	Determine the type or accessibility of a file
`delete (fname)`	Delete a file
`rename (old_fname, new_fname)`	Change the name of a file
`mktemp (root, fname, maxchars)`	Make a unique temporary filename
`falloc (fname, nchars)`	Preallocate file space
`stat = protect (fname, action)`	Protect a file from deletion
`fcopy (from_fname, to_fname)`	Copy a file
`fcopyo (from_fd, to_fd)`	Copy open files

Table 2.40: File Manipulation Commands

In the above procedures, the common calling sequence variables are declared as follows:

Variable Name	*Contents*
`int fd`	File descriptor
`char fname[SZ_FNAME]`	Filename string

Table 2.41: File Variables.

Any file may be accessed after specifying only the filename, access mode, and file type parameters using the `open()` call. Occasionally, however, it is desirable to change the default file control parameters, to optimize I/O to the file. The `fset()` procedure is used to set the FIO parameters for a particular file, while `fget()` is used to inspect the values of these parameters. The special value `DEFAULT` will restore the default value of the indicated parameter. The procedure `seek()` is used to move the file pointer (offset in a file at which the next data transfer will occur). With text files, one can only seek to the start of a line, the position of which must have been determined by a prior call to `note()`. For binary files, `seek()` merely sets the logical offset within the file. The logical offset is the character offset in the file at which the next I/O transfer will occur. In general, there is no simple relationship between the logical offset and the actual physical offset in the file.

Binary File I/O

The minimum size addressable SPP data item is a character, usually implemented as a `short` (two byte) integer. Therefore, in binary file I/O, the size of the buffer is specified in units of `char`s, or `short`s. It is possible to pack bit and byte data into `char`s. See the **osb** procedures described in “Bit & Byte Operations — osb” on page 123.

Procedure call	*Purpose*
`stat = read (fd, buffer, nch)`	Read a binary block of data from a file
`write (fd, buffer, nch)`	Write a binary block of data to a file

Table 2.42: Binary File I/O Functions.

The `read()` procedure reads a maximum of `nch` characters from the file with descriptor `fd` into the user supplied memory buffer. The following example (Example 2.25) illustrates reading a binary file and extracting values. This is a straightforward example because all of the desired values are short integers at the beginning of the file.

```
procedure alghead (alfn, nx, xoff, yoff, nbit)

# ALGHEAD -- Read header parameters from binary alias file. These
# are width & height of image, offsets in x & y, and number bits/pixel.

char    alfn[ARB]        # Alias file name
pointer al               # File descriptor
int     nx, ny           # Image size
int     xoff, yoff       # Offsets

int     nbit             # Bits per pixel
pointer al               # Alias file descriptor
short   sval[5]          # Header
int     status           # Return status

begin
     # Open the binary input alias file
     al = open (alfn, READ_ONLY, BINARY_FILE)

     # Read the 5 (short) word header

     status = read (al, sval, 5)
     # Parse header
     nx = sval[1]
     ny = sval[2]
     xoff = sval[3]
     yoff = sval[4]
     nbit = sval[5]
end
```

Example 2.25: Reading Values From a Binary File.

The next slightly more complicated example () demonstrates extracting individual bytes from a binary file. The fragment of code reads a single word consisting of four bytes and assigns the individual byte values to separate short integers using the **osb** `bytmov()` procedure.

```
# Read a word from the Alias file
status = read (al, albuf, 2)
run = 0    # Run length
call bytmov (albuf, 1, run, 4, 1)
# The color values
call bytmov (albuf, 4, rv, 2, 1)
call bytmov (albuf, 3, gv, 2, 1)
call bytmov (albuf, 2, bv, 2, 1)
```

Example 2.26: Extracting Bytes From a Binary File.

Text Character I/O

The procedures `getc()` and `putc()` read and write character data, a single character at a time.

Procedure Call	*Purpose*
`stat = getc    (fd, char)`	Get a `char` from a file
`       putc    (fd, char)`	Put `char` to a file
`       putcc   (fd, char)`	Handles unprintable characters
`stat = getchar (char)`	Get char from `STDIN`
`       putchar (char)`	Put char to `STDOUT`
`stat = getline  (fd, linebuf)`	Get a line of text
`stat = getlline (fd, linebuf, maxch)`	Get a line of text
`       putline  (fd, linebuf)`	Output a string to `fd`

Table 2.43: Text Character I/O Operations.

Note that `getchar()` and `putchar()` deal with `STDIN` and `STDOUT` respectively so they don't require a file descriptor. The other procedures require a previous call to `open()` or may specify one of the standard streams `STDIN`, `STDOUT`, or `STDERR`. The newline character is returned as part of a line read by `getline()`. The maximum size of a line (size of a line buffer) is set at compile time by the system wide constant `SZ_LINE`. `getline()` reads at most `SZ_LINE` characters. To read more in one call, use `getlline()` which includes an argument specifying how many characters to read.

Pushback

Characters and strings (and even binary data) may be *pushed back* into the input stream. `ungetc()` pushes a single character. Subsequent calls to `getc()`, `getline()`, `read()`, etc. will read out the characters in the order in which they were pushed (first in, first out). When all of the pushback data have been read, reading resumes at the preceding file position, which may either be in one of the primary buffers, or an earlier state in the pushback buffer.

Procedure Call	***Purpose***
`ungetc (fd, char)`	Push back a char
`ungetline (fd, string)`	Push back a string
`unread (fd, buf, nchars)`	Push back binary data

Table 2.44: Pushback Text Functions.

`ungets()` differs from `ungetc()` in that it pushes back whole strings, in a last in, first out fashion. `ungets()` is used to implement recursive macro expansions. The amount of recursion permitted may be specified after the file is opened, and before any data are pushed back. Recursion is limited by the size of the input pointer stack, and pushback capacity by the size of the pushback buffer.

Filename Templates

The filename template package contains routines to expand a filename template string into a list of filenames, and to access the individual elements of the list. It is primarily a convenience for users to allow wildcards in filenames and pointers to files containing lists of names. The template is a list of filenames, patterns, or list filenames. The concatenation operator (`//`) may be used within input list elements to form new output filenames. String substitution may also be used to form new filenames.

A sample template string is:

```
alpha, *.x, data* // .pix, [a-m]*, @list_file
```

This template would be expanded as the file `alpha`, followed in successive calls by all the files in the current directory whose names end in `.x`, followed by all files whose names begin with `data` with the extension `.pix` appended, and so on. The `@` character signifies a list file. That is, a file containing regular filenames.

String substitution uses the first string given for the template, expands the template, and for each filename generated by the template, substitutes the second string to generate a new filename. Some examples follow.

Sample String	*Performs Function*
`*.%x%y`	Change the extension to `y`
`*%%_abc%.imh`	Append `_abc` to `root`
`nite%1%2%.1024.imh`	Change `nite1` to `nite2`

Table 2.45: String Substitution Characters.

The following procedures (with a `b` suffix) are the highest level and most convenient to use.

Procedure Call	*Purpose*
`fntopnb (template, sort)`	Expand template and open a buffered filename list
`status = fntgfnb (list, fname, maxch)`	Get next filename from buffered list (sequential)
`status = fntrfnb (list, index, fname, maxch)`	Get next filename from buffered list (random)
`fntclsb (list)`	Close buffered list
`num = fntlenb (list)`	Get number of filenames in a buffered list
`fntrewb (list)`	Rewind the list

Table 2.46: High-Level Template Functions.

The remaining lower level routines expand a template on the fly and do not permit sorting or determination of the length of the list.

Procedure Call	*Purpose*
`fntopn (template)`	Open an unbuffered filename list
`fntgfn (pp, outstr, maxch)`	Get next filename from unbuffered list
`fntcls (pp)`	Close unbuffered list

Table 2.47: Low-Level Template Routines.

2.6 Vector (Array) Operators — vops

The vector operator (**vops**) procedures implement common operators for arrays of most supported SPP data types. They are *host-specific* in the sense that they may take advantage of specialized hardware and software available on a particular system such as vector processors and vectorizing compilers. This would substantially improve the performance of computationally intensive tasks dealing with large arrays such as images. Nevertheless, the interface to SPP (the calling sequence) is independent of the underlying architecture.

Each section below describes a family of **vops** *operators* related by functionality. Each operator (procedure) is implemented with the same root name and calling sequence for several data types. However, not all operators are implemented (nor do they make sense) for every data type. The tables list the root procedure name, implemented data types, calling sequence, and description of the operation. All of the functions require an `int` argument that specifies the number of elements in the passed vector or vectors. If the procedure requires more than one vector, they are assumed to have the same number of elements. In nearly every case, multiple array arguments to **vops** procedures are also the same data type. A significant exception is `acht`*TT*`()`, which converts a vector of one data type to another vector of a different data type.

All vector operations may be performed *in place*. That is, the same array may be used on input as well as output. An array passed to a vector procedure need not be one-dimensional. In all cases, the vectors are treated simply as contiguous words. Since there is assumed to be no functional relationship among the pixel positions in the vectors, arrays of any dimensionality may be passed. Only the total number of pixels in the array need be passed to the **vops** procedure. Many procedures are implemented for the case of two vectors or a vector and a scalar. In the latter case, the procedure name has a `k` inserted before the last character (the initial of the data type) and one argument must be a constant or scalar variable.

Arithmetic Operators

These procedures implement basic arithmetic operations. The binary operators (add, subtract, multiply, and divide) include operations between two vectors or between a vector and a scalar. In the former case, each element of the output vector is the result of the operation on the corresponding elements of the input vectors. In the second case, each element of the output vector represents the result of the operation between the corresponding element of the input vector and the same scalar.

Procedure Call	***Purpose***
`aneg`*T* `(a, b, npix)`	Negate a vector $b_i = a_i$
`aadd`*T* `(a, b, c, npix)`	Add two vectors $c_i = a_i + b_i$
`aaddk`*T* `(a, k, c, npix)`	Add a vector and a scalar $c_i = a_i + k$
`asub`*T* `(a, b, c, npix)`	Subtract two vectors $c_i = a_i - b_i$
`asubk`*T* `(a, k, c, npix)`	Subtract a scalar from a vector $c_i = a_i - k$
`amul`*T* `(a, b, c, npix)`	Multiply two vectors $c_i = a_i b_i$
`amulk`*T* `(a, k, c, npix)`	Multiply a vector and a scalar $c_i = a_i k$
`adiv`*T* `(a, b, c, npix)`	Divide two vectors $c_i = a_i / b_i$
`adivk`*T* `(a, k, c, npix)`	Divide a vector by a scalar $c_i = a_i / k$
`advz`*T* `(a, b, c, npix, errfcn)`	Vector divide, detect divide by zero $c_i = a_i / b_i$

Table 2.48: Arithmetic Functions.

Each of these procedures is implemented for the following data types: `short`, `int`, `long`, `real`, `double`, and `complex`. To use the appropriate data type, replace *T* with the representative of the data type name, `amulr()` or `aaddki()`, for example. Most of these are the first character of the data type, except for `complex`, whose representative character is `x`. The last procedure, `advz`*`T`*`()`, implements dividing vectors, but upon dividing by zero it calls `errfcn()`, supplied by the application as an external function.

Bitwise Boolean operators

These procedures perform boolean operations on integer arrays, returning the same type result. The resulting vector is the result of the boolean operation on each bit of each element of the arrays.

Procedure Call	***Purpose***
`anot`*`T`* `(a, b, npiix)`	NOT of a vector
`aand`*`T`* `(a, b, c, npix)`	AND of two vectors
`aandk`*`T`* `(a, b, c, npix)`	AND of a vector and a scalar
`abor`*`T`* `(a, b, c, npix)`	OR of two vectors
`abork`*`T`* `(a, b, c, npix)`	OR of a vector and a scalar
`axor`*`T`* `(a, b, c, npix)`	XOR (exclusive or) of two vectors
`axork`*`T`* `(a, b, c, npix)`	XOR of a vector and a scalar

Table 2.49: Bitwise Boolean Operators.

All of the above procedures are implemented *only* for the integer data types: `short`, `int`, and `long`.

Logical Comparison

These procedures return an `int` array containing the result of the logical comparison between elements of the input vectors. If the result of the comparison is `true`, the value in the vector is one, otherwise, it is zero.

Procedure Call	*Purpose*
`abeq`*T* `(a, b, c, npix)`	$a_i = b_i$?
`abeqk`*T* `(a, k, c, npix)`	$a_i = k$?
`abge`*T* `(a, b, c, npix)`	$a_i \geq b_i$?
`abgek`*T* `(a, k, c, npix)`	$a_i \geq k$?
`abgt`*T* `(a, b, c, npix)`	$a_i > b_i k$?
`abgtk`*T* `(a, k, c, npix)`	$a_i > k$?
`able`*T* `(a, b, c, npix)`	$a_i \leq b_i$?
`ablek`*T* `(a, k, c, npix)`	$a_i \leq k$?
`ablt`*T* `(a, b, c, npix)`	$a_i < b_i$?
`abltk`*T* `(a, k, c, npix)`	$a_i < k$?
`abne`*T* `(a, b, c, npix)`	$a_i \neq b_i$?
`abnek`*T* `(a, k, c, npix)`	$a_i \neq k$?

Table 2.50: Logical Comparison Functions.

All of the above are implemented for the range of SPP data types: `char`, `short`, `int`, `long`, `real`, `double`, and `complex`. Note, however, that the output vector, `c` is always an `int` array.

Fundamental Array Operators

These procedures implement various basic array operations. The `acht`*TT*`()` procedure is unique in that the input and output vectors are of different data types. It requires two data type specifiers (*t*) for the input and output vectors.

Function	*Data Type*	*Parameters*	*Purpose*
`amov`*T*	`csilrdx`	`(a, b, npix)`	Move (copy or shit) a vector
`amovk`*T*	`csilrdx`	`(k, b, npix)`	Move a scalar into a vector
`aclr`*T*	`bcsilrdx`	`(a, npix)`	Clear (zero) a vector
`acht`*TT*	`ubcsilrdx`	`(a, b, npix)`	Change datatype of a vector

Table 2.51: Fundamental Array Operators.

All of the above are implemented for the full range of SPP data types: `char`, `short`, `int`, `long`, `real`, `double`, and `complex`. In addition, `ach`*TT*`()` is implemented for unsigned byte `b` and unsigned short `u` types. These are used primarily in low-level image I/O (**imio**) code. `aclr`*T*`()` is also implemented for byte.

Algebraic Operators

These procedures implement various functions. The log and square root functions include an external function passed to the procedure that sets the returned value in the case of an invalid function result such as √-1.

Procedure Call	*Definition*
`aabs`*T* `(a, b, npix)`	Absolute value $b_i = \lvert a_i \rvert$
`amod`*T* `(a, b, c, npix)`	Modulus of two vectors
`amodk`*T* `(a, k, c, npix)`	Modulus of a vector and a scalar
`apow`*T* `(a, b, c, npix)`	Vector to an integer vector power $c_i = a_i^b i$
`apowk`*T* `(a, k, c, npix)`	Vector to an integer scalar power $c_i = a_i^k$
`aexp`*Tt* `(a, b, c, npix)`	Vector to a real vector exponent $c_i = a_i^b i$
`aexpk`*T* `(a, k, c, npix)`	Vector to a real scalar exponent $c_i = a_i^k$
`arcp`*T* `(a, k, c, npix)`	Reciprocal of a scalar and a vector $c_i = k/\mathrm{a}_i$
`arcz`*T* `(a, k, c, npix, errfcn)`	Reciprocal, detect divide by zero $c_i = k/\mathrm{a}_i$
`alln`*T* `(a, b, npix, errfcn)`	Natural logarithm $b_i = \ln' \mathrm{a}_i$
`alog`*T* `(a, b, npix, errfcn)`	Common logarithm $b_i = \log a_i$
`asqr`*T* `(a, b, npix, errfcn)`	Square root $b_i = \sqrt{a_i}$
`amag`*T* `(a, b, c, npix)`	Magnitude of vectors $c_i = (a_i^2 + b_i)^{1/2}$
`amgs`*T* `(a, b, c, npix)`	Magnitude squared of vectors $c_i = a_i^2 + b_i^2$

Table 2.52: Algebraic Operators.

All of these procedures are implemented for the data types: `short`, `int`, `long`, `real`, `double`, and `complex`, except the modulus functions `amod`*T*`()` and `amodk`*T*`()`, which are not implemented for `complex`.

Complex Operators

These procedures involve complex operators, but involve not only `complex` arguments.

Procedure	*Data Type*	*Arguments*	*Function*
`acjg`*T*	`x`	`(a, b, npix)`	Complex conjugate of a complex vector
`aimg`*T*	`silrd`	`(a, b, npix)`	Imaginary part of a complex vector
`aupx`*T*	`silrdx`	`(a, b, c, npix)`	Unpack the real and imaginary parts of a complex vector
`apkx`*T*	`silrds`	`(a, b, c, npix)`	Pack a complex vector given the real and imaginary parts

Table 2.53: Complex Operators.

`acjg`*T*`()` is implemented only for `complex` arrays. The first argument to `aimg`*T*`()` and `aupx`*T*`()` must be a `complex` array. The last argument to `aupx`*T*`()` must be a `complex` array.

Fourier Transforms

Procedure	*Arguments*	*Transform Type*
`afftrr`	`(sr, si, fr, fi, npix)`	Forward real Fourier transform, real arrays
`afftrx`	`(a, b, npix)`	Forward real Fourier transform, complex output
`afftxr`	`(sr, si, fr, fi, npix)`	Forward complex Fourier transform, real arrays
`afftxx`	`(a, b, npix)`	Forward complex Fourier transform, complex arrays
`aiftrr`	`(sr, si, fr, fi, npix)`	Inverse real Fourier transform, real arrays
`aiftrx`	`(a, b, npix)`	Inverse real Fourier transform, complex output
`aiftxr`	`(sr, si, fr, fi, npix)`	Inverse complex Fourier transform, real arrays
`aiftxx`	`(a, b, npix)`	Inverse complex Fourier transform, complex arrays

Table 2.54: Fourier Transforms.

The transform may be performed in place. The size of the arrays must be a power of two.

Transformations

Function	*Data Types*	*Parameters*	*Purpose*
`aglt`*T*	`csilrdx`	`(a, b, npix, low,high, kmul, kadd, nrange)`	General piecewise linear transformation
`altr`*T*	`silrdx`	`(a, b, npix, k1, k2, k3)`	Linear transformation of a vector $b_i = (a_i + k_i) \times k_2$
`alta`*T*	`silrdx`	`(a, b, npix, k1, k2)`	Linear map vector to vector $b_i = (a_i + k_1) \times k_2$
`altm`*T*	`silrdx`	`(a, b, npix, k1, k2)`	Linear map vector to vector $b_i = a_i k_1 + k_2$
`amap`*T*	`silrd`	`(, b, npix, a1, a2, b1, b2)`	Linear mapping of a vector with clipping
`alui`*T*	`silrd`	`(a, b, x, npix)`	Vector lookup and interpolate (linear)
`alut`*T*	`csil`	`(a, b, nchar, lut)`	Vector transform via lookup table

Table 2.55: Transformations.

Miscellaneous Procedures

Function	*Data Type*	*Parameters*	*Purpose*
`amin`*T*	`csilrdx`	`(a, b, c, npix)`	Vector minimum of two vectors
`amink`*T*	`csilrdx`	`(a, b, c, npix)`	Vector minimum of a vector and a scalar
`amax`*T*	`csilrdx`	`(a, b, c, npix)`	Vector maximum of two vectors
`amaxk`*T*	`csilrdx`	`(a, b, c, npix)`	Vector maximum of a vector and a scalar
`amed3`*T*	`csilrd`	`(a, b, c, med, npix)`	Vector median of three vectors
`amed4`*T*	`csilrd`	`(a, b, c, d, med, npix)`	Vector median of four vectors
`amed5`*T*	`csilrd`	`(a, b, c, d, e, med, npix)`	Vector median of five vectors
`arlt`*T*	`silrdx`	`(a, npix, floor, newval)`	Vector replace pixel if < scalar
`argt`*T*	`silrdx`	`(a, npix, ceil newval)`	Vector replace pixel if > scalar
`asel`*T*	`csilrdx`	`(a, b, c, sel, npix)`	Vector select from two vectors based on boolean flag vector
`asok`*T*	`csilrdx`	`(a, npix, ksel)`	Selection of the k^{th} smallest element of a vector
`acnv`*T*	`silrd`	`(a, b, npix, kernel, kpix)`	Convolve two vectors
`acnvr`*T*	`silrd`	`(a, b, npix, kernel, kpix)`	Convolve a vector with a real kernel
`asrt`*T*	`csilrdx`	`(a, b, npix)`	Sort a vector in increasing order
`abav`*T*	`silrdx`	`(a, b, nblocks, npix_block)`	Block average a vector
`absu`*T*	`silrd`	`(a, b, nblocks, npix_block)`	Block sum a vector
`awsu`*T*	`silrdx`	`(a, b, c, npix, k1, k2)`	Weighted sum of two vectors $c_i = k_1 a_i + k_2 b_i$
`ahgm`*T*	`csilrd`	`(a, npix, hgm, nbins, z1, z2)`	Accumulate the histogram of a series of vectors

Table 2.56: Miscellaneous Procedures.

Scalar Results

These procedures return a scalar value from computation upon a vector. In most cases, the data type of the function, vector or vectors, and the returned value must match. Exceptions are `aravt()` and `awvgt()`, in which the returned value is the number of points remaining in the sample after rejection.

Procedure Call	Data Types	Parameters	Purpose
`hival = ahiv`*T*	`csilrdx`	`(a, npix)`	Compute the high (max) value of a vector
`loval = alov`*T*	`csilrdx`	`(a, npix)`	Compute the low (min) value of a vector
`alim`*T*	`csilrdx`	`(a, npix, minval, maxval)`	Compute the limits (min and max) of a vector
`dot = adot`*T*	`silrdx`	`(a, b, npix)`	Dot product of two vectors $\sum a_i b_i$
`aavg`*T*	`silrdx`	`(a, npix, mean, sigma)`	Mean and standard deviation of a vector
`ngpix = arav`*T*	`silrdx`	`(a, npix, mean, sigma, ksig)`	Mean and standard deviation of a vector with pixel rejection (mean and sigma are floating point)
`ngpix = awvg`*T*	`silrdx`	`(a, npix, mean, sigma, lcut, hcut)`	Mean and standard deviation of a windowed vector (mean, sigma, lcut and hcut are floating point)
`med = amed`*T*	`csilrdx`	`(a, npix)`	Median value of a vector
`ssqrs = assq`*T*	`silrdx`	`(a, npix)`	Sum of squares of a vector $\sum a_i^2$ (returns floating point results)
`sum = asum`*T*	`silrdx`	`(a, npix)`	Sum of a vector $\sum a_i$ (returns floating point results)
`y = apol`*T*	`rd`	`(x, coeff, ncoeff)`	Polynomial evaluation $\sum a_i x^{i-1}$

Table 2.57: Scalar Results.

2.7 Vector Graphics — gio

The **gio** package allows an IRAF application written in SPP to draw graphics without regard to the ultimate plotting device. There is a complete description in the document *Graphics I/O Design* [Tody84b] available using the `help` command in the cl: `help gio$doc/gio.hlp fi+`. Here we primarily list the procedures, their calling sequences and a brief description of their function. The **gio** library allows a task to draw graphics with relatively little regard for specific graphics hardware. Nevertheless, some features are rather dependent on particular device characteristics.

High-Level Plotting Procedures

There are two procedures that allow an application to simply draw a graph using a set of data.

Procedure Call	***Purpose***
`gplotv (v, npts, x1, x2, title)`	Complete plot
`gploto (gp, v, npts, x1, x2, title)`	Plot a vector

Table 2.58: Graph Drawing Functions.

`gplotv` is completely self-contained. The application simply passes an array of real values in the argument `v` and the number of elements in the array in `npts`. The arguments `x1` and `x2` may be used to specify the X-axis values to assign to the first and last elements of the data vector `v`. Finally, the argument `title` is a character string plotted at the top of the graph. This may be specified as EOS, a null string, in which case no title is plotted. Note that `gplotv()` does not require the graphics descriptor argument (`gp` here). Opening and closing graphics are done entirely within the procedure. On the other hand, `gploto()` does require the descriptor. That is, the graphics must have been opened by `gopen()` (see below). All other **gio** procedures require the graphics descriptor argument. `gploto()` therefore permits more flexibility in resetting default plotting parameters.

Setup

These procedures enable graphics to be written to a particular device and control such operations as clearing the device (starting a new frame or page).

Procedure Call	***Purpose***
`gp = gopen (device, mode, fd)`	Open graphics
`gclose (gp)`	Close graphics
`gdeactivate (gp, flags)`	Deactivate graphics workstation
`greactivate (gp, flags)`	Activate graphics workstation
`gcancel (gp)`	Discard buffered graphics output
`gflush (gp)`	Flush buffered graphics output
`gclear (gp)`	Clear and reset the workstation
`gframe (gp)`	Advance the frame
`greset (gp, f)`	Reset graphics state
`gmftitle (gp, metafile_title)`	Comment metacode
`gpagefile (gp, fname, prompt)`	Page a file

Table 2.59: Graphics Device Setup Functions.

Note the distinction between the arguments to `gopen()`. The first is a string specifying the device on which to plot. This is most often coded using a string assigned from a cl parameter to be assigned by the user. The second argument is the **gio** I/O mode, analogous to the fio I/O modes. This is usually coded using a parameter constant. `NEW_FILE` will initialize graphics, erasing the screen or starting a new page while `APPEND` will not initialize graphics but will use the scaling and other parameters from the most recent graph (as long as the graphics buffer was not flushed). The final parameter is the *graphics stream* to use for the graphics metacode output. There are three streams specified using defined parameter constants: `STDGRAPH`, `STDPLOT`, and `STDIMAGE`. The streams behave identically but are resolved separately in disposing of the final plot. Example 2.27 briefly demonstrates the most common way of opening graphics:

```
char    device[SZ_LINE]    # Device name
bool    append             # Append?
int     mode               # Graphics I/O mode
pointer gp                 # Graphics I/O descriptor
pointer gopen()
bool    clgetb()
.
.
     call clgstr ("device", device, SZ_LINE)
     if (clgetb ("append"))
        mode = APPEND
     else
        mode = NEW_FILE
     gp = gopen (device, mode, STDGRAPH)
.
.
```

Example 2.27: Opening Graphics.

Graphics Parameters

There are a number of internal **gio** parameters that can be set in an SPP task. These control such aspects of the plot such as line width and text format. The system include file `<gset.h>` must be `include`d to allow reading or writing these parameters. It is also possible to query, but not set, certain attributes of the specified graphics device.

Procedure Call	*Purpose*
`gset`*T* `(gp, param, value)`	Set graphics parameter
`val = gsta`*T* `(gp, param)`	Query numeric graphics parameter
`nchar = gstats (gp, param, outstr, maxch)`	Query string graphics parameter
`val = gget`*T* `(gp, devcap)`	Query numeric device parameter
`nchar = ggets (gp, devcap, outstr, maxch)`	Query string device parameter

Table 2.60: Graphics Parameter Control Functions.

Use `gset`*T*`()` to set the value of a parameter and `gstat`*T*`()` to inquire its value. Note the distinction between these procedures and the `gget`*T*`()` procedures to query device characteristics from the *graphics capabilities* (`graphcap`) file. `gset`*T*`()` is implemented for `int`, `real`, and `string` data types, `gstat`*`T`*`()` is implemented for `int`, and `real` data types, and `gget`*`T`*`()` is implemented for `bool`, `int`, and `real` data types.

Scaling

These procedures deal with plot scaling. There are two fundamental coordinate systems used by *gio*: *normalized device coordinates* or *NDC*, whose range is always 0:1 in both directions regardless of the device, and the *world coordinate system* or *WCS*, defined by the application and corresponding to the user's data coordinates.

Procedure Call	***Purpose***
`gsview (gp, x1, x2, y1, y2)`	Set NDC viewport
`ggview (gp, x1, x2, y1, y2)`	Get NDC viewport
`gswind (gp, x1, x2, y1, y2)`	Set WCS window
`ggwind (gp, x1, s2, y1, y2)`	Get WCS window
`gascale (gp, v, npts, axis)`	Set absolute WCS scale
`grscale (gp, v, npts, axis)`	Set relative WCS scale
`ggscale (gp, x, y, dx, dy)`	Get WCS scale
`gctran (gp, x1, y1, x2, y2, wcs1, wcs2)`	Transform coordinates
`gcurpos (gp, x, y)`	Get current pen position

Table 2.61: Plot Scaling Functions.

NDC is associated with WCS using `gsview()` and `gswind()` to establish the plot scale. This may also be accomplished for a given set of data using `gascale()` or `grscale()`.

Drawing

The usual graphics primitives are available in **gio** such as basic pen moves and draws, line, marker, polyline, polymarker, and text drawing. The coordinates in every case are assumed to be in world coordinates (WC).

Procedure Call	*Purpose*
`gamove (gp, x, y)`	Pen up move in absolute WC
`grmove (gp, x, y)`	Pen up move in relative WC
`gadraw (gp, x, y)`	Pen down move in absolute WC
`grdraw (gp, x, y)`	Pen down draw in relative WC

Table 2.62: Pen Movement Primitives.

Move and draw may be absolute or relative to the last pen position.

Procedure Call	*Purpose*
`gline  (gp, x1, y1, x2, y2)`	Draw a line
`gpline (gp, x, y, npts)`	Draw a polyline
`gvline (gp, v, npts, x1, x2)`	Vector a polyline
`gtext  (gp, x, y, text, format)`	Draw text
`gfill  (gp, x, y, npts, style)`	Area fill
`glabax (gp, title, xlabel, ylabel)`	Draw labeled axes
`gmark  (gp, x, y, marktype, xsize, ysize)`	Draw a marker
`gpmark  (gp, x, y, npts, marktype, xsize, ysize)`	Draw a polymarker
`gvmark  (gp, v, npts, x1, x2, marktype, xsize, ysize)`	Vector a polymarker
`gumark  (gp, x, y, npts, xcen, ycen, xsize, ysize, fill)`	User defined marker

Table 2.63: Drawing Primitives.

`gpline()` and `gpmark()` take two vectors, with the X and Y coordinates of each point, while `gvline()` and `gvmark()` take a single vector of Y coordinates, and the X coordinates are evenly distributed along the X-axis, ranging from `x1` at `v[1]` to `x2` at `v[npts]` in WCS coordinates.

Procedure Call	*Purpose*
`gpcell (gp, m, nx, ny, x1, y1, x2, y2)`	Draw a cell array
`ggcell (gp, m, nx, ny, x1, y1, x2, y2)`	Read a cell array

Table 2.64: Cell Array Primitives.

A *cell array* is a gray-scale image. It is up to the graphics kernels (device drivers) to support capabilities such as drawing cell arrays or filled polygons. Most of the kernels do not support these.

Cursor Interaction

IRAF supports cursor read back through the cl so that a task may query the cursor. See “Interactive Graphics Cursor” on page 51 for a slightly more complete description of cursor interaction.

Procedure Call	*Purpose*
`gscur  (gp, x, y)`	Move device cursor
`stat = ggcur (gp, x, y, key)`	Get cursor position
`clgcur (param, wx, wy, wcs, key, strval, maxch`	Graphics cursor

Table 2.65: Cursor Interaction Functions.

Note that `clgcur()` is a **clio** procedure, not a **gio** procedure. Therefore, it does not require the graphics descriptor argument. Not all devices support moving the cursor from host software so `gscur()` may not have any effect.

2.8 Terminal I/O — tty

The **tty** interface is a table driven, device independent interface for controlling terminal and printer devices. Devices are described either by environment definitions, or by an entry in the **tty** database file. The **tty** database file is the standard Berkeley Unix *termcap* terminal capability database file (a text file), to which have been added entries for local printer devices. Accessing the Unix `termcap` file directly without modification is sometimes awkward, but the benefits of accessing a widely used, standard database more than compensate for any clumsiness.

When the cl starts up, the following environment variables are defined to describe the default terminal and printer devices. The user may subsequently change the values of these variables with the `set` statement or with the `stty` program.

Variable	***Contents***
`printer`	Default printer (e.g., `versatec`)
`terminal`	Default terminal (e.g., `vt100`, `tek4012`)
`termcap`	Terminal or printer database filename
`ttybaud`	Baud rate, default 9600
`ttyncols`	Number of characters per line
`ttynlines`	Number of lines per screen

Table 2.66: TTY Environment Variables.

The variables defining the names of the default terminal and printer devices will normally correspond to the names of device entries in the termcap file. The name of a file containing a single `termcap` entry for the device may optionally be given; the filename must contain a virtual filename (VFN) or operating system filename (OSFN) directory prefix to be recognized as a filename. The default termcap file is `dev$termcap`. Terminal initialization files (used to set tab stops) are files of the form `dev$tty.tbi`, where tty is the last field of the Unix pathname in the if termcap entry. If the first character of the if filename string is not a /, an IRAF VFN should be given.

The value strings for the environment variables `ttyncols` and `ttynlines`, defining the screen dimensions, are extracted from the `termcap` file by the `stty` program during start-up. The screen dimensions are defined in the environment for two reasons: efficiency, and if a window is used, the logical screen dimensions may be less than the physical screen dimensions. Most applications programs should therefore use `envgeti()` rather than `ttygeti()` to get the screen dimensions.

`ttygeti()` returns the physical screen dimensions as given in the `termcap` file.

Open and Close

Before any tty control sequences can be output, the tty device descriptor must be read from the termcap file into a buffer for efficient access. `ttyodes()` is used to open the **tty** descriptor; `ttycdes()` should be called when done to close the descriptor, returning all buffer space used. If `ttyname` is `terminal` or `printer`, the descriptor for the default terminal or printer is accessed.

Procedure Call	*Purpose*
`tty = ttyodes (ttyname)`	Open tty descriptor
`ttycdes (tty)`	Close tty

Table 2.67: TTY Open and Close Functions.

Low Level Database Access, TTY Control

The `ttyget()` procedures are used to get capabilities from the database entry. If the named capability is not found, `ttygeti()` returns zero, `ttygetb()` returns false, and `ttygets()` returns the null string. `ttysubi()` performs argument substitution on a control sequence containing at most two integer arguments (such as a cursor motion control sequence), generating an output sequence suitable for input to `ttyputs()`. `ttyputs()` puts the control sequence to the output file, padding as required given the number of affected lines. The baud rate and pad character, used to generate padding, are evaluated at `ttyodes()` time and are conveyed to `ttyputs()` in the tty descriptor.

Procedure Call	*Purpose*
`value = ttygett (tty, cap)`	Get a numeric parameter
`nchars = ttygets (tty, cap, outstr, maxch)`	Get a string parameter
`ttyputs (fd, tty, ctrlstr, afflncnt)`	Put a string parameter
`ttysub (ctrlstr, outstr, maxch, arg1, arg2)`	

Table 2.68: Low-Level TTY Database Functions.

`ttygett()` is implemented for `int`, `real`, and `bool` data types.

High-Level Control

Procedure Call	*Purpose*
`stat = ttyctrl (fd, tty, cap, afflncnt)`	Output a control sequence
`ttyso (fd, tty, YES|NO)`	Turn standout mode on or off
`ttygoto (fd, tty, col, line)`	Move cursor absolute
`ttyinit (fd, tty)`	Send `:is` and `:if`, if defined
`ttyclear (fd, tty)`	Clear screen
`ttyclearln (fd, tty)`	Clear the current line
`ttyputline (fd, tty, textline, map_cc)`	Put a text line

Table 2.69: High-Level TTY Functions.

`ttyctrl()` calls `ttygets()` and `ttyputs()` to process and output a control sequence (slightly less efficiently than if the control string is buffered by the user code). `ttygoto()` moves the cursor to the desired column and line. `ttyputline()` is like the fio `putline()`, except that it processes any form feeds, standout mode directives, and other control characters (including tabs) embedded in the text. Lines longer than `ttyncols` are broken into several output lines. `ttyputline()` is used by the `help`, `page`, `type`, and `lprint` utilities to map tabs and standout mode directives for a particular output device. Standout mode is mapped as reverse video on most VDTs, and as underscore on most printers and on overstrike terminals such as the Tektronix 4012.

2.9 Bit & Byte Operations — osb

Byte and Character Conversions

Procedure Call	***Purpose***
`strpak (str, os_sttr, maxch)`	Pack OS string
`strupk (os_str, str, maxch)`	Unpack OS string
`chrpak (a, a_off, b, b_off, nchars)`	Pack `char`
`chrupk (a, a_off, b, b_off, nchars)`	Unpack `char`
`bitpak (ival, wordp, offset, nbits)`	Pack an integer into a bitfield
`bitupk (wordp, offset, nbits)`	Unpack an unsigned integer bit field
`bitmov (a, a_off, b, b_off, nbits)`	Move a sequence of bits
`bytmov (a, a_off, b, b_off, nbytes)`	Move bytes
`bswaps (a, b, nshorts)`	Byte swap `short`
`bswap1 (a, b, nlongs)`	Byte swap `long`

Table 2.70: Byte and Character Conversions.

`chars` are signed integers, whereas bytes as unsigned integers. The `bswap`*`T`*`()` routines are used to swap bytes in short and long integer arrays, as is sometimes required when transporting data between machines. The **mii** package is available for conversions between a machine independent integer format and the SPP data types (documented elsewhere). See “Binary File I/O” on page 98 for an example of extracting individual bytes from a word.

Character Comparisons

The following are macro functions defined in the system include file `ctype.h`. The statement

```
include <ctype.h>
```

must be in the code in order to use them.

Procedure Call	*Purpose*
`bool = IS_UPPER (char)`	Upper case?
`bool = IS_LOWER (char)`	Lower case?
`bool = IS_DIGIT (char)`	Numeral?
`bool = IS_PRINT (char)`	Printable character?
`bool = IS_CNTRL (char)`	Control Character?
`bool = IS_ASCII (char)`	7-bit ASCII character?
`bool = IS_ALPHA (char)`	Letter (either case)?
`bool = IS_ALNUM (char)`	Alphanumeric character?
`bool = IS_WHITE (char)`	White space character?
`char = TO_DIGIT (char)`	Convert integer to char
`int = TO_INTEG (char)`	Convert digit to integer
`char = TO_UPPER (char)`	Convert to upper case
`char = TO_LOWER (char)`	Convert to lower case

Table 2.71: Character Comparison Functions.

These are macro definitions, not procedures (they produce in-line code and need not be declared). `TO_UPPER()` and `TO_LOWER()` must only be applied to letters of the proper case (use the procedures `chrupr()`, `chrlwr()` otherwise).

Pack and Unpack Characters

These procedures convert between SPP character strings (`short int` arrays) and packed byte blocks, i.e., a sequence of characters stored one per byte, delimited by `EOS` (ASCII NUL). The conversion may be performed in-place. That is, the input and output arrays may be the same.

Procedure Call	***Purpose***
`strpak (instr, outstr, maxch)`	Pack an SPP string into bytes
`strupk (instr, outstr, maxch)`	Unpack an SPP string from bytes
`chrpak (a, aoff, b, boff, nchars)`	Pack chars into bytes
`chrupk (a, aoff, b, boff, nchars)`	Unpack chars from bytes

Table 2.72: Pack and Unpack Functions.

Fortran Strings

There are two procedures that convert between SPP and Fortran character strings: `f77pak()` converts an SPP string to a Fortran string and `f77upk()` converts a Fortran string to an SPP string. An example is shown in Example 2.28.

Procedure Call	***Purpose***
`f77pak (spp, f77, maxch)`	Convert SPP string to Fortran string
`f77upk (F77, spp, maxch)`	Convert Fortran string to SPP string

Table 2.73: SPP/Fortran String Conversion.

```
# Declare the Fortran string
%      character*8    fstr
# Declare the SPP string
char        sstr[8]
.
.
     # Convert the SPP string to a Fortran string
     call f77pak (sstr, fstr, 8)
     # Call the fortran subroutine
     call forsub (fstr, ...)
.
.
```

Example 2.28: Converting Fortran/SPP Strings.

Note the *escaped* Fortran statement, preceded by `%`. See also “Fortran statements” on page 7.

Machine Independent I/O — mii

The mii integer format provides for three machine independent integer data types and two IEEE floating point formats.

Data Type	*Type of Number*
`MII_BYTE`	8-bit unsigned byte
`MII_SHORT`	16-bit twos-complement signed integer
`MII_LONG`	32-bit twos-complement signed integer
`MII_REAL`	32-bit IEEE floating point
`MII_DOUBLE`	64-bit IEEE floating point

Table 2.74: Machine-Independent Integer Data Types.

These types are defined in the system include file `mii.h` which must be included if using **mii**. The **mii** data types are the same as are used in the FITS transportable image format. In the case of the short and long integers, the most significant bytes of an integer are given first.

The routines in this package are provided for converting to and from the **mii** format and the SPP format. The latter format, of course, is potentially quite machine dependent. The implementation given here assumes that the SPP data types include 16-bit and 32-bit twos-complement integers; the ordering of the bytes within these integer formats is described by the machine constants `BYTE_SWAP2` and `BYTE_SWAP4`. Byte swapping for the IEEE floating formats is defined by the machine constants `IEEE_SWAP4` and `IEEE_SWAP8`.

Procedure Call	*Purpose*
`miipak (spp, mii, nelems, spptype, miitype)`	Pack an SPP array into an mii array
`miiupk (mii, spp, nelems, miitype, spptype)`	Unpack an mii array into an SPP array
`nchars = miipksize (nelems, miitype)`	Size (`chars`) of the SPP array required to store mii
`nelem = miinelem (nchars, miitype)`	Number of mii elements in an SPP array

Table 2.75: Machine-Independent/SPP Conversion Functions.

Note the distinction in the above table between the size of an mii array, specified as the number of array elements and the size of the SPP buffer, specified as the number of SPP chars. The following example illustrates reading an **mii** binary file consisting of byte (eight bit unsigned) values:

```
include <mii.h>
int     rf                # Rasterfile file descriptor
int     nelem             # Number of mii elements
pointer rpm, rps          # Rasterfile buffer descriptor
int     nchar             # SPP size of mii array
int     read(), miipksize()
begin
     nchar = miipksize (nelem, MII_BYTE)
     # Allocate buffer for reading mii data
     call malloc (rps, nchar, TY_SHORT)
     # Allocate SPP data array
     call malloc (rpm, nelem, TY_CHAR)
     # Read the file
     if (read (rf, Memc[rpm], nchar) != nchar)
        call error (0, "Could not read input file")
     # Unpack the data
     call miiupk (Memc[rpm], Mems[rps], nelem, MII_BYTE, TY_SHORT)
     .
     .
     call mfree (rpm, TY_CHAR)
     call mfree (rps, TY_SHORT)
end
```

Example 2.29: Reading an mii Binary File.

2.10 Pixel Lists — plio

The pixel list package is a general package for flagging individual pixels or regions of an image, to mark some subset of the pixels in an image. This may be done to flag bad pixels, or to identify those regions of an image to be processed by some applications program. When the pixel list package is used to flag the bad pixels in an image we call this a *bad pixel mask*, or BPM. When used to identify the regions of an image to be processed (or ignored), the list is called a *region mask*. The document *Pixel List Package Design* [Tody88] fully describes the details of the pixel list package. Here we only summarize and present a brief example. Example 2.30 opens a data image and the associated mask image, and sums the pixels within the area indicated by the mask.

```
include <pmset.h>

task    sum = t_sum
# SUM -- Sum the image pixels lying within the given mask

procedure t_sum()

char    image[SZ_FNAME]              # Input data image
char    mask[SZ_FNAME]               # image mask
int     npix, mval, totpix, mflags
long    v[PM_MAXDIM]
pointer im, mp, pp
real    sum
bool    clgetb()
real    asumr()
int     mio_glsegr()
pointer immap(), mio_open()

begin
     call clgstr ("image", image, SZ_FNAME)
     call clgstr ("mask", mask, SZ_FNAME)
     m_flags = 0
     if (clgetb ("invert"))
        m_flags = INVERT_MASK
     im = immap (image, READ_ONLY, 0)
     mp = mio_open (mask, m_flags, im)
     sum = 0; totpix = 0
     while (mio_glsegr (mp, pp, mval, v, npix) != EOF) {
        sum = sum + asumr (Memr[pp], npix)
        totpix = totpix + npix
     }
     call mio_close (mp)
     call imunmap (im)
     call printf ("%d pixels, sum=%g, mean=%g\n")
        call pargi (totpix)
        call pargr (sum)
     if (totpix > 0)
        call pargr (sum / totpix)
     else
        call pargr (INDEF)
end
```

Example 2.30: Opening Data Image and Associated Mask.

A more complex application might use the spatial information provided by `v` and `npix`, or the flag values provided by `mval` (for an integer mask). For example, a surface fitting routine would accumulate each line segment into a least squares matrix, using the coordinate information provided as well as the pixel values.

2.11 World Coordinates — mwcs

The mini-World Coordinate System (**mwcs**) interface is a package of procedures to handle the general problem of representing a linear or nonlinear world coordinate system (WCS). It may be used for determining the coordinates of pixels in an image, for example. Of course, enough information must be available to perform the appropriate coordinate transformations. While the interface is designed with the typical application to image data in mind, **mwcs** is intended as a general coordinate transformation facility for use with any type of data, as an embedded interface in other software, including system interfaces such as **imio** and **gio** as well as user applications. The **mwcs** package is referred to as a prototype since some functionality is missing.

- All WCS functions are built in (hard coded), hence the interface is not extensible at runtime and the only way to support new applications is through modification of the interface (by adding new function drivers).
- There is no support for modeling geometric distortions, except possibly in one dimension.
- There is no provision for storing more than one world coordinate system in FITS oriented image headers, although multiple WCS are supported internally by the interface, and are preserved and restored across `mw_save()` and `mw_load()` operations.
- Coordinate transforms involving dependent axes must include all such axes explicitly in the transform. Dependent axes are axes which are related, either by a rotation, or by a WCS function. Operations which could subset dependent axis groups, and which are therefore disallowed, include setting up a transform with an axes bitmap which excludes dependent axes, or more importantly, an image section involving dimensional reduction, where the axis to be removed is not independent. This could happen, for example, if a two-dimensional image were rotated and one tried to open a one-dimensional section of the rotated image.

For a more detailed discussion of the mwcs implementation and coordinate transformations in general, refer to the document *Mini-WCS Interface* [Tody89], also available on-line in `sys$mwcs/MWCS.hlp`. Use the help facility in the IRAF cl to read or print it.

Coordinate Systems

The **mwcs** package defines three coordinate systems between which two transformations are performed. The three coordinate systems are defined as follows:

- ***Physical*** - The physical coordinate system is the raw coordinate system of the data. In the case of an image, the physical coordinate system refers to the pixel coordinates of the original data frame. All other coordinates systems are defined in terms of the physical system (reference frame).
- ***Logical*** - The logical coordinate system is defined by the *Lterm* (see below) in terms of the physical coordinate system. In the case of an image, the logical coordinate system specifies raw pixel coordinates relative to some image section or derived image, i.e., the coordinates used for image I/O. In the **mwcs** the Lterm specifies a simple linear transformation, in pixel units, between the original physical image matrix and the current image section.
- ***World*** - The world coordinate system is defined by the *Wterm* (see below) in terms of the physical coordinate system. Any number of different kinds of world coordinate systems are conceivable. Examples are the tangent (gnomonic) projection, specifying right ascension and declination relative to the original data image, or any linear WCS, e.g., a linear dispersion relation for spectral data. Multiple world coordinate systems may be simultaneously defined in terms of the same physical system.

The coordinate systems are referred to by the strings `physical`, `logical`, and `world`. Note that there may be many Wterms specified for any one WCS. The *world* system refers to the current Wterm defined. Other Wterms are referred to by user-supplied names (see `mw_newsystem()`) and can be made the current system by `mw_ssystem()`. The two transformations are specified by the Lterm and the Wterm. The Lterm specifies a linear transformation between the physical and logical coordinate systems. The Wterm specifies the transformation between the physical and world coordinate systems The general flow of transforming coordinates is:

1. ***Retrieve*** or ***Create*** the Lterm and/or Wterm using `mw_open()`, `mw_openim()`, etc.
2. ***Modify*** the Lterm and/or Wterm (if necessary) using `mw_slterm()`, `mw_swterm()`, etc.

3. ***Precompute*** the transformations between the coordinate systems using the procedure `mw_sctran()`
4. ***Perform*** the transformations for specific coordinates using `mw_ctran()`, etc.

A WCS always has a number of predefined attributes, and may also have any number of user defined, or WCS specific, attributes. These are defined when the WCS is created, in the `wattr` argument input to `mw_swtype()`, or in a subsequent call to `mw_swattrs()`. The WCS attributes for a specific axis may be queried with the function `mw_gwattrs()`. Attribute values may be modified, or new attributes defined, with `mw_swattrs()`. The issue of WCS attributes is discussed further in the next section. The WCS attributes which can be set by the `wattr` term consist of a number of standard attributes, plus an arbitrary number of additional WCS specific (application defined) attributes. The following standard attributes are reserved (but not necessarily defined) for each WCS:

Attribute	***Definition***
`units`	Axis units (`pixels`, etc.)
`label`	Axis label, for plots
`format`	Axis numeric format, for tick labels
`wtype`	WCS type, e.g., `linear`

Table 2.76: WCS Standard Attributes.

In addition, the following are defined for the entire WCS, regardless of the axis:

Attribute	***Definition***
`system`	System name (logical, physical, etc.)
`object`	External object with which WCS is associated

Table 2.77: WCS Attributes.

For example, to determine the WCS type for axis 1:

```
call mw_gwattrs (mw, 1, "wtype", wtype, SZ_WTYPE)
```

Axis Mapping

The coordinate transformation procedures include support for a feature called axis mapping, used to implement dimensional reduction. A example of dimensional reduction occurs in **imio**, when an image section is used to specify a subraster of an image of dimension less than the full physical image. For example, the section might specify a one dimensional line or column of a two or higher dimensional image, or a two dimensional section of a three dimensional image. When this occurs the application sees a logical image of dimension equal to that of the image section, since logically an image section is an image. Dimensional reduction is implemented in **mwcs** by a transformation on the input and output coordinate vectors. The internal **mwcs** coordinate system is unaffected by either dimensional reduction or axis mapping; axis mapping affects only the view of the WCS as seen by the application using the coordinate transformation procedures. For example, if the physical image is an image cube and we access the logical image section `[*,5,*]`, an axis mapping may be set up which maps physical axis one to logical axis one, physical axis two to the constant 5, and physical axis three to logical axis two. The internal system remains three dimensional, but the application sees a two dimensional system. Upon input, the missing axis y=5 is added to the two dimensional input coordinate vectors, producing a three dimensional coordinate vector for internal use. During output, axis two is dropped and replaced by axis three. The axis map is entered with `mw_saxmap()` and queried with `mw_gaxmap()`. Here, `axno` is a vector, with `axno[i]` specifying the logical axis to be mapped onto physical axis *i*. If zero is specified, the constant `axval[i]` is used instead. Axis mapping may be enabled or disabled with a call to `mw_seti()`. Axis mapping affects all of the coordinate transformation procedures and all of the coordinate system specification procedures. Axis mapping is not used with those procedures which directly access or modify the physical or world systems (e.g., `mw_slterm()` or `mw_swterm()`) since full knowledge of the physical system is necessary for such operations.

Object Creation and Storage

The **mwcs** interface routines used to create or access **mwcs** objects, or save and restore **mwcs** objects in external storage, are summarized below.

Procedure Call	***Purpose***
`mw = mw_open (bufptr, ndim)`	Create an mwcs object
`mw = mw_openim (im)`	Create an mwcs object based on information from an image
`ms = mw_newcopy (mw)`	Create new copy of an mwcs object
`mw_close (mw)`	Remove an mwcs object
`mw_load (mw, bufptr)`	Reload an mwcs object
`mw_save (mw, bufptr, buflen)`	Save mwcs information in a buffer
`mw_laodim (mw, im)`	Reload a mwcs object from image header information
`mw_saveim (mw, im)`	Save an mwcs object into an image header

Table 2.78: MWCS Object Functions.

`mw_open()` creates a new **mwcs** object and a pointer to it is returned. If `bufptr` is `NULL`, then an identity transformation is created with the dimension specified by `ndim`. If `bufptr` is pointing to an encoded **mwcs** buffer, the **mwcs** object is loaded with that information `mw_openim()` initializes an **mwcs** object with data from the image pointed to by the image descriptor `im`. If the image contains no mwcs information, an identity transformation is loaded instead. `mw_newcopy()` creates a new mwcs object that is a copy of the **mwcs** object specified by `mw`. `mw_close()` deallocates the memory structures associated with the **mwcs** object `mw`. **mwcs** objects can be saved in an encoded, machine-independent format in a memory array. This array can then be saved into a file, sent over the network, etc. `mw_save()` will save the contents of the **mwcs** object `mw` into the memory pointed to by the char pointer `bufptr`. If `bufptr` is `NULL`, a memory buffer is allocated whose pointer is returned in `bufptr`. If `bufptr` is not `NULL`, the buffer, of length `buflen`, is used (and resized if necessary). The length of the buffer is returned. The buffer `bufptr` can be used in the calls `mw_open()` and `mw_load()`. `mw_load()` reloads the **mwcs** object `mw` with information contained in the buffer `bufptr` saved by `mw_save()`. `mw_loadim()` reloads an existing mwcs object `mw` with information from the image pointed to by the image descriptor `im`. `mw_saveim()` saves the contents of the **mwcs** object `mw` into the image pointed to by the image descriptor `im`.

Coordinate Transformation Procedures

The **mwcs** procedures used to perform coordinate transformations are summarized below.

Procedure Call	***Purpose***
`ct   = mw_sctran (mw, system1, system2, axes)`	Compile a coordinate transformation between systems
`ival = mw_gctrans`*`T`*` (ct, ltm, ltv, axtype1, axtype2, maxdim)`	Return the compiled transformation
`mx_ctfree (ct)`	Deallocate the coordinate transformation structure
`x2   = mw_c1tran`*`T`*` (ct, x1)`	Return the transformation of a single point
`mw_v1tran`*`T`*` (ct, x1, x2, npts)`	Return the transformation of an array of points
`mw_c2tran`*`T`*` (ct, x1, y1, x2, y2)`	Return the two-dimensional transformation of a point
`mw_v2tran`*`T`*` (ct, x1, y1, x2, y2, npts)`	Transform an array of two dimensional points
`mw_ctran`*`T`*` (ct, p1, p2, ndim)`	Transform an arbitrarily dimensioned point
`mw_vtran`*`T`*` (ct, v1, v2, ndim, npts)`	Transform an array of arbitrarily dimensioned points

Table 2.79: MWCS Coordinate Transformation Procedures.

The `mw_sctran()` procedure precomputes the transformation from one coordinate system, `system1`, to another, `system2`, for the specified axes in the **mwcs** object `mw` returning a pointer to the optimized coordinate transformation. This pointer, `ct` is used in the subsequent coordinate transformation calls, `mw_c2tran()`, etc. The `axes` argument is a bitfield that represents which axes the transformation should apply to. That is, if you wish to transform the first two axes (x and y), set `axis = 3`. The `mw_gctrant()` procedure retrieves a compiled linear transformation and returns the dimensionality of the transformation. The argument `ltm` contains the coefficient determination matrix, `ltv` contains the translation vector, `axtype1` contains the axis types for each of the axes in the source coordinate system, `axtype2` contains the axis types in the destination coordinate system, and `maxdim` specifies the maximum dimensionality that the arrays can handle.

Coordinate System Specification

The procedures used to enter, modify, or inspect the **mwcs** logical and world coordinate transformations are summarized below.

Procedure Call	***Purpose***
`mw_sltermT (mw, ltm, ltv, ndim)`	Set the Lterm for the specified object
`mw_gltermT (mw, ltm, ltv, ndim)`	Get the Lterm for the specified object
`mw_ssystem (mw, system)`	Set the default world system
`mw_newsystem (mw, system, ndim)`	Create a new world coordinate system
`mw_swtermT (mw, r, w, cd, ndim)`	Set the Wterm for the current system
`mw_gwtermT (mw, r, w, cd, ndim)`	Get the Wterm for the current system

Table 2.80: MWCS System Specification Functions.

The procedures `mw_sltermT()` and `mw_gltermT()` are used to directly enter or inspect the Lterm of the **mwcs** object `mw`, which consists of the linear transformation matrix `ltm` and the translation vector `ltv`, both of dimension `ndim`, defining the transformation from the physical system to the logical system.

Procedure Call	***Purpose***
`mw_saxmap (mw, axno, axval, ndim)`	Set the axis mapping
`mw_gaxmap (mw, axno, axval, ndim)`	Get the axis mapping
`mw_swtype (mw, axis, naxes, wtype, wattr)`	Set the axis type and attribute
`mw_swattrs (mw, axis, attribute, valstr)`	Set the axis attribute
`mw_gwattrs (mw, axis, attribute, valstr)`	Get the axis attributes
`mw_swsamp`*`T`* `(mw, axis, pv, wv, npts)`	Set a world system using sampled data
`mw_gwsamp`*`T`* `(mw, axis, pv, wv, npts)`	Get a world system using sampled data

Table 2.81: Axis Specification Functions.

Procedure Call	***Purpose***
`mw_translator (mw, ltv_1, ltm, ltv_2, ndim)`	Apply a general transformation to the Lterm, single precision
`mw_translated (mw, ltv_1, ltm, ltv_2, ndim)`	Apply a general transformation to the Lterm, double precision
`mw_rotate (mw, theta, center, axes)`	Apply a rotation transformation to the Lterm
`mw_scale (mw, scale, axes)`	Apply a scale transformation to the Lterm
`mw_shift (mw, shift, axes)`	Apply a translation (shift) transformation to the Lterm

Table 2.82: Applying Transformations to Lterm.

If the logical system undergoes successive linear transformations, `mw_translate()` may be used to translate, rather than replace, the Lterm of the **mwcs** object mw, where the given transformation matrix and translation vector refer to the relative transformation undergone by the logical system. This will always work since the Lterm is initialized to the identity matrix when a new **mwcs** object is created. See also `mw_rotate()`, `mw_scale()`, and `mw_shift()`.

Generic coordinate transformations are available using the procedures `mw_translate()`, `mw_rotate()`, `mw_scale`, and `mw_shift`. The `mw_translate()` procedure is the most general, with the others provided as convenient front-ends. Note that `mw_rotate()` rotates the Lterm of the **mwcs** object `mw` through the angle theta, specified in radians, about an arbitrary point center for the specified axes. The axes argument is a *bitfield* representing which axes to which the transformation applies. That is, each bit represents an axis to transform.

mwcs **Parameters**

The **mwcs** status procedures, used to query or set the **mwcs** parameters, are as follows.

Procedure Call	***Purpose***
`mw_seti (mw, what, ival)`	Set a parameter
`ival = mw_stati (mw, what)`	Retrieve a parameter

Table 2.83: MWCS Status Procedures.

Name	***Type***	***Description***
`MW_NDIM`	`int`	Dimensionality of logical system
`MW_NWCS`	`int`	Number of WCS defined
`MW_REFIM`	`int`	Reference image, if any
`MW_USEAXMAP`	`bool`	`true` if axis mapping is enabled
`MW_NPHYSDIM`	`int`	Dimensionality of physical system
`MW_SAVELEN`	`int`	Character required for `mw_save()` buffer

Table 2.84: MWCS Interface Parameters.

`MW_NDIM` may differ from `MW_NPHYSDIM` if dimensional reduction has been specified and axis mapping is enabled. `MW_NWCS` returns the number of WCS currently defined; at least two WCS are always defined, i.e., the logical and physical systems (the world system will default to the physical system if not otherwise defined).

Matrix Routines

The following general purpose matrix manipulation routines are used internally within the interface to compile or evaluate transformations, and may be useful in applications code as well.

Procedure Call	*Purpose*
`mw_invertt (o_ltm, n_ltm, ndim)`	Invert a square matrix
`mw_mmult (ltm_1, ltm_2, ltm_out, ndim)`	Multiply two matrices
`mw_vmult (ltm, ltv_in, ltv_out, ndim)`	Multiply a matrix and a vector

Table 2.85: Matrix Routines.

Each is implemented for both `real` and `double` variables. They operate on square matrices whose dimensions are specified by `ndim`, i.e., `ltm[ndim,ndim]`.

Examples

This section presents of a few simple examples to demonstrate the basic workings of the **mwcs** interface. The examples will be code fragments showing the necessary declarations, etc., and are not intended to be complete programs.

Example 2.31 shows how to retrieve the **mwcs** information from an image. Example 2.32 will create a WCS such that the world system is centered on an image and the axis decrease value with increasing pixel value. Example 2.33 shows some examples of transforming coordinates with an already opened **mwcs** object. Assume that the **mwcs** object describes a transformations for a three dimensional image. The final example (Example 2.34) prints all the values for all the attributes of all the axis of an image's **mwcs**.

```
pointer mw, im
char imagename[SZ_FNAME]
      .
      .
      # Open the image and the mwcs of the image
      call clgstr ("image", imagename, SZ_FNAME)
      im = immap (imagename, READ_ONLY, O)
      mw = mw_openim (im)
      # Perform any mwcs manipulation
      # Close the image and the mwcs.
      call mw_close (mw)
      call imunap (im)
```

Example 2.31: Retrieving mwcs Information From an Image.

This next example creates a WCS such that the world system is centered on an image and the axis decreases with increasing pixel values.

```
include <imhdr.h>

pointer mw, im, mw_open(), immap()
real    cd[2,2], r[2], w[2]
.
.
begin
     # Create a new 2-dimensional mwcs object
     mw = mw_open (NULL, 2)
     # Open an image
     im = immap (imagename, READ_WRITE, 0)
     # Modify the Wterm as described above.
     cd[1,1] = -1.0
     cd[2,2] = -1.0
     cd[1,2] = 0.0
     cd[2,1] = 0.0
     r[1]    = IM_LEN(im, 1) / 2.
     r[2]    = IM_LEN(im, 2) / 2.
     w[1]    = 0.0
     w[2]    = 0.0
     call mw_swtermr (mw, r, w, cd, 2)
     # Place the new mwcs object into the image header.
     call mw_saveim (mw, im)
     .
     .
```

Example 2.32: Creating WCS Centered on Image.

The following examples transform coordinates with an already opened **mwcs** object. Assume that the object describes a transformation for a 3-dimensional image.

```
pointer im, mw, immap(), mw_open()
pointer lw1ct, wl1ct, lw2ct, wl2ct, lw3ct, mw_sctran()
real    mw_c1tranr()
real    logical_point, world_point
real    logical3_array[3,npts], world3_array[3,npts]
double  world2d_x, world2d_y, logical2d_x, logical2d_y
double  logical_point_array[npts], world_point_array[npts]
.
.
begin
     # Open image and its mwcs
     im = immap (imagename, READ_ONLY, 0)
     mw = mw_openim (im)
     # Compute the 1-dimensional transformation from the logical to
     # world and world to logical systems for the first axis.
     lw1ct = mw_sctran (mw, "logical", "world", 1b)
     wl1ct = mw_sctran (mw, "world", "logical", 1b)
     # Define the 2-dimensional transformation for the 2nd and 3rd axis
     lw2ct = mw_sctran (mw, "logical", "world", 6b)
     wl2ct = mw_sctran (mw, "world", "logical", 6b)
     # Define the full 3-dimensional transformation for all the axis
     lw3ct = mw_sctran (mw, "logical", "world", 7b)
     wl3ct = mw_sctran (mw, "world", "logical", 7b)
     # Transforms various points
     world_point = mw_c1tranr (lw1ct, logical_point)
     logical_point = mw_c1tranr (wl1ct, world_point)

     call mw_v1trand (lw1ct, logical_point_array, world_point_array, npts)
     call mw_c2trand (wl2ct, world2d_x, world2d_y, logical2d_x, logical2d_y)
     call mw_vtranr (lw3ct, logical3_array, world3_array, 3, npts)
     .
     .
```

Example 2.33: Transforming Coordinates in an Open mwcs.

The example below prints all values for all attributes of all axes of an image's **mwcs**.

```
include <mwset.h>

pointer im, mw, immap(), mw_openim()
int     axis, attr_index, mw_stati()
char    attr_index_string[SZ_LINE], value[SZ_LINE]
.
.
begin
     # Open the image and its mwcs
     im = immap (imagename, READ_ONLY, 0)
     mw = mw_openim (im)

     do axis = 1, mw_stati (mw, MW_NDIM) {
        call printf ("For axis %d:\n")
        call pargi (axis)
           attr_index = 1

           repeat {
              call sprintf (attr_index_string, SZ_LINE, "%d")
              call pargi (attr_index)

              ifnoerr (call mw_gwattrs (mw, axis, attr_index_string,
                            value, SZ_LINE)
                 call printf ("For attribute %d, %s, the value is %s.\n")
                 call pargi (attr_index)
                 call pargstr (attr_index_string)
                 call pargstr (value)
                 attr_index = attr_index + 1

             } else {
                 call printf ("No more attributes for axis %d.\n")
                 call pargi (axis)
                 break
             }
          }
       }
       .
       .
```

Example 2.34: Printing Axis Attribute Values for a mwcs.

2.12 **Miscellaneous** — etc

cl **Environment Variables**

These procedures return the value of a cl environment variable. There is a separate procedure for each of the data types `bool`, `int`, `real`, `double`, and character strings. There is no distinction made between the variously sized integer variables. If the variable is not found or cannot be converted to the appropriate data type, the procedures abort.

Procedure Call	***Purpose***
`bool   = envgetb (varname)`	Get a `boolean` environment variable
`int    = envgeti (varname)`	Get an `integer` environment variable
`real   = envgetr (varname)`	Get a `real` environment variable
`double = envgetd (varname)`	Get a `double` environment variable
`envgets (key, value, maxch)`	Get a `string` environment variable

Table 2.86: Reading Environment Variables.

Time and Timing

These procedures deal with absolute local time as well as relative CPU clock time.

Procedure Call	*Purpose*
`brktime (ltime, tm)`	Convert a long integer time into year, month, day, etc.
`long = clktime (old_time)`	Get the clock time
`cnvdate (ltime, outstr, maxch)`	Convert long integer time to date string (short format)
`cnvtime (ltime, outstr, maxch)`	Convert long integer time to time string (long format)
`long = cputime (old_cputime)`	Get the CPU time consumed by process
`sys_mtime()`	Mark the time (for timing programs)
`sys_ptime()`	Print the elapsed time since last mark

Table 2.87: Clock and Timing Procedures.

The `clktime()` procedure gets the current clock time (local standard time) in units of seconds since 00:00:00 1 January 1980. This can be broken down into days, hours, seconds, etc. with `brktime()`, or printed as a date and time string with `cnvtime()`. The `brktime()` breaks the long integer time returned by `clktime()` into the fields of the structure defined in `<time.h>`. The procedure is valid from 00:00:00 on 1 January 1980 to 23:23:59 28 on February 2100. `cnvdate()` converts a time in integer seconds since midnight on 1 January 1980 into a short string such as "May 15 18:24". `cnvtime()` converts a time in integer seconds since midnight on 1 January 1980 into a string, i.e., "Mon 16:30:05 17-Mar-82". The length of the output strings for the procedures is given by the parameter `SZ_DATE` in `<time.h>`.

Parameter	*Contents*
`SZ_TIME`	Size of `dow 00:00:00 dd-Mmm-yy`
`SZ_DATE`	Size of `mmm dd hh:mm`
`LEN_TMSTRUCT`	Length of time struct
`TM_SEC`	Seconds (0-59)
`TM_MIN`	Minutes (0-59)
`TM_HOUR`	Hour (0-23)
`TM_MDAY`	Day of month (1-31)
`TM_MONTH`	Month (1-12)
`TM_YEAR`	Year, e.g., 1982
`TM_WDAY`	Day of week (Sunday is 1)
`TM_YDAY`	Day of year (1-366)

Table 2.88: Time Parameters.

Process Information

These procedures return information about the current process.

Procedure Call	*Purpose*
`getuid  (outstr, maxch)`	Get the name of the runtime user of a program
`gethost (outstr, maxch)`	Get the network name of the host machine
`int   = getpid()`	Get the process id
`sysid (oustr, maxch)`	Return a line of text identifying the process

Table 2.89: Process Information Functions.

The `getpid()` procedure returns an integer process identifier, while the others return a string value. The `sysid()` procedure returns a line of text identifying the current user, machine, and version of IRAF, and containing the current date and time of the form:

```
NOAO/IRAF V1.3 username@lyra Tue 09:47:50 27-Aug-85
```

The string `NOAO/IRAF V1.3` is given by the value of the cl environment variable `version`. The string `username` is the value of the environment variable `userid`, defined by the user in the `login.cl` file. The output string is not terminated by a newline.

Convert Flags

These procedures convert between `bool` variables and `int` logical flags having the values `YES` or `NO`.

Procedure Call	*Purpose*
`int  = btoi (boolean_value)`	Convert boolean to integer flag
`bool = itob (int_value)`	Convert integer to boolean

Table 2.90: Flag Conversion Functions.

Miscellaneous Functions

Procedure Call	*Purpose*
`int = lpopen (device, mode, type)`	Open the line printer as a file
`int = oscmd (cmd, infile, outfile, errfile)`	Send a command to the host operating system
`pagefiles (files)`	Display a text file or files on the standard output
`qsort (x, nelem, compare)`	General quick sort for any data structure
`tsleep (seconds)`	Delay process execution

Table 2.91: Miscellaneous Functions.

The `oscmd()` procedure sends a machine dependent command to the host operating system. It tries to spool the standard output and error output in the named files if the names for the files are not null. The integer flag `OK` is returned if the command executes successfully. The `qsort()` procedure is a general quicksort for arbitrary objects. The argument `x` is an int array indexing the array to be sorted. The user supplied function `compare(x1,x2)` is used to compare objects indexed by x. The value returned by compare has the following significance for sorting in increasing order:

$$compare = \left\{ \begin{array}{ll} -1 & \text{if' obj } [x_1] < \text{obj } [x_2] \\ 0 & \text{if obj } [x_1] = \text{obj } [x_2] \\ 1 & \text{if obj } [x_1] > \text{obj } [x_2] \end{array} \right\}$$

CHAPTER 3:

Error Handling

The SPP language provides two facilities for error handling (see Table 3.1).

Procedure	*Error Handled*
`error (errno, errtext)`	Signal error condition (`errtext` cannot include `\n`)
`fatal (errno, errtext)`	Signal fatal error condition

Table 3.1: Error Handlers in SPP.

An error is signalled by calling the `error()` procedure. The `error()` procedure takes two arguments. The first argument is the error number. Application programs that call the error procedure should use an error number between 1 and 500. Numbers above 500 are used for system errors. The error number is used by any code which catches errors to distinguish between the different types of errors. If your application program does not catch errors, the error number is arbitrary. The second argument is the error message. This argument is a string printed on the standard error stream, which is usually connected to the user's terminal. Note that the error message should *not* contain any newline (`\n`) characters. The procedure in Example 3.1 demonstrates the use of the error procedure.

```
# GEOMEAN -- Calculate the geometric mean of a real array
real procedure geoman (x, n)
real    x      # i: Array of positive numbers
int     n      # i: Size of array
#--
int     i
int     sum
begin
     if (n <= 0)
         call error (1, “Can’t compute geometric mean: no values”)
     sum = 0.0
     do i = 1, n {
         if (x[i] <= 0.0)
             call error (1, “Can’t compute geometric mean: <0”)
         else
             sum = sum + log (x[i])
     }
     return exp (sum / real(n))
end
```

Example 3.1: Errors Flagged by `error()` Procedure.

There is another procedure with the same arguments as `error()` named `fatal()`. The difference between the two procedures is the severity of the error level. Errors which are posted by the `fatal()` procedure cannot be caught.

iferr

Errors are caught by enclosing the statements to be checked for errors in an `iferr` block or an `ifnoerr` block. An `iferr` block has one of two forms. The first form can only check a single statement and the statement must either be an assignment statement or a procedure call. The second form can check any number of statements of any type. The two forms of the `iferr` block have the following syntax:

Single Condition, Assignment/Procedure Call

```
iferr (statement) {
   statements
} else {
   statements
}
```

Multiple Conditions, Multiple Statements

```
iferr {
   statements
} then {
   statements
} else {
  statements
}
```

Figure 3.2: Syntax for `iferr`.

The `else` portion of the `iferr` block is optional. The meaning of an `iferr` block is that if an error occurs (i.e., if `error()` was called by one of the statements in the block) while executing the statements checked by the block, then execute the following code, but otherwise execute the code in the `else` block. The normal action of the error procedure, which is to print a message on the standard error stream, is suppressed. The syntax of an `ifnoerr` block is the same as that of the `iferr` block, except that the keyword `iferr` is replaced by `ifnoerr`. The meaning of an `ifnoerr` block is the opposite of that of the `iferr` block. If no error occurs during the execution of an `ifnoerr` block, then the following code is executed, otherwise the `else` block is executed. The following example shows the two forms of the `iferr` block.

```
iferr (result = geoman (x, n)) {
    result = 0.0
}
```

```
iferr {
    result = geoman (x, n)
} then {
    result = 0.0
}
```

Example 3.3: Two Ways to Use the `iferr` Block.

If there is more than one procedure call in a given block, then `errchk()` all of them except the last (see below).

errchk

In Example 3.3, the `iferr` block catches an error in a procedure that it calls directly, `geomean`. It is possible, however, for the error to occur in a subroutine that is called indirectly, that is, called by the called procedure. In order for the `iferr` block to check for these errors, an `errchk` statement must be added to each of the procedures between the procedure with the `iferr` block and the procedure which contains the `error()` call. The `errchk` statement is placed in the declarations section of the procedure and has the following syntax:

`errchk` *`list of procedure names`*

When an error occurs in a procedure whose name is listed in an `errchk` statement, program execution in the calling procedure jumps to the `return` statement. Thus the rest of the code in the calling procedure is skipped. By including `errchk` statements in all of the routines between the procedure with the `iferr` block and the procedure which contains the `error()` call, program execution will return to the `iferr` block without executing any intervening code if the `error()` procedure is called. Example 3.4 shows the use of the `errchk` statement. The lowest level procedure, `gtdist`, computes the distance between two points. If this distance is zero, it calls the `error()` procedure. The intermediate level procedure, `gtinv`, computes the inverse of the distance. To prevent the procedure from trying to compute the inverse of zero, the procedure contains an `errchk` statement for `gtdist`. This causes the execution of the program to skip this statement and return to the `iferr` block in `gtline`.

```
include <mach.h>        # Defines EPSILONR

# GTLINE -- Compute the line between two points (A*x + B*y + C = 0)

bool procedure gtline (p1, p2, a, b, c)

real   p1[2]           # i: First point
real   p2[2]           # i: Second point
real   a               # o: X coefficient
real   b               # o: Y coefficient
real   c               # o: Constant term
#--
real   inv

begin
     iferr (call gtinv (p1, p2, inv)) {
         # The two points coincide
         a = 0.0 ; b = 0.0 ; c = 0.0
         return false

     } else {
         a = (p1[2] - p2[2]) * inv
         b = (p2[1] - p2[1]) * inv
         c = (p1[1] * p2[2] - p1[2]) * 8nv
         return true
     }
end

# GTINV -- Calculate inverse of the distance between two points

procedure gtinv (p1, p2, inv)

real     p1[2]         # i: First point
real     p2[2]         # i: Second point
real     inv           # o: Inverse distance
#--
real     dist
errchk   gtdist

begin
     call gtdist (p1, p2, dist)
     inv = 1.0 / dist
end
```

(Continued...)

Example 3.4: Using the `errchk` Statement.

```
# GTDIST -- Calculate the distance between two points

procedure gtdist (p1, p2, dist)

real   p1[2]          # i: First point
real   p2[2]          # i: Second point
real   dist           # o: Distance
#--
real   dx, dy, distsq

begin
     dx = p2[1] - p1[1]
     dy = p2[2] - p1[2]
     distsq = dx * dx + dy * dy
     if (distsq < EPSILONR)
         call error (1, "The two points coincide")
     else
         dist = sqrt (distsq)
end
```

Example 3.4 (Continued): Using the `errchk` Statement.

Additional Error Handling Procedures

IRAF provides several procedures for handling errors in an `iferr` block. The `errcode` procedure returns the error code that was passed to the `error()` procedure. This allows the program to distinguish between different kinds of errors. The `errget` procedure also returns the error code and in addition returns the error message that was passed to the `error()` procedure. The `erract` procedure allows a program to repost the error that was caught by the `iferr` block. The `erract` procedure has one argument, the severity level of the error. There are three error levels and they are defined in the include file `error.h`. The two highest levels, `EA_FATAL` and `EA_ERROR`, correspond to the error levels produced by the procedures `fatal()` and `error()` respectively. Thus calling `erract` with the argument `EA_FATAL` is the same as calling `fatal()` with the same error that was previously posted by `error()`. Similarly, calling `erract` with the argument `EA_ERROR` is the same as calling `error()` again. The lowest error level is `EA_WARN`. If `erract` is called with an argument of `EA_WARN`, the error message is printed on the standard error stream and execution of the program proceeds as usual. The calling sequences for these three routines are the following.

Call	*Error Handled*
`code = errcode ()`	Return error code
`code = errget (oustr, maxch)`	Return error code and message
`erract (severity)`	Repost error
`xer_reset ()`	Reset error state

Table 3.2: Error Handling Procedures.

The following example (Example 3.5) illustrates the use of the `errcode` and `erract` procedures. It converts all errors with a code of one to warnings and reposts all other errors as errors.

```
include <error.h>

# JDATE -- Print the Julian data for each date in the file

procedure jdate (fname)

char   fname[ARB]       # i: File name
#--
char   line[SZ_LINE]
int    fd, year, month, day, date

int    open(), getline(), errcode()

begin
     fd = open (fname, READ_ONLY, TEXT_FILE)
     while (getline (fd, line) != EOF) {
        iferr (call parse_date (line, month, day, year)) {
           if (errcode () == 1)
              call erract (EA_WARN)
           else
              call erract (EA_ERROR)
        } else {
           if (year < 50)
              year = year + 2000
           else if (year < 100)
              year = year + 1900
           # Formula from Van Flandern & Pulliken
           # Valid for dates after March 1900
           date = 367 * year - 7 * (year + (month + 9) / 12) / 4 +
                  275 * month / 9 + day + 1721014
           call printf ("%d%d%d is julian date %d\n")
           call pargi (month)
           call pargi (day)
           call pargi (year)
           call pargi (date)
        }
     }
end
```

(Continued...)

Example 3.5: Using the `errcode` and `erract` Procedures.

```
procedure parse_date (line, month, day, year)

char    line[ARB]       # i: String containing date
int     month           # o: Month
int     day             # o: Day
int     year            # o: Year (0 <= year <= 99)
#--
int     ic, nc, idate, date[3]

int     ctoi

begin
     ic = 1
     do idate = 1, 3 {
        nc = ctoi (line, ic, date[idate])
        if (nc == 0)
           call error (1, "Part of date is missing")
        while (line[ic] < 0 || line[ic] > 9) {
           if (line[ic] == EOS)
              call error (1, "Part of date is missing")
           ic = ic + 1
        }
     }
     month = date[1]
     day = date[2]
     year = date[3]
end
```

Example 3.5 (Continued): Using the `errcode` and `erract` Procedures.

Error Handlers

In addition to handling an error locally with an `iferr` block, it is also possible to handle an error globally by posting an error handling procedure. The purpose of posting an error handling procedure is to restore the computer to a known state when a program exits abnormally with an error. Error handlers can be posted with `onerror` or `xwhen`. Error handlers posted with `onerror` are called whatever the type of error that occurred. Also, the program will not continue executing after an error handler is called. Error handlers posted with `xwhen` are associated with a particular error code and execution of the program will continue after the error handler exits.

Call	***Error Handling***
`onerror (proc)`	Post an error handler
`xwhen (signal, handler, old_handler)`	Post and error handler
`zsvjmp (jumpbuf, status)`	Save system state
`zdojmp (jumpbuf, code)`	Jump

Table 3.3: Error Handlers.

The procedure `onerror()` has a single argument, the name of the error handling procedure. The error handling procedure must be declared external with the `extern` statement. If an error occurs in the program after the error handling procedure is posted, the error handling procedure will be called before the normal program cleanup. The error handling procedure will be passed a single argument, the error code passed to the error procedure. Other information necessary for the error handling procedure should be passed through the common block.

The following example shows how an error handling procedure is posted by `onerror` and what it looks like. The first procedure, `term_init`, opens the terminal for reading and writing and puts the terminal in raw mode. The second procedure, `term_end`, closes the terminal and restores the terminal from raw mode. Since leaving the terminal in raw mode after the program exits will cause a lot of problems, `term_init` posts an error handling routine to restore the terminal. The error handling routine simply calls the normal exit procedure, `term_end`. Note the file descriptors are set to `NULL` after they are closed. This is so that if an error occurs in the program after `term_end` is called, the error handling routine will not try to close the same file descriptors twice.

```
include <fset.h>

# TERM_INIT -- Initializa the terminal for raw mode i/o

procedure terminit (in, out)

int    in       # o: File descriptor used to read terminal
int    out      # o: File descriptor used to write to terminal
#--
int    ttyin, ttyout
common /term/  ttyin, ttyout

extern  term_error
int     ttopen()

begin
     # Open file descriptors used for terminal i/o

     in = ttopen ("dev$tty', READ_ONLY)
     out = ttopen ("dev$tty", APPEND)

     ttyin = in
     ttyout = out

     # Put terminal in raw mode

     call fseti (ttyin, F_RAW, YES)

     # Set up error exit routine

     call onerror (term_error)

end

procedure term_end()

#--
int    ttyin, ttyout
common /term/ ttyin, ttyout

begin
     if (ttyin != NULL) {
        call fseti (ttyin, F_RAW, NO)
        call close (ttyin)
        ttyin = NULL
     }
     if (ttyout != NULL) {
        call close (ttyout)
        ttyout = NULL
     }
end
```

(Continued...)

Example 3.6: An Error Handling Procedure.

```
# TERM_ERROR -- Procedure called on error exit

procedure term_error (status)

int     status          # i: Error code
#--

begin
     if (status > 0)
         call term_end
end
```

Example 3.6 (Continued): An Error Handling Procedure.

There are two kinds of errors that can occur during the execution of a program, *synchronous* and *asynchronous* errors. Synchronous errors occur when the task calls the `error()` procedure. These are synchronous errors because the task is in a known state when the error condition occurs. As a result, error handling is relatively simple. Synchronous errors can be caught by an `iferr` block, as described previously. Asynchronous errors, also known as exceptions, occur when the hardware detects an illegal condition. Because these errors are detected by the hardware and not by the program, the program is in an unknown state when the error occurs. This makes error handling more difficult. IRAF divides all asynchronous errors into four kinds: access violations, arithmetic errors, interrupts, and interprocess communication errors. IRAF has a default exception handler for all asynchronous errors. The default exception handler does a non-local jump to the IRAF `main` routine, prints an error message, performs task cleanup such as closing files, and exits normally. If this default behavior is not sufficient, a program can post its own error handler by calling `xwhen`.

`xwhen` takes three arguments. The first two are inputs and the third is an output. The two inputs are a symbolic constant indicating the error to be trapped and the address of the error handling procedure. The symbolic constants are defined in `xwhen.h`. The address of a procedure is computed from the function `locpr`. The output is the address of the old error handling procedure. This is provided so that the program can restore the old error handler later or so that it can chain error handlers by calling the old error handler when the error handler exits. The error handling procedure has two arguments. The first is an input, the symbolic constant representing the error code. The second is an output, the address of error handler to call after the error handler returns. If the error handler does not chain to another error handler, the second parameter should be set to the symbolic constant `X_IGNORE`.

Usually an error handler resumes execution of a program by performing a non-local jump. A non-local jump is performed by calling two procedures, `zsvjmp` and `zdojmp`. `Zsvjmp` saves the current state of the computer in an array. The length of this array is hardware dependent and is specified by a symbolic constant in `config.h`. `Zdojmp` takes the array generated by `zsvjmp` and uses it to restore the computer state to what it was when `zsvjmp` was called. Thus the program calls `zdojmp` and returns from `zsvjmp`. `Zsvjmp` has a second argument, `status`, which indicates whether the return from `zsvjmp` is a normal return or a result of a call of `zdojmp`. The value returned from `zsvjmp` is the second argument of `zdojmp` or `OK` if `zdojmp` was not called. When using non-local jumps, the condition which caused the error must not be repeated or the program will go into an infinite loop.

Example 3.7 shows how to post an error handler with `xwhen`. Only two of the four asynchronous errors are trapped, access violations and arithmetic errors. The old error handlers are saved in local variables so that they can be restored at the end of the subroutine. The system state is saved by procedure `zsvjmp`. The length of the array is given by a symbolic constant defined in the header file `config.h`. The procedure then calls `do_cmp`, which executes the command read from the file. If an access violation or arithmetic error occurs while the command is being executed, the program will call `err_cmd`. This procedure restores the system state by calling `zdojmp`. The array with the system state is passed through a common block. The program then returns from `zdojmp` and prints the error message.

```
include <xwhen.h>
include <config.h>

# BATCH_CMD -- Execute a series of commands in a file

procedure batch_cmd (file)

char    file[ARB]      # i: File name
#--
int     jumpbuf[LEN_JUMPBUF]
common  /jmpcom/   jumpbuf

char    command[SZ_LINE]
int     fd, nc, status
pointer acv_handler, arith_handler, junk

extern  err_cmd
int     open(), geline()
pointer locpr()

begin
     # Open batch file

     fd = open (file, READ_ONLY, TEXT_FILE)

     # Post error handlers

     call xwhen (X_ACV, locpr(err_cmd), acv_handler)
     call xwhen (X_ARITH, locpr(err_cmd), arith_handler)

     repeat {
        call zsvjmp (jumpbuf, status)
        if (status != OK) {
           call printf (STDERR, "Error in following command:\n")
           call printf (STDERR, "%s\n')
           call pargstr (command)
           call flush (STDERR)
        }
        # Exit on end of file, skip blank lines
        nc = getline (fd, command)
        if (nc == EOF)
           break
        else if (nc == 1)
           next
        # Strip trailing newline from command
        command[nc-1] = EOS
        call do_cmd (command)
     }
     # Restore old handlers
     call xwhen (X_ACV, acv_handler, junk)
     call xwhen (X_ARITH, arith_handler, junk)
end
```

(Continued...)

Example 3.7: Posting an Error Handler with xwhen.

```
# ERR_CMD -- Error handler for batch processor

procedure err_cmd (code, nxt_handler)

int     code         # i: Error code which triggered this exception
int     nxt_handler  # o: Handler called after this handler exits
#--
int     jumpbuf[LEN_JUMPBUF]
common /jmpcom/  jumpbuf

begin
     # Resume execution at zsvjmp
     nxt_handler = X_IGNORE
     call zdojmp (jumpbuf, code)
end
```

Example 3.7 (Continued): Posting an Error Handler with xwhen.

CHAPTER 4:

Making a Task

This chapter describes how to make SPP source into a working program. In most cases, this means creating an IRAF task. That is, a command to be executed in the IRAF cl. Inherent in creating the task is compiling and linking the source to create an executable program. We also describe the conventional structure of packages of tasks in the cl.

Program Structure

An SPP source file may contain any number of `procedure` declarations, zero or one `task` statements, any number of `define` or `include` statements, and any number of `help` text segments. By convention, global definitions and include file references should appear at the beginning of the file, followed by the task statement, if any, and the procedure declarations.

The `task` Statement

The `task` statement is used to make an IRAF task. That is, a command recognized in the cl as an executable program. Primarily, this is accomplished with the `task` statement, part of the SPP code. A file need not contain a task statement, and may not contain more than a single task statement. Files without task statements are separately compiled to produce object modules, which may subsequently be linked together to make a task, or which may be installed in a library. A single *physical task* (*ptask*) may contain one or more *logical tasks* (*ltask*s). These tasks need not be related. Several ltasks may be grouped together into a single ptask merely to save disk storage, or to minimize the overhead of task execution. Logical tasks should communicate with one another only via disk files, even if they reside in the same physical task.

```
task  ltask1, ltask2, ltask3 = proc3
```

The `task` statement defines a set of ltasks, and associates each with a compiled procedure (see Example 4.1). If only the name of the ltask is given in the task statement, the associated procedure is assumed to have the same name. A file may contain any number of ordinary procedures which are not associated (directly) with an ltask. The source for the procedure associated with a given ltask need not reside in the same file as the task statement. An ltask associated procedure *must not* have any arguments. An ltask procedure gets its parameters from the cl via the cl interface. Most commonly used are the `clget`*T*`()` procedures. The `clput`*T*`()` procedures may be used to change the values of parameters.

```
task alpha, beta epsiol=eps

procedure alpha()

int npix, clgeti ("npxi")
real lcut, clgetr()
char file[SZ_FNAME]

begin
     npix = clgeti ("npix")
     lcut = clgetr ("lower_cutoff")
     call clgstr ("input_file", file, SZ_FNAME)
     .
     .
```

Example 4.1: Making an IRAF Task.

An IRAF task be run by the cl or called from the command interpreter provided by the host operating system (the shell or DCL for example) without change. Parameter requests and I/O to the standard input and output will function properly in both cases. When running without the cl, of course, the interface is much more primitive. To run an IRAF task directly, without the cl begin by simply running the program. Such stand-alone operation is especially useful when debugging. The task will sense that it is being run without the cl and issue a prompt, see Example 4.2.

```
> ?
alpha beta epsilon
> alpha
npix: (response)
lower_cutoff: (response)
input_file: (response)
   ltask alpha continues
> bye
```

Example 4.2: Parameter Prompting.

Every IRAF task has some special commands built in. The command `?` will list the names of the ltasks recognized by the interpreter. The command `bye` is used to exit the interpreter, returning to the host command interpreter. To execute a host command at the `>` prompt, precede the command by an exclamation point (`!`).

Compiling and Linking

The steps necessary to transform SPP code into a working program are:

1. Preprocesses SPP to Ratfor and then to Fortran
2. Translate Ratfor to Fortran
3. Compile Fortran to object code
4. Link object with IRAF and system libraries resulting in executable binary

These could be performed individually and manually. However, to provide a simple and portable mechanism (remember that the goal is for IRAF to be host independent), IRAF provides tools to do this. While the tools are straightforward for simple cases, they provide the power to handle more sophisticated operations.

mkpkg

The *mkpkg* utility is used to make or update IRAF packages or libraries. It is the highest level means of compiling and linking in the IRAF environment. There is a `mkpkg` command available in the cl as well as the host environment. Usage is identical in either case, except that the details of when a particular argument may need to be quoted will vary depending on the command language used. It is analogous to the `make` utility in Unix in that it not only performs compilation and linking, but it also performs enough revision control to perform only the needed updates. While mkpkg uses several command line options to control its operation, the particular actions to perform are specified in a text file, the *mkpkg file*.

This section provides only the briefest introduction to mkpkg. For a complete discussion see the help pages in the cl by typing `help mkpkg`.

`mkpkg` provides two major facilities: a library update capability and a macro preprocessor. The macro preprocessor provides symbol definition and replacement, conditional execution, and a number of built-in commands. The usefulness of these facilities is enhanced by the ability of `mkpkg` to update entire directory trees, or to enter the hierarchy of `mkpkg` descriptors at any level. For example, typing `mkpkg` in the root directory of IRAF will make or update the entire system, whereas in the `iraf$sys` directory mkpkg will update only the system libraries, and in the `iraf$sys/fio` directory mkpkg will update only the **fio** portion of the system library `libsys.a`.

The mkpkg utility is quite simple to use to maintain small packages or libraries, despite its full complexity of the discussion which follows. The reader is encouraged to study several examples of working mkpkg files before reading further; examples will be found throughout the IRAF system. The mkpkg files for applications packages tend to be very similar to one another, and it is quite possible to successfully copy and modify the mkpkg file from another package without studying the reference information given here. A very simple mkpkg file is shown below:

```
$omake imtoal.x
$link imtoal.o
```

This will compile and link the SPP program in the file named `imtoal.x`, resulting in an executable program in the file `imtoal.e`. Note the `$` characters beginning the lines. The source file (`imtoal.x`) is assumed to have a `task` statement. This type of mkpkg file would be used for the most simple applications with a small number of procedures in one or at most a few source files and requiring no libraries other than the IRAF system libraries. A slightly more complicated example (Example 4.3) maintains a library for a small package of tasks.

```
$call  relink
$exit

relink:
     $update tutor.a
     $call   linktutor
     ;

linktutor:
     $omake  x_tutor.x
     $link   x_tutor.o tutor.a -o xx_tutor.e
     ;

tutor.a:
     arrows.x     <gset.h>
     bones.x      <imhdr.h>
     filter.x     <imhdr.h>
     hello.x      <gset.h>
     ;
```

Example 4.3: MKPKG File for Maintaining Small Library.

This introduces two features of `mkpkg`: calling modules and maintaining a library. The `$call` statement allows different blocks of statements to be executed. These are named by labels terminated by a colon. Note that each module block must terminate with a semicolon. Otherwise, the following block will also be executed. A block may also be called directly as an entry point by specifying the label name on the `mkpkg` command line, for example:

```
mkpkg update
```

The `$update` command maintains the library of procedures for the package (`tutor` in this case). The label `tutor.a` delimits the "dependencies" section which lists include files used by each source file. A source file will be compiled if either the source itself or any of the include files upon which it depends has changed since the last update. Note also the `-o` option on the `$link` statement, specifying the name of the output executable binary file. A library may include references to libraries in other directories, using the `@` syntax. These are mkpkg file in the specified directory. A `$link` statement may reference other libraries in addition to the implicit IRAF system libraries and local libraries defined in the current mkpkg. If these reside in the IRAF system (or an external package) library directory, they may be referenced using a `-l` prefix. For example:

```
$link x_stplot.o stplot.a -ltbtables -lxtools -o xx_stplot.e
```

Most often, an installed package will maintain binary executables in a common directory. These are maintained using mkpkg with the `$move` command:

```
install:
$move xx_stplot.e stsdasbin$x_stplot.e
;
```

This example is from the STSDAS external package, hence the symbol `stsdasbin` pointing to the location of the binary. Note that the executable is renamed in the move. The original has a prefix `xx_` while the target file has the prefix `x_`. This is conventional for tasks installed in packages. This permits the package to be remade without disturbing the installed binary until necessary. Even though the binaries are installed in a directory separate from the package directory, the tasks are defined pointing to the package directory as the location of the executable.

xc

The xc utility is a machine independent program for compiling and linking IRAF tasks or files. The xc utility may also be used to compile or link non-IRAF files and tasks. The VMS version of xc supports all of the important flags except `-D` which VMS C doesn't support in any way. It can be used to generate Fortran from SPP or Ratfor code, to compile any number of files, and then link them if desired. xc accepts and maps IRAF virtual filenames, but since it is a standalone utility (i.e., it need not run in the cl), the environment is not passed, hence logical names for directories cannot be used. Table 4.1 shows the IRAF virtual file name extensions that are supported by xc:

Extension	*File Type*
`.x`	SPP code
`.r`	Ratfor code
`.f`	Fortran code
`.c`	C code
`.s`	Macro assembler code
`.o`	Object module
`.a`	Library file
`.e`	Executable image

Table 4.1: XC-supported Virtual File Name Extensions.

xc is available both in the cl, via the foreign task interface, and as a standalone task callable in the host system. Usage is equivalent in either case. The simple example below compiles and links the source file `mytask.x` to produce the executable `mytask.e`.

```
xc mytask.x
```

The next example compiles but does not link `mytask.x` and the support file `util.x`.

```
xc -c file.x util.x
```

Now link these for debugging and link in the library `libdeboor.a` (the DeBoor spline routines in the `lib` directory).

```
xc -x file.o util.o -ldeboor
```

xc is often combined with mkpkg to automatically maintain large packages or libraries. For complete information on `xc` see the help pages in the cl by typing `help xc`.

Generic Preprocessor

The *generic preprocessor* is provided in addition to SPP to convert a generic operator into a set of type specific operators. Since Fortran requires that the data types of the calling and called procedure arguments match, it is the programmer's responsibility to ensure this. The generi preprocessor makes this easier. By coding only generic operators, the programmer only has to maintain a single piece of code, reducing the possibility of an error, and greatly reducing the amount of work.

Note that this section is taken substantially verbatim from the help text for the `generic` task. Type `help generic` in the cl to see it. The term "operator" here in general refers to an SPP procedure or function. The generic preprocessor takes as input files written in either the IRAF SPP language or C with embedded preprocessor directives and keywords. The calling sequence for the preprocessor (on the Unix system) is as follows:

```
generic [-t types] [-p prefix] [-o outfile] file [file...]
```

Any number of files may be processed.

Flags

The following (optional) flags are provided to control the types and names of the generated files:

- `-k` Allow the output files generated by `generic` to overwrite (clobber) any existing files.
- `-o` If an output filename is specified with the -o flag, only a single input file may be processed. Any `$t` sequences embedded in the output file name will be replaced by the type "suffix" character to generate the filenames of the type specific files in the generic family. If no `$t` sequence is given, the type suffix is appended to the filename. If no -o output filename is given, the names of the output files are formed by concatenating the type suffix to the root of the input filename.
- `-p` An optional prefix string to be added to each file name generated. Provided to make it convenient to place all generated files in a subdirectory. If the name of the file(s) being preprocessed is `aadd.x`, and the prefix is `d/`, the names of the generated files will be `d/aadds.x`, `d/aaddi.x`, `d/aaddl.x`, and so on.
- `-t` Used to specify the data types of the files to be produced. The default value is `silrdx`, meaning types SHORT through COMPLEX. Other possible types are `bu`, i.e., unsigned byte and unsigned short. The generic preprocessor does not support type boolean.

Directives

The action of the preprocessor is directed by placing $*xxx* directives in the text to be processed. The identifiers `INDEF` and `PIXEL` are also known to the preprocessor, and will be replaced by their type specific equivalents. `INDEF` will be replaced by `INDEFS`, `INDEFI`, etc., and

`PIXEL` will be replaced by `short`, `int`, `real`, etc. in the generated text. Comments (#... or `/*`...`*/`), quoted strings ("...") and escaped lines (^%) are passed on unchanged.

The generic operator shown in Example 4.4 computes the square root of a vector. The members of the generic family would be called `asqrs`, `asqri`, and so on.

```
# ASQR -- Compute the square root of a vector (generic)

procedure asqrt$t (a, b, npix)

PIXEL  a[npix], b[npix]
int    npix, i

begin
     do i = 1, npix {
        if (a[i] < 0$f || a[i] == INDEF)
           b[i] = INDEF
        else {
           $if (datatype != rdx)
             b[i] = sqrt(double(a[i]))
           $else
             b[i] = sqrt(a[i])
           $endif
        }
     }
end
```

Example 4.4: Generic Operator.

The operators are explained in the following list.

- `$/`*text*`/` - The text enclosed by the matching slashes is passed through unchanged.
- `$t` - The lowercase value of the current type suffix character (one of the characters `bucsilrdx`).
- `$T` - The uppercase value of the current type suffix character (one of the characters `BUCSILRDX`).
- *digits*`$f` - Replaced by *digits*`.0` if the current type is real, by *digits* `.0D0` if the current type is double, by `(`*digits,digits*`)` if the type is complex, or by *digits* for all other datatypes.
- `$if` - Conditional compilation. Two forms of the `$if` statement are implemented:
  - `$if (datatype ==` *t*`)` or
    `$if (datatype !=` *t*`)` where *t* is one or more of the data type characters (`s`, `i`, `l`, `r`, `d`, etc.).

- `$if (sizeof(`t_1`)` *op* `sizeof(`t_2`))` where t_1 and t_2 are type suffix characters (`silrd`, etc.), and where *op* is one of the relational operators `==`, `!=`, `<=`, `<`, `>=`, or `>`.

Nesting is permitted. Conditional statements need not be left justified, i.e., white space may be placed between the beginning of the line (`BOL`) and a `$xx` preprocessor directive.

- `$$if` - Replaced by `$if`. Not evaluated until the second time the file is processed. These may include an `$else` or `$$else` block executed if the `$if` condition was false and should be terminated by an `$endif` or `$$endif`.
- `TY_PIXEL` - Replaced by `TY_INT`, `TY_REAL`, and so on.
- `SZ_PIXEL` - Replaced by `SZ_INT`, `SZ_REAL`, and so on.
- `PIXEL` - Replaced by the datatype keyword of the file currently being generated (`int`, `real`, etc.).
- `XPIXEL` - Replaced by the defined type (`XCHAR`, `XINT`, etc.). Used in generic C programs which will be called from the subset preprocessor, and which must manipulate the subset preprocessor datatypes.
- `$PIXEL` - Replaced by the string `PIXEL` (used to postpone substitution until the next pass).
- `INDEF` - Replaced by the `INDEF` symbol for the current data type (`INDEFS`, `INDEFI`, `INDEFL`, `INDEF`, or `INDEFX`).
- `$INDEF` - Replaced by the string `INDEF`.

Doubly Generic Operators

The preprocessor can also be used to generate doubly generic operators (operators which have two type suffixes). A good example is the type conversion operator `acht`*xy*, which converts a vector of type *x* to a vector of type *y*. If there are seven datatypes (`c`, `s`, `i`, `l`, `r`, `d`, `x`), this generic family will consist of 49 members. Doubly generic programs are preprocessed once to expand the first suffix, then each file generated by the first pass is processed to expand the second suffix. On the Unix system, this might be done by a command such as

```
generic acht.x; generic -p dir/ acht[silrd].x
rm acht[silrd].x
```

This would expand `acht` in the current directory (generating five files), then expand each of the `acht$t` files in the subdirectory `dir/`, creating a total of 25 files in the subdirectory. The final command removes the 5 intermediate files.

For an example of double generic code, see source for the **vops** procedure family `acht()` in `vops$acht.gx`.

Parameter Files

Each logical task that reads parameters from the cl using **clio** may specify attributes of those parameters using a *parameter file*. Parameter attributes include the name, data type, default value, and others. The file is a text file created by the programmer and should be located in the same directory as the physical task. There is one parameter file for each logical task. Its root name is the same as the name of the associated logical task and there is an extension `.par`. Each task parameter is described by an entry in the parameter file consisting of positional fields separated by commas:

```
name,type,mode,value,minimum,maximum,prompt
```

All of the fields after *value* are optional. Fields may be omitted with adjacent commas.

- ***name*** - The parameter name as known to the cl and to the application task. This is the value of the string used in the *clio* `clgetT()` and `clputT()` procedures. Examples of code to read task parameters are in “Interaction with the cl — clio” on page 45.
- ***type*** - The data type of the parameter. That is, the type as known to the cl. Note that this *need not* match the data type of the corresponding SPP variable used in the application, but it makes sense to do so. This attribute takes a string value representing the type.

String Value	*Data Type*
`b`	Boolean
`i`	Integer
`r`	Floating point
`s`	String
`f`	File name
`struct`	Structure
`gcur`	Graphics cursor
`imcur`	Image cursor
`pset`	Parameter set

Table 4.2: cl Parameter Data Types.

Note that there is no distinction between sizes of numeric parameters; i.e., there is no concept of a “short” integer or a “double precision” floating point parameter. The character `*` preceding a type attribute indicates a “list structured” parameter. The cursor parameters must be declared as list structured: `*gcur` and `*imcur`. A pset specifies a pointer to another parameter file. See the document *Named External Parameter Sets in the CL* [Tody86] for a complete description (on line in the IRAF file `doc$pset.ms`).

- ***mode*** - The manner in which the cl handles *prompting* and *learning* of the parameter.
    - `q` **- Q**uery the user each time. Prompt for the parameter value even if the default is not null.
    - `l` **- L**earn the value of the parameter. Store the value as the new default value.
    - `a` **- A**utomatically take the mode of the next higher level in the cl, such as the task, package or the cl itself.
    - `h` **- H**ide any prompting for the parameter value unless the cl cannot resolve the default value.
- ***value*** - The default or initial value for the parameter.
- ***minimum*** - The minimum acceptable value for the parameter. If the entered value is smaller, the cl will prompt again. In addition, a string

type parameter may be defined with an “enumeration string” as the minimum value. The parameter’s value may then take on *only* one of the enumerated values. The enumeration string is enclosed in quotes and each enumerated value should be separated by pipe characters (`|`), for example:

```
color,s,h,"white","white|black|red|green|blue",,
"Graphics color"
```

- ***maximum*** - The maximum acceptable value for the parameter. If the entered value is larger, the cl will prompt again.
- ***prompt*** - The string printed by the cl as part of the prompt to describe the parameter. This may be enclosed in double quotes, required if the string contains commas.

There are other fields as well that are slightly beyond this brief explanation. For a more detailed explanation of parameter files and parameter fields, see the *CL Programmer’s Manual* [Downey82], a copy of which is on line in the file `iraf$doc/clman.ms`.

Package Structure

Tasks in IRAF (and external packages such as STSDAS) are organized by *package* in the cl. The structure directories containing the source and run-time files reflects the package structure apparent from the cl. For example, in the case of STSDAS, each package resides in a directory under the `stsdas` root directory just as the STSDAS packages are organized under the `stsdas` package in the cl. There are several files common to the package as a whole and several similar files required for each task in the package. These files need to be modified when installing a new task. The required common files in the package directory are:

- ***package.cl*** - Package cl procedure, cl task definitions
- ***x_package.x*** - SPP task definitions
- ***mkpkg*** - How to build the package

In the above file names, the name of the package is used in place of *package*. For example, the `playpen` package in STSDAS is in the directory `stsdas$pkg/playpen` and the procedure script is called `playpen.cl`. In addition, documentation files exist in the package level directory as well as a `doc` directory containing individual help files for the tasks in the package.

- ***package.hd*** - Help database pointers
- ***package.hlp*** - Package level help
- ***package.men*** - Package menu, one line task descriptions
- ***doc*** - Directory containing task help files

Tasks in the Package

Each task has additional files, the type of which depends on the nature of the task. These files would be added when you install a new task. Each task must also have entries in the package files. A cl procedure task requires only a *task*`.cl` file in the package directory, containing the cl statements and parameter definitions. For example, `disconlab.cl` in the `playpen` package. It also requires a *task*`.hlp` file in the `doc` subdirectory. An SPP (physical) task requires SPP source, at least one source file, by convention called `t_`*task*`.x` (with *task* replaced by the task name) `playpen$t_wcslab.x`, for example. Additional source files may reside in the package directory or in subdirectories. The task may use an *include* (header) file with the name *task*`.h`, `playpen$wcslab.h`, e.g. Each task requires a parameter file (unless it is a script, defined by a `.cl` file), *task*`.par`, containing definitions of the task parameters, such as `playpen$wcslab.par`. The `doc` directory contains the task help files, one for each task in the package.

Implementation

The procedure, then, is to develop the application in a private directory with a structure similar to the intended target package. Development should be done in a local user directory rather than the system directories, not even the development system. Use an existing package as an example of how to proceed. When you are ready to install the package, copy the task files to the intended package directory and edit the existing package files to include references to the new package. Run `mkpkg` to rebuild the package with the changes (the added task). When you are satisfied that things work, run `mkpkg install` to move the executable to the appropriate binaries directory.

APPENDIX A:

Predefined Constants

The SPP language includes a number of predefined symbolic constants and macro definitions. These allow SPP programs to use keyword names for commonly used values. Included are various machine dependent constants describing the hardware and data types. Other symbolic constants are used for basic file I/O. All predefined constants are of type integer. The include files described here are automatically included when an SPP program is compiled.

Language Definitions

The value of these definitions may vary from one machine and host operating system to another. SPP code using the symbolic constants need not be modified, however, when porting software. The include file defining these macros is `hlib$iraf.h`. However, it is included implicitly by xc and the definitions are available at all times. You do not need to include it explicitly.

Generic Constants

Constant	*Meaning*
`ARB`	Arbitrary; array dimension
`BOF`	Beginning of file
`BOFL`	Beginning of file
`EOF`	End of file
`EOFL`	End of file
`EOS`	End of string
`EOT`	End of tape
`ERR`	Error status return
`NO`	Opposite of YES (int flag)
`YES`	Opposite of NO (int flag)
`OK`	Status return, opposite of ERR
`NULL`	Invalid pointer

Table A.1: Generic Constants.

Data Type Sizes

These macros define the sizes of the fundamental SPP data types in units of `char`, the smallest addressable word.

Macro	*Size Defined*
`SZ_BOOL`	Number of `chars` per `bool`
`SZ_CHAR`	Number of `chars` per `char`
`SZ_SHORT`	Number of `chars` per `short`
`SZ_INT`	Number of `chars` per `int`
`SZ_LONG`	Number of `chars` per `long`
`SZ_REAL`	Number of `chars` per `real`
`SZ_DOUBLE`	Number of `chars` per `double`
`SZ_COMPLEX`	Number of `chars` per `complex`
`SZ_POINTER`	Number of `chars` per `pointer`
`SZ_STRUCT`	Number of `chars` per `struct`
`SZ_USHORT`	Number of `chars` per `ushort`
`SZ_FNAME`	Maximum number of `chars` in a file name
`SZ_LINE`	Maximum number of `chars` in a line
`SZ_PATHNAME`	OS dependent file name size
`SZ_COMMAND`	Maximum size of command block

Table A.2: Sizes of SPP Data Types.

Data Type Codes

The data type codes are used, for example, in dynamic memory allocation, in which it is necessary to know how many bytes each value occupies. The sizes are in units of `char`s, where a `char` *usually* occupies two bytes. The lines shown Example A.1 will allocate a `short` and `double` buffer, each of `size` elements. The resulting memory buffers will consist of different numbers of bytes, but will logically contain the same number of elements.

```
pointer sbuf, dbuf
int     size
begin
     .
     .
     call malloc (sbuf, size, TY_SHORT)
     call malloc (dbuf, size, TY_DOUBLE)
     .
     .
     call mfree (sbuf, TY_SHORT)
     call mfree (dbuf, TY_DOUBLE)
end
```

Example A.1: Using Data Type Codes.

Code	*Data Type*
TY_BOOL	Boolean
TY_CHAR	Character
TY_SHORT	Short integer
TY_INT	Integer
TY_LONG	Long integer
TY_REAL	Single precision real
TY_DOUBLE	Double precision real
TY_COMPLEX	Complex
TY_POINTER	Pointer
TY_STRUCT	Structure
TY_USHORT	Unsigned short integer (for image I/O only)
TY_UBYTE	Unsigned byte (for image I/O only)

Table A.3: Data Type Codes.

File and Image I/O

The macros described in this section are used in accessing text files, binary files, and images.

File Types

The file type specifies the kind of file to be read or written.

Macro	*File Type*
TEXT_FILE	Plain text (ASCII)
BINARY_FILE	Binary, host dependent
DIRECTORY_FILE	Directory
STATIC_FILE	
SPOOL_FILE	Internal, no permanent location
RANDOM	
SEQUENTIAL	

Table A.4: File Types.

File I/O Modes

The mode parameters are used on opening the file and specify the manner in which the file will be accessed.

Parameter	***I/O Mode***
`READ_ONLY`	Read only, no output
`READ_WRITE`	Read and write
`WRITE_ONLY`	Write only, no input
`APPEND`	Append to an existing file
`NEW_FILE`	New file
`TEMP_FILE`	Temporary file, deleted at task end
`NEW_COPY`	Copy of an existing file
`NEW_IMTE`	Alias for `NEW_FILE`
`NEW_STRUCT`	
`NEW_TAPE`	

Table A.5: File I/O Modes.

I/O Streams

Stream Name	***Contents***
`CLIN`	Standard input of the physical task
`CLOUT`	Standard output of the physical task
`STDIN`	Standard input
`STDOUT`	Standard output
`STDERR`	Standard error
`STDGRAPH`	Standard graph (usually a graphics terminal)
`STDIMAGE`	Standard image (usually an image display)
`STDPLOT`	Standard plot (usually a hardcopy plotter)

Table A.6: I/O Streams.

The following example (Example A.2) opens two files. The first statement opens for reading an existing text file whose name is specified in the `char` variable `fname`. The second statement opens a new image whose name will be the string in `imname`.

```
int       fp              # File descriptor
pointer   ip              # Image descriptor
char      fname[SZ_FNAME] # File name
char      fname[SZ_FNAME] # Image name

int       open()
pointer   immap()

begin
     .
     .
     # Open the text file
     fp = open (fname, READ_ONLY, TEXT_FILE)

     # Open the image
     ip = immap (imname, NEW_FILE, 0)
     .
     .
     call close (fp)
     call imunmap (ip)
end
```

Example A.2: Opening Files.

Indefinites

Indefinite values may be used to flag data for specific purpose, to exclude from further consideration or indicate an error, for example. Each SPP data type has its own indefinite value. The actual value of the various indefinites may be different, so the appropriate one must be used. In addition, there are macro functions to test values against `INDEF`.

Values

Value	*Data Type*
`INDEFS`	Short integer
`INDEFL`	Long integer
`INDEFI`	Integer
`INDEFR`	Single precision real
`INDEFD`	Double precision real
`INDEFX`	Complex
`INDEF`	Alias for `INDEFR`

Table A.7: Indefinite Values.

Logical Functions

These macros (Table A.8) define functions to test values against indefinite. There is a macro for each SPP data type. Example A.3 shows how to execute a block of code in the case where a particular value is indefinite.

Function	*Data Type*
`IS_INDEFS()`	Short integer
`IS_INDEFL()`	Long integer
`IS_INDEFI()`	Integer
`IS_INDEFR()`	Single precision real
`IS_INDEFD()`	Double precision real
`IS_INDEFX()`	Complex
`IS_INDEF()`	Alias for `IS_INDEFR()`

Table A.8: Logical Functions.

```
short    sval  # A short integer
real     rval  # A single precision real
.
.
begin
     .
     .
     if (IS_INDEF(rval)) {
        # If rval is indefinite, execute this block
        .
        .
     }
     if (IS_INDEFS(sval)) {
        # If sval is indefinite, execute this block
        .
        .
     }
end
```

Example A.3: Executing Code with INDEF Values.

Pointer Conversion

These macros are used for pointer conversions in data structures. Since all dynamically allocated arrays share the same memory (implemented by Fortran `COMMON` and `EQUIVALENCE`), the correct offset to data types having different word sizes must be computed. These macros perform that computation. Note that there is no `P2I` or `P2R` since these are assumed to be the same size according to the Fortran standard. See “Macro Definitions” on page 16 for more discussion of SPP macros.

Macro	***Purpose***
`P2C()`	Convert pointer to character
`P2S()`	Convert pointer to short integer
`P2L()`	Convert pointer to long integer
`P2D()`	Convert pointer to double precision real
`P2X()`	Convert pointer to complex

Table A.9: Pointer Conversion Macros.

The following example from `lib$gio.h` is part of the definition of the **gio** data structure that maintains information about a plot. It defines (among other things) a string containing a label format. This is stored in a dynamically allocated `char` array.

```
define GL_AXISWIDTH     Memr[$1+16]       # linewidth of axis
define GL_TICKLABELSIZE Memr[$1+17]       # char size of tick labels
define GL_TICKFORMAT    Memc[P2C($1+18)]  # printf format of ticks
```

Example A.4: GIO Data Structures.

Machine Parameters

These macros relate to values specific to the host system architecture. These are defined in `hlib$mach.h` and must be included with the following statement if they are to be used in code:

```
include <mach.h>
```

Parameter	***Contents***
SZB_CHAR	Machine bytes per `char`
SZB_ADDR	Machine bytes per address increment
SZ_VMPAGE	Page size (1 if no virtual memory)
MAX_DIGITS	Maximum digits in a number
NDIGITS_RP	Number of digits of real precision
NDIGITS_DP	Number of digits of precision (double)
MAX_EXPONENT	Maximum exponent, base 10
MAX_EXPONENTR	Maximum exponent for single precision real
MAX_EXPONENTD	Maximum exponent for double precision real

Table A.10: Machine Parameters.

Extreme Numbers

Parameter	*Contents*
MAX_SHORT	Largest short integer
MAX_INT	Largest integer
MAX_LONG	Largest long integer
MAX_REAL	Largest single precision real; anything larger is INDEF
MAX_DOUBLE	Largest double precision real
NBITS_BYTE	Number of bits in a machine byte
NBITS_SHORT	Number of bits in a short integer
NBITS_INT	Number of bits in an integer
EPSILONR	Smallest e such that $1 + e > 1$
EPSILOND	Double precision epsilon
EPSILON	Alias for EPSILONR

Table A.11: Extreme Numbers.

Byte Swapping

Is byte swapping needed for a 2 or 4 byte MII integer or a 4 or 8 byte IEEE floating to convert to or from MII format on this machine?

Parameter	*Contents*
BYTE_SWAP2	Byte swap 2 byte MII integer?
BYTE_SWAP4	Byte swap 4 byte MII integer?
IEEE_SWAP4	Byte swap 4 byte IEEE integer?
IEEE_SWAP8	Byte swap 8 byte IEEE integer?
IEEE_USED	Use IEEE?

Table A.12: Byte Swapping Boolean Parameters.

Mathematical Constants

Definitions of various mathematical constants are in `hlib$math.h`. Use the following statement to use the macros:

```
include <math.h>
```

Values (listed in Table A.13) are given to 20 decimal places and therefore may be assigned to `real` or `double` variables without loss of precision. However, note that they are not explicitly double precision, in certain expressions in which implicit data type conversion occurs may result in truncation of precision. The definitions are from Abramowitz and Stegun, *Handbook of Mathematical Functions*, Chapter 1 [Abramowitz65].

Constant	***Value***
SQRTOF2	$\sqrt{2}$
E	e
EXP_PI	e^{π}
LN_2	$\ln 2$
LN_10	$\ln 10$
LN_PI	$\ln \pi$
LOG_E	$\log e$
PI	π
TWOPI	2π
FOURPI	4π
HALFPI	$\pi/2$
SQRTOFPI	$\sqrt{\pi}$
RADIAN	radian $(180°/\pi)$
RADTODEG	Convert radians to degrees
DEGTORAD	Convert degrees to radians
GAMMA	γ (Eulers' Constant)
LN_GAMMA	$\ln \gamma$
EXP_GAMMA	e^{γ}

Table A.13: Mathematical Constants.

Most of these are constants, except for the macros `RADTODEG` and `DEGTORAD` which convert between degrees and radians. For example the following procedure converts angles in an array from radians to degrees:

```
include <math.h>

procedure vradeg (rads, degs, nelem)

real  rads[ARB]    # Angles in radians
real  degs[ARB]    # Angles in degrees
int   nelem        # Size of array

int   i

begin
      do i = 1, nelem
         degs[i] = RADTODEG(rads[i])
end
```

Example A.5: Converting Radians to Degrees.

Note that one might alternately use a **vops** procedure to accomplish the same result.

Character and String-Related Definitions

Character Types

These macro definitions (Table A.14) test whether a single character (type `char`) is a member of a particular class of characters, lower case letter or white space, for example. They resolve to a logical (`bool`) value which may be used in boolean expressions, including conditional statements such as `while` or `for`. They are defined in `lib$ctype.h` and, if they are to be used in code, must be included with the statement:

```
include <ctype.h>
```

Macro	*Definition*
`IS_UPPER()`	Upper case letters (A-Z)?
`IS_LOWER()`	Lower case letter (a-z)?
`IS_DIGIT()`	Numeral (0-9)?
`IS_PRINT()`	Printable character (!-~)?
`IS_CNTRL()`	Control character (CTRL-A - CTRL-_)?
`IS_ASCII()`	ASCII character (values 0-127 decimal)?
`IS_ALPHA`	Alphabetic character (A-Z or a-z)?
`IS_ALNUM()`	Alphanumeric character (A-Z, a-z, or 0-9)?
`IS_WHITE()`	White space (space or tab)?
`TO_UPPER()`	Convert to upper case
`TO_LOWER()`	Convert to lower case
`TO_INTEG()`	Convert character to digit
`TO_DIGIT()`	Convert numeral to ASCII value

Table A.14: Character Types.

Note that these definitions work for ASCII, but not for EBCDIC (IBM). By using macros, this machine dependent knowledge of the character set is concentrated into a single file. For example

```
for (ip = 1;  IS_WHITE(str[ip]);  ip = ip + 1)
    ;
```

Finds the first non-white-space character in the string `str`.

Token Definitions

Tokens are the smallest recognized string fragments such as a word, number, or operator. The encoded values of the recognized tokens is defined in the include file `lib$ctotok.h`. See “Internal Formatting” on page 85.

Token Value	*Interpretation*
TOK_IDENTIFIER	[A-Za-z][A-Za-z0-9_.$]*
TOK_NUMBER	0-9][-+0-9.:xXa-fA-F]*
TOK_OPERATOR	All other printable sequences
TOK_PUNCTUATION	[:,;] or any control character
TOK_STRING	"..."
TOK_CHARCON	'\n', etc.
TOK_EOS	End of string
TOK_NEWLINE	End of line
TOK_UNKNOWN9	Control characters

Table A.15: Tokens.

VOS Library Includes

Most VOS library package have an associated include file for constants and structures unique to that package. These are the most commonly needed include files for various packages.

Package	*Include Files*
etc	time.h
fmtio	pattern.h, evexpr.h
gio	gset.h
imio	imhdr.h

Table A.16: VOS Library Includes.

APPENDIX B: Examples

Here are a few simple SPP applications. They illustrate a range of tasks including image I/O, cl I/O, dynamic memory, and graphics, including cursor interaction. They are complete, including a task statement to implement cl tasks. More examples are provided in Rob Seaman's *An Introductory User's Guide to IRAF SPP Programming* [Seaman92].

"Hello World"

One useful way to get started with a language is to build and run a simple program, before attempting to learn all the details. It often provides an introduction to the flavor of the language and its syntax and can provide a template for developing useful applications. Here is the SPP version of the common "hello world" program. It prints the text "`hello world`" on the user's terminal.

```
# Simple program to print "hello, world" on the standard output

task  hello           # CL callable task

procedure hello()     # common procedure

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

Example B.1: Hello World Example.

The text of this program would be placed in a file with the extension "`.x`" and compiled with the command xc (X Compiler) in the host system or in the IRAF cl as follows:

```
xc hello.x
```

The xc compiler will translate the program into Fortran, call the Fortran compiler to generate the object file (`hello.o`), and call the loader to link the object file with modules from the IRAF system libraries to produce the executable program. xc may be used to compile C and Fortran programs as well as SPP programs, and in general behaves very much like `cc` or `f77` (note that the `-o` flag is not required; by default the name of the output module is the base name of the first file name on the command line). The `-f` flag may be used to inspect the Fortran created by the preprocessor; this is sometimes necessary to interpret error messages from the `F77` compiler. Finally, to run the program, you may define it as a task in the cl by using the `task` statement:

```
task $hello = hello.e
```

Then run it by typing `hello`.

cl **Interaction**

Example B.2 demonstrates simple use of **clio**, reading and writing cl parameters and simple **imio**, reading an image. While the application does little significant, it illustrates a task that analyzes an image and extracts information from it.

The procedure called by the above procedure to perform the operation on the images is shown in Example B.3.

```
include <imhdr.h>

procedure bones ()

# This is a skeleton (bare bones) of a task to do something with a
# 1-dimensional image and get a single value for an answer. It writes
# to STDOUT & to a parameter. It gets an arbitrary parameter from the
# header and writes to STDOUT. 2 input parameters: image file & header param file

char     inimg[SZ_FNAME]       # Input image file name
pointer  im                    # Image descriptor
int      npts                  # Number of pixels
pointer  line                  # Pixels
char     param[SZ_LINE]        # Header parameter name
real     parval                # Parameter value
real     answer                # The result
pointer  immap(), imglir()     # Function declarations
real     imgetr()

begin
     call clgetr ("image", inimg, SZ_FNAME)     # Get input image name
     im = immap (inimg, READ_ONLY, 0)           # Open image
     npts = IM_LEN(im,1)                        # Assume 1-D image
     call clgstr ("param", param, SZ_LINE)      # Get header param name
     parval = imgetr (im, param)                # Get header parameter
     call printf ("%s = %f\n")
        call pargstr (param)
        call pargr (parval)

     # Read the data into dynamic memory
     line = imgl1r (im)
     # Use data. You can plug in Fortran subroutine for stuff and treat
     # the first argument as a REAL array
     call stuff (Memr[line], npts, answer)
     # Write answer
     call printf ("The answer is: %f\n")
        call pargr (answer)
     # Put answer in cl parameter
     call clputr ("answer", answer)
     # close the image
     call imunmap(im)
end
```

Example B.2: Simple Use of clio.

```
procedure stuff (pixels, npts, answer)

# This is a dummy applications routine for the bones task. It just
# finds the average of the input pixel vector.
real    pixels[ARB]
int     npts
real    answer
real    sigma
begin
     call aavgr (pixels, npts, answer, sigma)
end
```

Example B.3: Procedure Called by Bones.

A Simple Filter

This example (Example B.4) illustrates a simple *filter*. That is, a task that takes a file as input and produces a similar but changed file on output. In this case the input and output are IRAF images and the operation is the absolute value. Note particularly the use of dynamic memory allocation and basic image I/O.

```
include <imhdr.h>

procedure filter ()

# Skeleton task to processs a 1-D image and use another file for output. This
# type of task is called a filter. The output file is similar to the input file,
# but with different values. This task will work with images of any dimensionality.
# There are two input parameters: the input file name and the output file name.

pointer sp          # Memory stack pointer
pointer if, ofn     # File name string pointers
pointer im, om      # Image descriptors
int     npts, nrow  # Number of pixels
int     line        # Line number
pointer il, ol      # Pixels

pointer immap(), imgl2r(), impl2r()     # Declare functions

begin
     # Initialize the dynamic memory stack
     call smark (sp)
     call salloc (ifn, SZ_LINE, TY_CHAR)
     call salloc (ofn, SZ_LINE, TY_CHAR)

     # Get the input image names
     call clgstr ("input", Memc[ifn], SZ_FNAME)
     call clgstr ("output", Memc[ofn], SZ_FNAME)

     # Open the images
     im = immap (Memc[ifn], READ_ONLY, 0)
     om = immap (Memc[ofn], NEW_COPY, im)

     # Find the image size (treat it as 2-D image)
     npts = IM_LEN(im,1)
     nrow = IM_LEN(im,2)

     # Do for each line in image
     do line = 1, nrow {
        il = imgl2r (im, line)         # Read data into dynamic memory
        ol = impl2r (om, line)         # Allocate output image line
        # Do something with data...can be SPP or Fortran subroutine.
        call fstuff (Memr[il], Memr[ol], npts)
     }
     call imunmap (im)                 # Close images
     call imunmap (om)
     call sfree (sp)                   # Free dynamic memory stack
end

procedure fstuff (input, output, npts)

# Dummy application routine for filter task--(find absolute value)
real   input[ARB], output[ARB]
int    npts
begin
     call aabsr (input, output, npts)  # Use VOPS absolute value procedure
end
```

Example B.4: Sample Filter.

Image I/O

The following is a complete example that demonstrates line by line image I/O by copying an existing image to a new image. Note that the procedure works the same regardless of the dimensionality and data type of the images. This is the code for the IRAF *imcopy* task in the *images* package which is in `images$imutil/imcopy.x`. There are comments scattered interspersed with the code to clarify it.

`IM_MAXDIM` and other constants used for image I/O are defined in `<imhdr.h>`. Other constants such as `ARB` and `SZ_FNAME` are defined in `iraf.h` which needs not be included explicitly.

```
include <imhdr.h>

# IMG_IMCOPY -- Copy an image. Use sequential routines to permit
# copying images of any dimension. perform pixel I/O in the
# datatype of the image, to avoid unnecessary type conversion.
procedure img_imcopy (image1, image2, verbose)
char    image1[ARB]           # Input image
char    image2[ARB]           # Output image
bool    verbose               # Print the operations
int     npix, junk
pointer buf1, buf2, im1, im2
pointer sp, imtemp, section
long    v1[IM_MAXDIM], v2[IM_MAXDIM]

# Declare function calls
int     imgnls(), imgnll(), imgnlr(), imgnld(), imgnlx()
int     impnls(), impnll(), impnlr(), impnld(), impnlx()
pointer immap()

begin
     call smark (sp)
     call salloc (imtemp, SZ_PATHNAME, TY_CHAR)
     call salloc (section, SZ_FNAME, TY_CHAR)

     # If verbose, print operation
     if (verbose) {
        call eprintf ("%s -> %s\n")
           call pargstr (image1)
           call pargstr (image2)
     }

     # Map the input image
     im1 = immap (image1, READ_ONLY, 0)

     # If output has section part, write only image section. Otherwise,
     # get temporary image & map as copy of existing image. Copy image
     # image to temporary and unmap images
     call imgsection (image2, Memc[section], SZ_FNAME)
```

(Continued...)

Example B.5: Image I/O.

`imgsection()` returns only the *image section* from an image file name. If `image2 = mosaic.imh[100:200,150:350]`, then the image section is `[100:200,150:350]` and we want to overwrite this space with the same space from the input image, i.e., pixels 100 to 200 inclusive in the first axis, and rows 150 to 350 in the second axis. If the output image already exists, the access mode is `READ_WRITE`. If it does not exist open it as a `NEW_COPY` of an existing image, passing the open image descriptor, `im1`, to `immap()`. All necessary header information will be copied.

The array `v1` keeps track of the current line to read from image1 by `imgnl()` and `v2` keeps track of the line written to `image2` using `impnl()`. `amovkl()` initializes the vectors with the long integer constant 1.

The macro defined constant `IM_LEN` contains the size of the image. It is defined in `<imhdr.h>`. It is a vector storing the size of each dimension up to the maximum number of dimensions supported by **imio** (seven). There is a case for each data type to preserve the precision of the pixels.

```
if (Memc[section] != EOS) {
   call strcpy (image2, Memc[imtemp], SZ_PATHNAME)
   im2 = immap (image2, READ_WRITE, 0)
} else {
   call xt_mkimtemp (image1, image2, Memc[imtemp], SZ_PATHNAME)
   im2 = immap (image2, NEW_COPY, im1)
}

# Setup start vector for sequential reads and writes
call amovkl (long(1), v1, IM_MAXDIM)
call amovkl (long(1), v2, IM_MAXDIM)
# Copy image
npix = IM_LEN(im1, 1)

switch (IM_PIXTYPE(im1)) {
case TY_SHORT:
    while (imgnls, (im1, buf1, v1) != EOF) {
        junk = impnls (im2, buf2, v2)
        call amovs (Mems[buf1], Mems[buf2], npix)
    }
case TY_USHORT, TY_INT, TY_LONG:
   while (imgnll (im1, buf1, v1) != EOF) {
      junk = impnll (im2, buf2, v2)
      call amovl (Meml[buf1], Meml[buf2], npix)
   }
```

(Continued...)

Example B.5 (Continued): Image I/O.

The pixel type unsigned short (`TY_USHORT`) will be copied to a buffer of type long. The routine `imgnll()` (the last letter denote the pixel type) returns a pointer in `buf1` that points to the beginning of the current line in the input image. The routine `impnll()` returns a pointer `buf2` that points to the beginning of the next line in the output image. `amovl()` copies `npix` pixel values from the input buffer to the output one. The input and output buffers in `Meml[]` have already been allocated in memory by `imgnll()` and `impnll()`. The loops will be repeated until all the lines have been copied, in which case an `EOF` is returned.

```
     case TY_REAL:
        while (imgnlr (im1, buf1, v1) != EOF) {
           junk = impnlr (im2, buf2, v2)
           call amovr (Memr[buf1], Memr[buf2], npix)
        }
     case TY_DOUBLE:
        while (imgnld (im1, buf1, v1) != EOF) {
           junk = impnld (im2, buf2, v2)
           call amovd (Memd[buf1], Memd[buf2], npix)
        }
     case TY_COMPLEX:
        while (imgnlx (im1, buf1, v1) != EOF) {
           junk = impnlx (im2, buf2, v2)
           call amovx (Memx[buf1], Memx[buf2], npix)
        }
     default:
        call error (1, "unknown pixel datatype")
     }
     # Unmap the images
     call imunmap (im2)
     call imunmap (im1)
     call xt_delimtemp (image2, Memc[imtemp])
end
```

Example B.5 (Continued): Image I/O.

Basic Graphics

Example B.7, below, demonstrate a very simple **gio** (IRAF graphics) application. It draws a box in graphics and writes a text string. It follows the conventions of most IRAF graphics applications. The graphics device is specified in the task parameter `device` and the graphics stream is `STDGRAPH`. Note that `gopen()` returns a pointer and this value is passed to all subsequent graphics procedures. In addition, the include file `<gset.h>` is specified. This contains `define`s for **gio** macros such as `G_TXSIZE`.

```
include <gset.h>

procedure hello ()

# HELLO -- Demonstrates simple GIO: Draws a box and a text string

pointer gp                      # Graphics descriptor
char    device[SZ_LINE]         # Device name string
pointer gopen()

begin
     # Get device name (nominally "stdgraph")
     call clgstr ("device", device, SZ_LINE)

     # Open graphics
     gp = gopen (device, NEW_FILE, STDGRAPH)

     # Set the viewport
     call gsview (gp, 0.2, 0.8, 0.2, 0.8)

     # Set the data window
     call gswind ("gp, 0.0, 1.0, 0.0, 1.0)

     # Draw a box around viewport
     call gamove (gp, 0.0, 1.0)
     call gadraw (gp, 1.0, 0.0)
     call gadraw (gp, 1.0, 1.0)
     call gadraw (gp, 0.0, 1.0)
     call gadrwa (gp, 0.0, 0.0)

     # Set graphics parameters: Center text horizontally
     call gseti (gp, G_TXHJUSTIFY, GT_CENTER)

     # Set size of text
     call gsetr (gp, G_TXTSIZE, 3.0)

     # Draw a text string
     call gtext (gp, 0.5, 0.5, "Hello World", EOS)

     # Close graphics
     call gclose (gp)
end
```

Example B.6: Basic Graphics.

Interactive Graphics

This example builds somewhat on the previous example. In addition to simply writing graphics, it uses the `clgcur()` procedure to return cursor coordinates to the application. Depending upon how the task is run, this is resolved in various ways. The usual situation is for the task to be run from the cl with the interactive graphics cursor activated. The user would then move the cursor and pressing a keyboard key would result in the coordinates of the cursor being returned to the task.

The `clgcur()` procedure is a **clio** function that returns a value that is `EOF` upon the end of cursor interaction. Note that the function call is within a `while` loop that terminates on the value `EOF`.

Note also that several cursor keys have been defined for the task. That is, when the user types that key with the graphics cursor active, the task performs some function. These functions are in addition to the built-in functions of the IRAF graphics cursor. The implementation of the cursor keys is also an example of the `switch` ... `case` syntax.

```
include <gset.h>

define UP               1
define DOWN             2
define LEFT             3
define RIGHT            4
define DEF_SIZE         0.15

procedure arrows ()

# ARROWS -- Demonstrates interactive capabilities of GIO
# Draw arrows in cardinal directions at coordinates of cursor.
# Optionally, specify size of arrow with a colon command.
# Cursor keys recognized:
#   d Down arrow
#   l Left arrow
#   q Quit
#   r Right arrow
#   u Up arrow
#   : Colon command
#   :s size  Change arrow size

pointer gp                  # Graphics descriptor
char    device[SZ_LINE]     # Device name string
real    wx, wy              # Cursor coordinates in WCS
int     wcs                 # Graphics wcs
int     key                 # Cursor key value
char    command[SZ_LINE]    # Cursor command string
char    cmdword[SZ_LINE]    # Command word
int     ip                  # Character in string
real    xs, ys, size        # Arrow size (in NDC)
string  coord "coord"       # Cursor parameter name
pointer gopen()
int     ctowrd(), ctor(), clgcur()

begin
     # Get graphics device from cl, nominally "stdgraph"
     call clgstr ("device", device, SZ_LINE)
     # Open graphics device
     gp = gopen (device, NEW_FILE, STDGRAPH)
     # Draw coordinate axes to orient ourselves
     call glabax (gp, EOS, EOS, EOS)
     # Set starting arrow size
     xs = DEF_SIZE
     ys = DEF_SIZE
```

(Continued...)

Example B.7: Interactive Graphics.

```
    while (clgcur (coord, wx, wy, wcs, key, command, SZ_LINE) != EOF) {
       # Cursor mode loop. interpret cursor commands until EOF.
       # Case statement switches on cursor key character value
       switch (key) {
       case 'd':
          call arrow (gp, DOWN, wx, wy, xs, ys)      # Down
       case 'l':
          call arrow (gp, LEFT, wx, wy, xs, ys)      # Left
       case 'r':
          call arrow (gp, RIGHT, wx, wy, xs, ys)     # Right
       case 'q':
          break                                      # Quit
       case 'u':
          call arrow (gp, UP, wx, wy, xs, ys)        # Up
       case ':':
          call printf (command)                      # Parse command
          ip = 1
          if (ctoword (command, ip, cmdwrd, SZ_LINE) ,+ 0)
             next     # No command on line
          # Case switches on 1st char of 1st word on command line
          switch (cmdwrd[1]) {
          case 's':                                 # Change arrow size
             if (ctor (bommand, ip, size) > 0) }
                call printf ("%f")
                call pargr (size)
                xs = size
                ys = size
             }
          }
       }
    }
    call gclose (gp)                 # Close graphics
end
```

(Continued...)

Example B.7 (Continued): Interactive Graphics

```
procedure arrow (gp, direc, x, y, xsize, ysize)

# ARROW --- Draw arrow as gio marker. 4 predefined markers: arrow pointing
# in each cardinal direction. Define mark as polyline to pass to gumark ().

pointer gp                      # Graphics descriptor
int     direc                   # Arrow direction (parameterized)
real    x, y                    # WCS of arrow center
real    xsize, ysize            # Arrow size in NDC
define  NPTS     5              # Number of points per polyline
define  NA       4              # Number of markers
define  NA       4              # Number of markers
real    px[NPTS,NA], py[NPTS,NA]                      # Arrow polylines
data    px /0.5, 0.5, 0.25, 0.5, 0.75,                # Up   (X)
            0.5, 0.5, 0.25, 0.5, 0.75,                # Down
            1.0, 0.0, 0.5,  0.0, 0.5,                 # Left
            0.0, 1.0, 0.5,  1.0, 0.5 /                # Right
data    py /0.0, 1.0, 0.5, 1.0, 0.5,                  # Up   (Y)
            1.0, 0.0, 0.5, 0.0, 0.5,                  # Down
            0.5, 0.5, 0.75, 0.5, 0.25,                # Left
            0.5, 0.5, 0.75, 0.5, 0.25/                # Right

begin
     call numark (gp, px[1,direc], py[1,direc], NPTS,
        x, y, xsize, ysize, NO)
end
```

Example B.7 (Continued): Interactive Graphics: the Arrow Procedure.

Task

The following code is a task statement that creates a task for the above procedures.

```
task    arrows,
        bones,
        filter,
        hello
```

To compile the code, use xc directly or use `mkpkg`, which also uses xc. If you extract the SPP code in the previous sections in files named `bones.x`, `filter.x`, `hello.x`, `arrows.x`, and `x_tutor.x`, respectively, the following command will compile and link them:

```
xc x_tutor.x bones.x filter.x hello.x arrows.x
```

producing `x_tutor.e` as the executable. You can either run this directly or define tasks in the cl:

```
task    arrows, bones, filter, hello = x_tutor.e
```

B.12.1 mkpkg

The following is a sample `mkpkg` file to make the package comprising the above examples. It creates a library (`tutor.a`) containing the procedures and links a single executable (physical task) containing several logical tasks.

```
$call  relink
$exit
update:
      $call  relink
      ;
relink:
      $update tutor.a
      $call   linktutor
      ;
linktutor:
      $omake x_tutor.x
      $link  x_tutor.o tutor.a -o xx_tutor.e
      ;
tutor.a:
```

Example B.8: Sample mkpkg File.

APPENDIX C:

Tips and Pitfalls

This reference documents the major features of the SPP language. However, it is necessarily incomplete. For the most complete and up-to-date details of any specific library package or procedure, consult the on-line source and documentation. There is high-level documentation in the IRAF `doc$` directory. The source for each library package described here, **imio**, **clio**, etc., resides in a separate directory in the IRAF hierarchy, having the name of the package. In addition, a cl environment variable is defined for each library package. Thus, the source for imio is in the directory `imio$`. There is a directory containing documntation describing the packages in a `doc` subdirectory of each library package and the source also contains documentation.

Procedure Arguments

If a procedure has formal parameters, they should agree in both number and type in the procedure declaration and when the procedure is called. In particular, beware of `short` or `char` parameters in argument lists. An `int` may be passed as a parameter to a procedure expecting a `SHORT` integer on some machines, but this usage is *not portable*, and is not detected by the compiler. The compiler does not verify that a procedure is declared and used consistently. Do not use type coercion in procedure actual arguments. Such as:

```
call foobar (..., short (intvar), ...)
```

In some cases, the coercion is not performed in passing the argument to the procedure. A particular problem is using a literal (quoted) character in the calling sequenct to a procedure expecting a `char` such as `stridx()`. Such a literal is converted into an integer constant. On some systems, it won't matter if the called procedure expects a long or short integer, but on some, it will result in the wrong value passed.

Calling Fortran

Since SPP is preprocessed into Fortran, in most cases, it is quite straightforward to call an existing Fortran subroutine from an SPP procedure. The most important caution is the case of character strings. SPP strings are not the same as Fortran strings. SPP strings are implemented as arrays of integers. However, there are procedures available to transform between the two: `f77pak()` converts an SPP string to a Fortran string, and `f77upk()` converts a Fortran string to an SPP string. Note that you must declare the Fortran string in the SPP procedure with a Fortan statement. This is possible with the `%` escape character as the first character on a line. This indicates to the xc compiler that the following statement should not be processed but copied directly to the Fortran code. See Example C.9, below.

```
.
.
# Declare the Fortran string
%character*8   fstr

# Declare the SPP string
char           sstr[8]
.
.
     # Convert the SPP string to a Fortran string
     call f77pak (sstr, fstr, 8)

     # Call the Fortran subroutine
     call forsub (fstr, ...)
.
.
```

Example C.9: Declaring a Fortran String in SPP.

Character Strings

SPP strings are not scalar variables. Their value cannot be changed by an assignment statement. Strings are, in fact, arrays of short integers, with the additional complication of an extra element at the end for the `EOS` character. It is possible to declare strings with dynamic memory allocation. In fact, is a common practice to use stack memory for temporary string storage.

```
pointer sp
pointer infile, outfile
pointer errmsg
.
.
begin
     # Mark the memory stack
     call smark (sp)
     # Allocate memory for the strings
     call salloc (infile, SZ_FNAME, TY_CHAR)
     call salloc (outfile, SZ_FNAME, TY_CHAR)
     .
     .
     # Get strings from the cl
     call clgstr ("infile", Memc[infile], SZ_FNAME)
     call clgstr ("outfile", Memc[outfile], SZ_FNAME)
     .
     .
     # Free the memory stack
     call sfree (sp)
end
```

Example C.10: Stacking Memory for Temporary String Storage.

Arrays of Strings

It is possible to declare an array of strings, but remember that each string element needs its own `EOS` character. Typically, the strings would be allocated dynamically and referenced in a called procedure, as shown in Example C.11.

```
define NUM_STR 16     # Array size
define STR_SIZ 79     # String size
                      # (note odd size to allow for EOS)

pointer strarr        # Pointer for array of strings
int     arrsiz

begin
     arrsiz = (STR_SIZE + 1) * NUM_STR
     # Allocate string array
     call malloc (strarr, arrsize, TY_CHAR)
     .
     .
     call myproc (Memc[strarr], STR_SIZ, NUM_STR)
end

procedure myproc (strarr, strsize, numstr)
char    strarr[strsiz,numstr]  # Array of strings
int     strsiz                 # String size
int     numstr                 # Number of strings

begin
     .
     .
end
```

Example C.11: Referencing Dynamically Allocated Strings.

The important points to keep in mind are that strings implemented as arrays of `char`s (`short`s), even though they are declared a fixed size, they may not use the entire declared space. A special character value (`EOS`, implemented as ASCII `NUL`) is used as the string terminator. Most procedures that require strings also take an argument specifying the string length. This does not mean that the entire declared string will be used, only the maximum possible string size. There are a few important exceptions.

Characters vs. Strings

Note the distinction between single and double quoted characters. Single quotes indicate the ASCII value of a single character and are treated as an `int` scalar in processed SPP. Double quoted strings are literal strings and may only be specified as actual procedure arguments or the object of a `string` declaration. Using single quoted characters in place of a `char` array can cause unexpected problem, for example in:

```
stridx ('x', string)
```

`'x'` is an `int`, while `stridx()` expects a `char`. Other routines with this problem include `ungetc()` and `putc()`. Note that the cast operator `char ('x')` does not work! It translates into `int(120)`. You should use something like:

```
char    x_char
        x_char = 'x'
        i = stridx (x_char, string)
```

Formatted I/O

Newlines are significant. Lines of output, to `STDOUT` for example, is separated by newlines, a carriage return and a line feed. The `printf()` procedure does not automatically issue a newline with every call. You must explicitly write the newlines using the `\n` escape as part of the format string. Otherwise, your output will be strung together, rather unintelligably. Actually, this can be useful, as you can use multiple `printf()` calls to build a single line of output. On input, a text file consists of lines delimited by newlines. The file may be read line by line using `getline()`. The newline terminating each line is returned as part of the string. Note that `getline()` and `putline()` are two of the procedures dealing with strings that do not have a string length argument. It is assumed that the string buffer is allocated with the size `SZ_LINE`.

The % Character

To output a percent character (%) using any of the formatted output procedures, use two adjacent percent characters, %% in the format string.

```
call printf ("Ratio: %f%%\n")
    call pargr (ratio)
```

Results in:

```
Ratio: 12.34%
```

(assuming the value of `ratio` is 12.34).

Buffered Output

Standard formatted output is normally buffered. The result is that output to `STDOUT` may not appear on the user's terminal right away. The buffer is flushed when it is full, at the end of the task, or when it is explicitly flushed. The buffer may be flushed with `flush()`, whose argument is the file descriptor of the stream, `STDOUT` for example. In some cases, particularly in deing stages of development, it may be desirable to have output appear more quickly. Rather than using `flush()` repeatedly, you may set the fio parameter `F_FLUSHNL` to `YES` with a call to `fset()`. This advises **fio** to flush the buffer whenever it prints a newline character. Thus, output will appear on every line. Output to `STDERR` always flushes on newlines.

Dynamic Memory Allocation

In order to use dynamic memory pointers properly, you must declare at least one `pointer` variable in the appropriate procedures. This will generate the code defining a common block with declarations for all of the `Mem` arrays: `Memd`, `Memr`, `Memi`, `Mems`, etc. Otherwise, you will get a compiler error complaining of undeclared variables.

Image I/O

Perhaps the most confusing aspect of image I/O is the rather unintuitive way images are written in **imio**. It is necessary to obtain an output pointer using one of the `imp`... procedures and then filling in the values in the output buffer. The pixels are not actually written to the output file until the output buffer is flushed or the image is closed. This can, in fact, lead to another pitfall. If you wish to write and read the same image in the same task, you must be sure that the pixels are written out before trying to read them in again. This may be assured with a call to `imflush()` after filling the output buffer. Alternately, you might close the image using `imunmap()` and then reopen it with `immap()`. A brief example may clarify this situation. The following fragment of code opens an image for read and write access, writes some pixels and reads them back in.

```
pointer im, ip
int     nx

# Map the image
im = immap (image, READ_WRITE, 0)

# Get the line size
nx = IM_LEN(im, 1)

# Map an output buffer
ip = impl2r (im, 1)

# Fill the output buffer
call amovkr (1.0, Memr[ip], nx)

# Flush the output
call imflush (im)

# Read the line back in
ip = imgl2r (im, 1)
.
.
```

Example C.12: Image I/O.

If you read two lines using arbitrary line I/O with two separate buffer pointers, the second call may make the first pointer `x1` invalid.

```
x1 = imgl2r (im, i)
x2 = imgl2r (im, i+1)
```

This applies to output, `impl2`*T*`()` as well as input.

Group Format

One additional wrinkle involves multi-image *group format* STF (STSDAS[1] format) images. This format allows more than one image in a single logical image (pair of files; header and pixel file) with a common image header. It is possible to access more than one image in the group simultaneously in a task. With imio, each sub-image (sometimes referred to confusingly as a *group*) you need to use `immap()` separately. To specify which image in the set to open, append the image number enclosed in square brackets to the file name in the `immap()` call. The following opens the second image in a multi-image group format file:

1. For more information about STSDAS, see the *STSDAS Users Guide*, available from the STSDAS Group at STScI.

```
# Get the image name from the cl
call clgstr ("image", image, SZ_FNAME)
# Append the "group" number
call strcat ("[2]", image, SZ_FNAME)
# Open the image
g1 = immap (image, READ_ONLY, 0)
```

Example C.13: Opening the Second Group of a Group Format STSDAS Image.

In many cases, it would be up to the user to specify the group number on the image file name when using the task. There may be cases, however, in which a task would use specific groups in an image. To create a new multi--image file, you must specify the total number of images in the set as well as the image number. Example C.14 creates a four image set and opens the first image.

```
# Get the image name from the cl
call clgstr ("image", image, SZ_FNAME)
# Append the "group" number and number of images
call strcat ("[1/4]", image, SZ_FNAME)
# Open the image
g1 = immap (image, READ_ONLY, 0)
```

Example C.14: Creating a Four-Image Set and Opening the First Image.

A slight complication arises when you wish to create a multi-image group format file and simultaneously access more than one image. In this case, you must create the image, close it, and reopen the individual images. Note also that the pixel file will *not* be created propeunless a write operation is performed. This may be done by simply writing a single line before closing the image.

```
# Get the image name from the cl
call clgstr ("image', image, SZ_FNAME)
call strcpy (image, img1, SZ_FNAME)
# Append the "group" number and number of images
call strcat ("[1/4]", img1, SZ_FNAME)
# Open the new image
g1 = immap (image, NEW_IMAGE, 0)
IM_NDIM(im)  = 2
IM_LEN(im,1) = 512
IM_LEN(im,2) = 512
# Write a dummy line to create the pixel file
junk = impl2r (image, 1)
# Close the image
call imunmap (g1)
# Reopen the individual images
call strcpy (image, img1, SZ_FNAME)
call strcat ("[1]", img1, SZ_FNAME)
g1 = immap (img1, READ_WRITE, 0)
.
.
call strcpy (image, img4, SZ_FNAME)
call strcat ("[4]", img4, SZ_FNAME)
g4 = immap (img4, READ_WRITE, 0)
```

Example C.15: Accessing More Than One Image in a Multi-Image File.

Logical Flags

In addition to `bool` data type variables, many SPP programs use the macro predefined constants `YES` and `NO` as flag or switch values. *Note that these are* `int` *constants, not* `bool`*s.* The `bool` literal constants are `true` and `false`

APPENDIX D:
Debugging

The SPP preprocessor, xc, recognizes many syntax errors. Needless to say, not all programming errors will be caught this way. Since SPP is preprocessed into Fortran, it is useful to know a bit about the resulting Fortran code in order to find programming errors. The most instructive way to understand the code is to look at it. Use the `-f` option of xc to preserve the Fortran output. Many times errors are apparent in the Fortran code without having to use a source-level debugger at all.

Identifier Mapping

Since the Fortran produced by xc is Fortran 66, identifier names must be six characters or fewer, with no special characters such as underscores. SPP however, permits longer identifier names with the underscore character. The xc preprocessor maps such names by first removing underscores and using up to the first five characters of the identifier and the last character. The xc preprocessor writes a table of the original SPP identifiers and the mapped Fortran names at the end of the output Fortran as comments. If different SPP identifiers map to the same Fortran identifier, xc issues a warning that the identifier mapping is not unique and creates a unique identifier by replacing the last character with a digit in one case.

Dynamic Memory

It is possible to examine the values of dynamically allocated memory. These are treated as a Fortran `common` block, with all of the `Mem` arrays `equivalenced` to a single array. The relevant Fortran code generated is shown in Example D.1.

```
logical Memb(1)
integer*2 Memc(1)
integer*2 Mems(1)
integer Memi(1)
integer*4 Meml(1)
real Memr(1)
double precision Memd(1)
complex Memx(1)
equivalence (Memb, Memc, Mems, Memi, Meml, Memr, Memd, Memx)
common /Mem/ Memd
```

Example D.1: Fortran Code for Handling Dynamically Allocated Memory.

VMS

The VAX/VMS debugger permits examining the `Mem` arrays. Keep in mind the manner in which the array was allocated, however. The pointer is an arbitrary offset into virtual memory. The elements of your array are located relative to the pointer. The debugger will not know the size of the array, but you can specify a range of elements to examine. Once the pointer is *dereferenced* by passing to a procedure, it is treated as a normal Fortran array. However, be particularly careful of arrays declared in procedures with ARB. ARB is a macro that translates into a very large number. If you examine an array declared ARB without specifying a range of elements, the debugger will try and list what it thinks are all of the elements of the array. *Remember to specify a range of array elements.*

Unix

In the Unix `dbx` debugger, it is a bit more tedious to examine the contents of a dynamically allocated arrays. You need to specify the memory location (pointer address) and the data type to display. For example, if a pointer to `Memr` is in a variable called `line`, then the following `dbx` command will display the first element:

```
print (line-1)*4/f
```

To look at the n^{th} element, add *n* to the word location:

```
print ((line-1)*4+10)/f
```

will show the 10^{th} element. The following `dbx` initialization file defines command aliases to help examine contents of the `Mem` buffers. It may be placed in the file `.dbxinit` in the Unix root directory.

```
alias memd "((!:1)-1)*8/g"
alias memi "((!:1)-1)*4/D"
alias mems "((!:1)-1)*2/d"
alias memr "((!:1)-1)*4/f"
alias memc "((!:1)-1)*2/!:2 c"
alias veci "&!:1[!:2]/!:3 D"
alias vecc "&!:1[!:2]/!:3 c"
```

Example D.2: Unix `.dbxinit` Debugging File.

The commands are used by specifying the symbol name of the memory pointer. For example if the SPP code contained:

```
call malloc (buf, npix, TY_REAL)
call myproc (Memr[buf], npix)
```

Then you could examine the first element of the memory buffer pointed to by `buf` with the `dbx` command:

```
memr buf
```

Task

The single line SPP `task` statement results in a very large amount of Fortran code. This implements a single procedure called `sys_runtask`, which is mapped to the Fortran name `SYSRUK`. This is because there is a great deal of processing dealing with selecting tasks and handling errors. Normally, there is no need to look at the preprocessed code for the task. When your task is compiling, you will see this procedure being compiled. Be aware also that when you are debugging, your top-level applications procedure is a *subroutine* of the task, which is, in turn, a subroutine of the IRAF main procedure. The top level IRAF main is part of the IRAF kernel and therefore written in C. Most debuggers will somehow make it known that they are trying to debug C code. This is usually not important.

APPENDIX E:
STSDAS Tables

STSDAS tables[1] are binary files that contain data in row and column format. Each column has a name, data type, print format, and unit. All the values in a given column are of the same data type, but different columns may have different data types. The column name should be unique within a table. The print format may be used to display the values but does not affect the way the values are stored in the table. The `units` string may contain any information that will fit; calling it "units" is just a suggestion. A table may also contain header parameters in a format similar to FITS header keywords.

The data types supported for tables are double precision real, single precision real, integer, boolean, and text strings. Values are stored in the table file in the host machine's binary format. Elements that have not been assigned values or that have been set to "undefined" are flagged as such in the table.

The object library specified to xc as `-ltbtables` contains all the spp-callable table I/O routines. The include file `tbset.h` defines parameters for getting such information as the number of rows or columns in a table. Some items may also be set. The maximum lengths of column names and similar values are also specified in that file. Further details are given below. The `tbpset` routine is used to set parameter values, and the integer function `tbpsta` returns values.

A table with more than one column is a 2-D array of values. A 2-D array can be stored in the file in row or column ordered format. That is, as you step from word to word in the file, you could be stepping along a row or down a column. Both options are supported for STSDAS tables. Simple

1. The STSDAS system, including the tables package and libraries for table manipulation and multigroup access, is available via anonymous ftp to `stsci.edu`. If you need more information, contact the STSDAS Group via e-mail to: `hotseat@stsci.edu`

text files in row and column format can also be accessed as tables by the STSDAS table I/O routines.

The file name for a binary table must include an extension, with `tab` as the default. A text table, on the other hand, need not have an extension. `STDIN` and `STDOUT` may be used for input and output text tables.

The table interface includes routines for accessing table files, columns, header parameters, table parameters, and table data. The name of each routine begins with "tb", the next letter indicates what type of object is involved (row, column, parameter, etc.), and the last three letters specify what is to be done (e.g., open, close, get, put). For example, `tbtopn` opens a table. The third letter ("t") implies that the routine applies to a table as a whole, and "opn" means "open". Similarly, `tbtclo` closes a table. For some routines the last letter indicates the data type of the input or output buffer. For example, `tbegtr` operates on a table element ("e") to get ("gt") an element, and the output buffer is of type real ("r"). The corresponding "put" routine is `tbeptr`. Table E.1 is a list of third letters and what they refer to:

Letter	*Object*	*Examples of use*
`t`	Table file	Open, close, get table name
`p`	Table parameter	Number of rows, number of columns
`h`	Header parameter	Get or put header parameter
`c`	Column	Find, create, get or put column
`r`	Row	Get or put values in a row
`e`	Element	Get or put a single value

Table E.1: Table I/O Procedure Naming Conventions.

Procedure	*Description*
`tp = tbtopn (tablename, iomode, template)`	Initialize (and open the table if not `NEW_FILE` or `NEW_COPY`)
`tbtcre (tp)`	Create new table (after initializing with `tbtopn`)
`tbtclo (tp)`	Close a table

Table E.2: Procedures to Open and Close Tables.

Example E.1 reads all values from one table column and prints the values that are defined. If this were in a file called `test.x`, it could be compiled and linked by typing[2]:

```
xc -p stsdas test.x -ltbtables.
```

```
task    test
include <tbset.h>               # defines TBL_NROWS, SZ_COLNAME, etc
procedure test()
pointer tp                      # pointer to table descriptor
pointer cp                      # pointer to column descriptor
char    intable[SZ_FNAME]       # table name
char    colname[SZ_COLNAME]     # column name
real    value                   # a single value from a table element
int     nrows                   # number of rows in table
int     row                     # loop index for row number
pointer tbtopn()
int     tbpsta()
begin
        call clgstr ("intable", intable, SZ_FNAME)
        call clgstr ("colname", colname, SZ_FNAME)
        tp = tbtopn (intable, READ_ONLY, NULL)  # open the table
        call tbcfnd (tp, colname, cp, 1)        # find the column in the table
        if (cp == NULL) {
            call tbtclo (tp)
            call error (1, "column not found")
        }
        nrows = tbpsta (tp, TBL_NROWS)
        do row = 1, nrows {
            call tbegtr (tp, cp, row, value)    # get value in current row
            if (!IS_INDEF(value)) {             # is the value defined?
                call printf ("%14.6g\n")
                    call pargr (value)
            }
        }
        call tbtclo (tp)                        # close the table
end
```

Example E.1: Table I/O Example.

2. Notice that the `-p stsdas` flag means that you need to have the STSDAS external package available on your system.

Procedure	*Description*
`tbcdef (tp, colptr, colname, colunits, colfmt, datatype, lendata, numcols)`	Define columns
`tbcfnd (tp, colname, colptr, numcols)`	Find a column from its name
`tbcinf (colptr, colnum, colname, colunits, colfmt, datatype, lendata, lenfmt)`	Get information about a column
`int = tbcigi (colptr, param)`	Get specific info about a numeric column (e.g. name or data type)
`tbcigt (colptr, param, outstr, maxch)`	Get specific info about a string column (e.g. name or data type)

Table E.3: Procedures Dealing with Columns.

Procedure	*Description*
`tbtcpy (inname, outname)`	Copy a table
`tbtdel (tablename)`	Delete a table
`tbtren (oldname, newname)`	Rename a table
`int = tbtacc (tablename)`	Test for the existence of a table
`tbtext (inname, outname, maxch)`	Append default extension (if it's not already there)
`tbtnam (tp, tblname, maxch)`	Get the name (including extension) of the table
`tbtflu (tp)`	Flush FIO buffer for table

Table E.4: Table File Operations.

Reading and Writing Data

Three sets of get and put routines are provided for accessing table data. The "tbe..." routines get or put single elements; that is, values at a specified row and column. The "tbr..." routines get or put one or more elements in a single row. The "tbc..." routines get or put values in a single column over a range of rows. The last (sixth) letter of each routine name specifies the buffer data type: "t" for a text string, "b" for boolean, "i" for integer, "r" for real, and "d" for double precision. The data type of the buffer does not need to be the same as the data type of the table column; the table I/O routines convert data type when the column and buffer do not match.

The `tbrgt`*T* and `tbcgt`*T* routines return a boolean array that indicates whether the table elements gotten are undefined. A true value means the table element *is* undefined. The `tbegt`*T* routine returns the data type-specific `INDEF` value when the table element is undefined. When writing values into a table, values may be set to undefined by calling `tbrudf`. If a row exists, but no value has ever been written to a particular column in that row, the element at that row and column will automatically be undefined; that is, it is not necessary to call `tbrudf`. A row exists if a value has been put into any column in that row or into a subsequent row (larger row number).

Procedure	***Data Types***	***Description***
`tbegt`*T* `(tp, colptr, rownum, buffer)`	`b i r d`	Get a numeric value from the table
`tbegtt (tp, colptr, rownum, buffer, maxch)`		Get a string value from the table
`tbept`*T* `(tp, colptr, rownum, buffer)`	`t b i r d`	Put a value into the table
`tbrgt`*T* `(tp, colptr, buffer, nullflag, numcols, rownum)`	`b i r d`	Get numeric values from a row
`tbrgtt (tp, colptr, buffer, nullflag, lenstr, numcols, rownum)`		Get string values from a row
`tbrpt`*T* `(tp, colptr, buffer, numcols, rownum)`	`b i r d`	Put numeric values into a row
`tbrptt (tp, colptr, buffer, lenstr, numcols, rownum)`		Put string values into a row
`tbcgt`*T* `(tp, colptr, buffer, nullflag, firstrow, lastrow)`	`b i r d`	Get numeric values from a column
`tbcgtt (tp, colptr, buffer, nullflag, lenstr, firstrow, lastrow)`		Get string values from a column
`tbcpt`*T* `(tp, colptr, buffer, firstrow, lastrow)`	`b i r d`	Put numeric values into a column
`tbcptt (tp, colptr, buffer, lenstr, firstrow, lastrow)`		Put string values into a column
`tbrudf (tp, colptr, numcols, rownum)`		Set values in a row to undefined

Table E.5: Table Get and Put Procedures.

Example E.2 gets two values from each row of a table and copies them to another table if neither value is undefined. A double-precision buffer is used so that data of any numerical type will be copied without loss of precision.

```
include <tbset.h>
define  NCOLS           2       # number of columns to get
procedure test()
pointer sp                      # stack pointer
pointer intable, outtable       # scratch for table names
pointer ira, idec               # scratch for arrays of input values
pointer ora, odec               # scratch for arrays of output values
pointer ra_flag                 # scratch for array of null flags
pointer dec_flag                # scratch for array of null flags
char    cname[SZ_COLNAME,NCOLS] # column names
pointer itp, otp                # pointers to table descriptors
pointer icp[NCOLS]              # pointers to column descriptors in input
pointer ocp[NCOLS]              # pointers to column descriptors in output
int     inrows, onrows          # number of rows in input, output tables
int     irow                    # loop index for row number in input table
int     orow                    # row number in output table
int     i                       # loop index
bool    nullflag[NCOLS]         # null flags for getting info from a row
bool    bad                     # true if any element of nullflag is true
double  value[NCOLS]            # values gotten from a table
pointer tbtopn()
int     tbpsta()
begin
        # Allocate scratch space for table names.  We'll allocate space
        # for column values later, after we know the size of the table.
        call smark (sp)
        call salloc (intable, SZ_FNAME, TY_CHAR)
        call salloc (outtable, SZ_FNAME, TY_CHAR)

        # Get table names.
        call clgstr ("intable", Memc[intable], SZ_FNAME)
        call clgstr ("outtable", Memc[outtable], SZ_FNAME)

        # Get column names.
        call clgstr ("ra_col", cname[1,1], SZ_COLNAME)
        call clgstr ("dec_col", cname[1,2], SZ_COLNAME)

        # Open input table.
        itp = tbtopn (Memc[intable], READ_ONLY, NULL)

        # Find columns in input table.  Check if they were found.
        call tbcfnd (itp, cname, icp, NCOLS)
        if (icp[1] == NULL || icp[2] == NULL) {
            call tbtclo (itp)
            call error (1, "column not found")
        }
```

(Continued...)

Example E.2: Copying Columns.

```
    # Create an output table with the same columns as the input table.
    otp = tbtopn (Memc[outtable], NEW_COPY, itp)
    call tbtcre (otp)

    # Copy header parameters from input to output.
    call tbhcal (itp, otp)

    # Find columns in output table.  They will be there since they were
    # in the input table.
    call tbcfnd (otp, cname, ocp, NCOLS)

    # There will be fewer rows in the output table if the columns
    # we’re interested in contain undefined elements.
    inrows = tbpsta (itp, TBL_NROWS)

    # Here are three different ways of copying the values.
    # 1.  Copy element by element.
    orow = 0
    do irow = 1, inrows {
        call tbegtd (itp, icp[1], irow, value[1])
        call tbegtd (itp, icp[2], irow, value[2])
        if (!IS_INDEFD(value[1]) && !IS_INDEFD(value[2])) {
            orow = orow + 1
            call tbeptd (otp, ocp[1], orow, value[1])
            call tbeptd (otp, ocp[2], orow, value[2])
        }
    }

    # 2.  Use the get-row and put-row routines.  This will copy
    # any number of columns, one row at a time.
    orow = 0
    do irow = 1, inrows {
        call tbrgtd (itp, icp, value, nullflag, NCOLS, irow)
        bad = false
        do i = 1, NCOLS
            if (nullflag[i])
                bad = true
        if (!bad) {
            orow = orow + 1
            call tbrptd (otp, ocp, value, NCOLS, orow)
        }
    }
```

(Continued...)

Example 5.2 (Continued): Copying Columns.

```
        # 3.  Use the get-column and put-column routines.
        call salloc (ira, inrows, TY_DOUBLE)
        call salloc (idec, inrows, TY_DOUBLE)
        call salloc (ra_flag, inrows, TY_BOOL)
        call salloc (dec_flag, inrows, TY_BOOL)
        call salloc (ora, inrows, TY_DOUBLE)    # possibly more than we need
        call salloc (odec, inrows, TY_DOUBLE)
        call tbcgtd (itp, icp[1], Memd[ira], Memb[ra_flag], 1, inrows)
        call tbcgtd (itp, icp[2], Memd[idec], Memb[dec_flag], 1, inrows)

        # Note that irow and orow are zero indexed in this loop.
        orow = -1
        do irow = 0, inrows-1 {
            if (!Memb[ra_flag+irow] && !Memb[dec_flag+irow]) {
                orow = orow + 1
                Memd[ora+orow] = Memd[ira+irow]
                Memd[odec+orow] = Memd[idec+irow]
            }
        }
        onrows = orow + 1               # number of rows in output table
        if (orow > 0) {
            call tbcptd (otp, ocp[1], Memd[ora], 1, onrows)
            call tbcptd (otp, ocp[2], Memd[odec], 1, onrows)
        }

        # Done.  Three times, even.
        call tbtclo (itp)
        call tbtclo (otp)
        call sfree (sp)
end
```

Example 5.2 (Continued): Copying Columns.

Header Parameters

Tables may contain header parameters consisting of a keyword name, data type flag, and a value. These are stored in the table as text strings. These parameters are *not* used for information such as the number of rows or columns, and the table I/O routines do not use header parameters when getting or putting table elements. The same data types are supported for header parameters as for table data, and type conversion is performed, except that a value stored as a text string may only be gotten as text, not as numeric or boolean. The distinction between *adding* and *putting* values is the same as for image header keywords. You can call `tbhpt`*T* to put a header parameter only if that parameter already exists in the table, but you can call `tbhad`*T* to either add a new header parameter or replace an existing one. In contrast to the `imio` interface, when you open a table `NEW_COPY`, the header parameters are not copied.

.

Procedure	*Data Types*	*Description*
`value = tbhgtT (tp, param)`	b d i r	Get a numeric header parameter
`tbhgtt (tp, param, text, maxch)`		Get a string header parameter
`tbhadT (tp, param, value)`	t b d i r	Add a new header parameter or replace existing one
`tbhptT (tp, param, value)`	t b i r d	Replace an existing header parameter
`tbhcal (itp, otp)`		Copy all header parameters
`tbhgnp (tp, parnum, keyword, dtype, str)`		Get Nth header parameter as a string

Table E.6: Header Parameter Procedures.

The `tbset.h` Include File

This section describes the include file `tbset.h`. In most situations the only parameters that will be needed are `SZ_COLNAME` and `TBL_NROWS`.

These three are used for declaring the sizes of `char` variables for column names, units, and print formats.

- `SZ_COLNAME` - Maximum length of a column name
- `SZ_COLUNITS` - Maximum length of string for units
- `SZ_COLFMT` - Maximum length for print format

The next four parameters may be read by `tbpsta` but may not be set:

Parameter	***Meaning***
TBL_NROWS	Number of rows written to
TBL_NCOLS	Number of columns defined
TBL_ROWLEN_USED	Amount of row length used (unit = size of single precision)
TBL_NPAR	Number of user parameters

Table E.7: Non-settable Parameters Read by `tbpsta`.

These may be set by `tbpset` or read by `tbpsta`. Parameters `TBL_ROWLEN` and `TBL_INCR_ROWLEN` are relevant only to row-ordered tables, while `TBL_ALLROWS` and `TBL_INCR_ALLROWS` are relevant only to column-ordered tables. `TBL_ROWLEN` is for setting the row length to a specific value. In contrast, `TBL_INCR_ROWLEN` is used to increase the row length by the specified amount over its current value, whatever that may be. The latter is more useful. When creating a new table, we suggest the following procedure for a row-ordered table. After calling `tbtopn`, define columns using `tbcdef`. Then the row length will be sufficient for the columns that have been defined. If you will need to define more columns after the table has been created, you can call `tbpset` with `TBL_INCL_ROWLEN` to preallocate the needed space before creating the table with `tbtcre`. The numerical value would be one for each single-precision or integer column, and two for each double-precision column. For character strings, divide the maximum string length by the number of bytes in a single-precision variable and round up.

Parameter	*Meaning*
TBL_ROWLEN	Row length to allocate (units are the size of a single-precision)
TBL_INCR_ROWLEN	Increase row length (in single-precision units)
TBL_ALLROWS	Number of rows to allocate
TBL_INCR_ALLROWS	Increase number of allocated rows
TBL_WHTYPE	Type of table? (see below)
TBL_MAXPAR	Maximum number of user parameters
TBL_MAXCOLS	Maximum number of columns

Table E.8: Table Parameters That Can be Read or Set.

The table type as set or read using TBL_WHTYPE is defined byt the parameters in Table E.9.

Parameter	*Meaning*
TBL_TYPE_S_ROW	Row-ordered binary table
TBL_TYPE_S_COL	Column-ordered binary table
TBL_TYPE_TEXT	Text file

Table E.9: Table Types.

The parameters described in Table E.10 have to do with the file size and file I/O buffer size.

Parameter	*Meaning*
TBL_ADVICE	Set RANDOM or SEQUENTIAL
TBL_BUFSIZE	Get buffer size in characters
TBL_DATA_SIZE	Get size of table data in characters

Table E.10: Table Size and File I/O Buffer Size.

The parameters are for getting information about a column using `tbcigt` or `tbcigi`.

Parameter	*Meaning*
`TBL_COL_NAME`	Column name
`TBL_COL_UNITS`	Units for column
`TBL_COL_FMT`	Print format for displaying values
`TBL_COL_DATATYPE`	Data type (`-n` for character string)
`TBL_COL_NUMBER`	Column number
`TBL_COL_FMTLEN`	Length for printing using print format
`TBL_COL_LENDATA`	Number of elements if colum is an array

Table E.11: Getting Column Information.

Procedure	*Description*
`tbpset (tp, setwhat, value)`	Set a table parameter
`int = tbpsta (tp, param)`	Get the value of a table parameter (e.g. number of rows)
`int = tbcigi (colptr, param)`	Get information about column (integer)
`tbcigt (colptr, param, outstr maxch)`	Get information about column (string)

Table E.12: Table Parameter Procedures.

Print Formats

The print format is used by such tasks as **tprint**, **tedit**, and **tread** to determine how the column values are to be displayed. The earlier statement that the print format does not affect the way the values are stored in the table is really only true for binary tables. For output (or read-write) text tables the print format is actually used to write the file, so it is critical with regard to the precision of the data values. Most of the ordinary Fortran formats are supported for tables. SPP formats are discussed in the `fmtio` section of this document. The only SPP print formats that are not allowed are those that are simply irrelevant, such as `t`, `w`, and `z`. The field width may not be zero, however. The procedure `tbbftp` may be used to convert a user-supplied Fortran style format to an SPP style format.

Table E.13 is a list of the default print format for each data type, given in both SPP style and Fortran style.

Data type	***SPP***	***Fortran***
`real`	`%15.7g`	`G15.7`
`double prec`	`%25.16g`	`G25.16`
`integer`	`%11d`	`I11`
`boolean`	`%6b`	`L6`
`text string`	`%-ns`	`A-n`

Table E.13: Default Print Formats.

For character strings “n” is the string size as given when the column was defined. The minus sign means that the string will be left justified. While a format such as “A-12” is not available in standard Fortran, the `tbbftp` routine will convert it to “%-12s”.

SPP formats and Fortran equivalents that are supported for tables are listed in this table. The syntax is `%w.dC` (SPP style) or `Cw.d` (Fortran style), where `w` is the field width, `d` is the number of decimal places (or precision for `g` format), and `C` is the format code as given in the left column below. When giving a format in Fortran style, use the format code given in the second column; these are shown in upper case but may also be given in lower case. Note that `H` and `M` are not standard Fortran formats; in particular, `H` is not interpreted as Hollerith.

SPP	*Fortran*	*Meaning*
`b`	`L`	Boolean “yes” or “no”
`d`	`I`	Integer, displayed in decimal
`x`	`Z`	Integer, displayed in hexadecimal
`e`	`E or D`	Exponential format
`f`	`F`	Floating point
`g`	`G`	Use F or E as appropriate
`h`	`H`	HH:MM:SS.d (sexagesimal)
`m`	`M`	HH:MM.d (sexagesimal)
`s`	`A`	Character string

Table E.14: Table Print Formats.

Table Utilities

Table E.15 lists some table utility procedures. These permit operating on entire columns or rows and performing other funtions on the table as a whole.

Note also that the **tbtables** package of tasks in the STSDAS external package that allows flexible and sophisticated manipulation of existing tables without writing any code. These include such database-related functions as extracting selected rows based on the value of particular fields, extracting given columns by name, printing a report from a table or editing a table in-place. See `help tbtables` for a list of the tasks and a brief description of each.

Procedure	***Description***
`tbtchs (tp, maxpar, maxcols, rowlen, allrows)`	Change allocated space of any/all portions of a table
`tbrcpy (itp, otp, irownum, orownum)`	Copy an entire row (only for tables with identical columns)
`tbrcsc (itp, otp, icptr, ocptr, irownum, orownum, ncols)`	Copy a row, but copy only selected columns
`tbrswp (tp, row1, row2)`	Swap two rows
`tbtsrt (tp, numcols, colptr, fold, nindex, index)`	Sort an index for the table rows
`tbrdel (tp, firstrow, lastrow)`	Delete a range of rows
`tbrnll (tp, firstrow, lastrow)`	Set all columns in a range of rows to INDEF
`tbcnam (tp, colptr, colname)`	Change the name of a column
`tbcfmt (tp, colptr, colfmt)`	Change the format for printing a column
`tbcnit (tp, colptr, colunits)`	Change the units for a column
`colptr = tbcnum (tp, colnum)`	Get the column pointer from the column number

Table E.15: Table Utility Procedures.

Bibliography

[Abramowitz65] Abramowitz, M. and I. Stegun, *Handbook of Mathematical Functions*, 1965, Dover, New York.

[Downey82] E. Downey, D. Tody, and G. Jacoby, *CL Programmer's Manual*, 1982. Describes programming in the IRAF cl , including definition of the parameter file syntax, `doc$clman.ms` in IRAF.

[Downey83] E. Downey, et. al., *IRAF Standards and Conventions*, 1983. Describes conventions for writing SPP applications.

[Seaman92] R. Seaman, *An Introductory User's Guide to IRAF SPP Programming*, 1992. Complementary to this document, providing more implementation instructions and examples.

[Shames86] P. Shames and D. Tody, *A User's Introduction to the IRAF Command Language*. Another introduction to using the IRAF cl, `doc$cluser.tex` in IRAF.

[Tody83] D. Tody, *A Reference Manual for the IRAF Subset Preprocessor Language*, 1983. The original SPP reference manual. Available in IRAF as `doc$spp.hlp`.

[Tody83b] D. Tody, *Programmer's Crib Sheet for the IRAF Program Interface*, 1983. Original description of the VOS interface. Available in IRAF as `doc$crib.hlp`

[Tody84] D. Tody and G. Jacoby, *A User's Guide to the IRAF Command Language*, 1984. An introduction to using the IRAF cl, `doc$cluser.ms` in IRAF.

[Tody84b] D. Tody, *Graphics I/O Design*, 1984. The defining description of the **gio** library package for using graphics in SPP applications and the basis for the material presented in "Vector Graphics — gio" on page 114, `gio$doc/gio.hlp` in IRAF. See also `help cursors` in the IRAF cl for more information on graphics cursor interaction.

[Tody86] D. Tody, *Named External Parameter Sets in the CL*, 1986. The defining description of parameter sets in the IRAF cl , `doc$pset.ms` in IRAF.

[Tody88] D. Tody, *The IRAF Pixel List Package and IMIO Extensions to Support Image Masks*, 1988. The defining description of the plio library package and the basis for "Pixel Lists — plio" on page 127, `plio$PLIO.hlp` in IRAF.

[Tody89] D. Tody, *Mini-WCS Interface*, 1989. The defining description of the **mwcs** library package and the basis for "World Coordinates — mwcs" on page 129, `mwcs$MWCS.hlp` in IRAF.

Glossary

The following terms and acronyms are used in SPP, additional terms, generic to IRAF and STSDAS, are defined in the glossary in the *STSDAS Users Guide.*

access mode - How to open a file or image, read-only, read-write, for example.

argument - A value passed to a procedure. Also in the cl, a value passed to a task.

assignment - Replace the value of a variable.

asynchronous error - An error that results in control passing to a procedure other than the one in which the error occured.

boolean - A binary value, yes or no, true or false.

cell array - Grey scale image, sometimes also known as a raster or pixmap.

clio - Interaction with the cl. The VOS library of procedures for accessing cl parameters.

coercion - (As in *type coercion*.) Conversion of a value from one data type into another. Commonly by simple assignment of variables.

comment - Text in a program file that is not executed and is retained for information purposes. In SPP, comments begin with the # character.

common blocks - A set of ariables available to more than one procedure through common memory.

compile - To process source code into *object code*, combined with other procedures to make a program (see "link").

constant - An identifier having a fixed value.

data structure - The organization of data in a commonly accessible form. Often includes multiple data types and arrays.

data type - The basic attribute of a variable, constant or data value such as integer, floating point (real), double precision, boolean or complex.

dimensionality - The number and sizes of axes of an array.

double precision - A floating point value having more bits for the mantissa.

error - An abnormal condition in a program

error handler - A procedure called on an error condition to perform some activity such as closing files and cleaning up memory.

escape sequence - Characters including metacharacters that change the interpretation of other characters. The backslash ("\") is an escape to permit specifying a character constant in SPP.

file descriptor - A pointer to a structure describing a file.

file name template - A file name possibly referring to more than one file, including wild-cards or a list of individual file names, or a pointer to a file containing a list of files.

filter - A program that transforms a data set in some way without altering the fundamental structure of the data.

fio - Basic binary file I/O not limited to images or any particular structure.

flag - A variable indicating one of a set of possible conditions.

floating point - A value having a decimal and fractional part.

fmtio - Formatted I/O. The procedures for standard text and numeric I/O to files and terminals.

function - A procedure returning a value assigned to a variable.

gcur - Graphics cursor. Treated by the cl as a cl parameter and accessed in SPP via a clio procedure returning the coordinates of the cursor.

generic operator - A function or operator that can be used for any of several data types.

generic preprocessor - The program that converts generic source into compilable code specific to a given data type.

gio - Graphics I/O. The set of VOS procedures for drawing graphs.

graphcap - The file that describes attributes of graphics devices.

header parameter - A value stored as part of an image file, used to describe the image.

heap memory - Dynamically allocated memory accessed with the `malloc` family of procedures.

identifier - A string or sequence of characters having a recognized meaning such as a variable or procedure name.

image section - (see "section.")

imcur - Image cursor. A cl parameter type returning coordinates from an image display.

imio - Image I/O. The library of procedures for accessing IRAF images.

include file - Source code that can be inserted as-is into other source by referring to a file name.

index - An integer constant or variable indicating a particular element of an array.

integer - A constant or variable having no fractional part.

intrinsic function - A function built in to the language. In general, the data type of the arguments and returned value may be any valid data type.

kernel - The low-level routines implementing the system. The system procedures dealing with a particular image format. The “device drivers” for rendering graphics on a class of devices.

keyword - An identifier or character string reserved for some purpose such as image header parameters.

learning - The capability of the IRAF cl to remember the value of a task parameter from execution to execution.

library - A file containing compiled procedures (object code) and linked with an application.

link - Combine compiled code to make an executable program.

logical task - An IRAF task implemented as part of a package or physical task.

longword boundary - Locations in data memory separating the longest addressable units of data.

macro - A string identified with a symbol and replaced by string substitution in code.

mask - An image whose values indicate particular properties of another image or matching size. A mask might specify bad detector element or relative errors of pixels.

matrix - A grouping of values in a rectangular array.

memio - The VOS library of procedures for dynamically allocating memory.

metacharacters - Literal characters interpreted by a parser.

mii - Machine Independent I/O. A method of converting data that is independent of the host computer architecture. The library of procedures to perform these conversions.

mixed mode - An expression involving variables or constants of different data types.

mkpkg - The program that combines compiling, linking and maintaining source and objects.

mode - Manner in which CL handles prompting and learning when dealing with parameters.

mtio - Magnetic tape I/O.

mwcs - Mini World Coordinate System.

NDC - Normalized Device Coordinates. A graphics coordinate system relative to the device.

newline - A character interpreted as a delimiter between lines of text.

OIF - Old IRAF format. The native IRAF image format consisting of a pair of binary files, a header describing the image and a separate pixel file.

operators - Functions combining values in an expression such as `+`, `-`, `&&`.

osb - Bit and byte operations.

package - A library of procedures grouped by common function or a group of application tasks grouped by common function.

parameters - The arguments to a program accessed via clio from the cl.

pen - The logical position of drawing graphics.

physical task - An executable IRAF program, possibly comprising multiple "logical tasks."

plio - Pixel list I/O.

pointer - Reference to dynamically allocated memory addresses.

predefined constant - A program value defined at compile time, either in a `data` statement or as a symbolic macro.

preprocessor - An operation applied to program source before compilation. The generic preprocessor permits defining common code for multiple data types. xc is the preprocessor for converting SPP into Fortran.

primitives - Relatively low-level procedures performing well-defined functions.

procedure - The smallest executable unit of a program, called by another procedure or as a task from the cl.

prompt - A request for input from the user via a prompt to the terminal (window).

pset - A file containing cl parameters. A pset must be defined as a task in the cl and assigned to another task parameter. The parameter values are then available to an application as any cl parameter.

pushback - The opposite of reading from an input stream or file. Data pushed back is then available for reading.

QPOE - Quick Position-Oriented Event image; the native image format for the **xray** analysis package developed by PROS.

Ratfor - Rational Fortran. One of the steps in converting SPP into Fortran.

scalar - A single-valued variable.

section - (As in “image section.”) A portion of an IRAF image treated in an application as any image.

stack memory - Dynmically allocated memory.

STF - STSDAS format images also known as GEIS format. The native image format for HST observations. STF images are largely interchangeable with OIF images.

stream - A source of data logically consisting of a string of characters. The standard input (STDIN), standard output (STDOUT) and standard graphics (STDGRAPH) are the most commonly used streams.

string - Sequence of characters enclosed in quotes, for example, `"abc"`.

structure - See “data structure.”

symbolic constant - A numeric value or literal string represented by an identifier. In compiled code, the value replaces the identifier by simple string substitution.

task - A program known to IRAF, a command in the cl.

templates - See “file name templates.”

termcap - The IRAF file that describes attributes of text terminals.

token - The smallest sequence of characters recognized by a parser, a number or identifier, for example.

tty - Terminal I/O.

unary operator - An operator requiring only one operand, such as negation.

vector - An array, a contiguous group of values accessed through a common variable name.

vops - Vector operators. The library of procedures that operate on arrays, potentially optimized for the host architecture.

VOS - Virtual operating system. The set of procedures called by an applications tasks for performing IRAF functions.

WCS - World Coordinate System. Coordinates associated with data rather than a device or an arbitrary scale.

white space - Any number of tabs, spaces or newline characters separating entities in a string.

word - The fundamental unit of accessing data in a program, usually several bytes long. The word size varies between host architectures.

xc - The program that compiles and links SPP, Fortran, and C code to produce an executable, or physical task.

Index

M

N

O

P

Q

R