A C Interface to IRAF's VOS Library

Allen Farris
Space Telescope Science Institute

February 15, 1996
Updated by M.D. De La Peña, 07 December 1999

1 Prologue

This document should only be used as a reference manual which describes how to generate C interface functions. The relevant discussion of the process of generating the interface functions is contained in Sections 4 and 5. Sections 2 and 3 have been superseded and expanded by the "CVOS User's Guide and Reference Manual" (De La Peña 1999). Sections 2, 3, and 6 of this document should be considered obsolete. This document is presented in its entirety for completeness only.

2 Introduction

The IRAF software environment provides a Virtual Operating System (VOS) that contains a large number of functions organized into various libraries. IRAF's VOS has its own implementation of a task and its own interface to these libraries. This document describes the development of a system for building interface functions to these libraries so that one may access the IRAF functions via ANSI C programs, compiled, linked and executed within the native operating system. In other words, this system allows one to treat the IRAF libraries much like other prominent libraries, such as the X Window System.

While a major purpose of this document is to describe how to generate the interface functions, it also contains interface functions to the clio, imio, mwcs, and qpoe IRAF libraries and to the STSDAS synphot and tables packages. These are provided both to illustrate the process and as completed interfaces. The interface functions themselves provide a calling sequence that is reasonable from a C language perspective, performing necessary data conversions and concealing other awkward issues in calling the actual IRAF functions. These interfaces and libraries are being used within the STSDAS pipelines for STIS and NICMOS.

Several naming conventions have been adopted in this system. The name of the C interface function that one calls to access a particular IRAF function is formed by prefixing "c_" to the IRAF function name. The name of the C header file that provides the definition of these interface functions for a particular library is formed by the letter "x" plus the name of the IRAF library. So, the header file for the clio interface library is "xclio.h"; for imio it is "ximio.h", etc. The header file "c_iraf.h" also provides commonly needed definitions for these interface functions. Names have been chosen to minimize collisions with other commonly used libraries.

The remainder of this document discusses two example programs to give an introduction to how to use these functions, discusses more details on the general nature of the interface functions themselves, and also describes the process of constructing the interface functions. A section is also included describing how to install the software.

3 Using the C Interface Functions

The C interface to the IRAF VOS library is best illustrated by some simple example programs. Two programs are discussed and are located in the test directory. After the cvos interface library has been built, these two programs may be compiled and executed. The first program, copyimg, uses the clio and imio IRAF libraries. It reads an image and copies it to another image, while prompting the user for names of input and output files. The second program, wrtable, constructs a table using the STSDAS tables package. For the details of how to compile and link these two programs see Section 5.

An important header file that is included by these programs is "c_iraf.h", containing definitions and declarations needed by the interface routines. Three additional data types are needed to accommodate the IRAF functions: Bool, IRAFPointer and IRAFComplex. These are typedefs defined in "c_iraf.h". Also included in "c_iraf.h" are the names of the IRAF data types and the names of the IRAF I/O modes, also needed by programs that use these interface programs. These are given below.

Included in the enumeration typedef IRAFType:

IRAF_BOOL	IRAF_CHAR	IRAF_SHORT	IRAF_INT
IRAF_LONG	IRAF_REAL	IRAF_DOUBLE	IRAF_COMPLEX
IRAF_POINTER	IRAF_STRUCT	IRAF_USHORT	IRAF_UBYTE

Included in the enumeration typedef IRAFIOMode:

IRAF_READ_ONLY	IRAF_READ_WRITE	IRAF_WRITE_ONLY
IRAF_APPEND	IRAF_NEW_FILE	IRAF_TEMP_FILE
IRAF_NEW_COPY		

3.1 Reading and Writing an Image

Program copyimg.c

```
# include <stdio.h>

# include <c_iraf.h>
# include <xclio.h>
# include <ximio.h>

void iraferr() {
        printf("IRAF error code %d\nIRAF message: %s\n",
                c_iraferr(),c_iraferrmsg());
        exit(0);
}

int main(int argc, char **argv) {
        IRAFPointer in, out;
        long *in_data;
        float *out_data;
        int i, j, k;
        int no_dims;
        long dim1, dim2;
        IRAFType in_type;
        char name[BUFSIZ];

        /* must always remember to call c_irafinit for host-level tasks */
        c_irafinit(argc, argv);

        /* install the error handler */
        c_pusherr(iraferr);

        printf("Test program test1 -- basic test of clio and imio\n");

        /* a little test of clio */
        c_clgstr("Enter the name of the input image", name, BUFSIZ);
        c_clpstr("name",name);

        printf("Reading image file %s\n",name);

        /* open the image */
        in = c_immap(name,IRAF_READ_ONLY,0);
        printf("Error code is %d\n",c_iraferr());

        /* get the number of dimensions */
        no_dims = c_imgndim(in);
        if (no_dims != 2) {
                printf("Sorry! This example only works for two dimensions\n");
                exit(1);
        }
        /* and the size */
        dim1 = c_imglen(in,1);
        dim2 = c_imglen(in,2);
        /* get the type of pixel */
        in_type = (IRAFType)c_imgtypepix(in);
        printf("Input:  %d dimensions -- dim1 = %d  dim2 = %d of type %d\n",
                no_dims, dim1, dim2, in_type);
        printf("Error code is %d\n",c_iraferr());

        /* Now, we'll open the output image */
        c_clgstr("Enter the name of the output image", name, BUFSIZ);
        c_clpstr("name",name);

        printf("Copying the input image to %s\n",name);

        /* open the image */
```

```
63:           out = c_immap(name,IRAF_NEW_COPY,in);
64:
65:           /* change the pixel type to float */
66:           c_imptypepix(out, (int)IRAF_REAL);
67:
68:           printf("Output:  %d dimensions -- dim1 = %d  dim2 = %d of type %d\n",
69:                   c_imgndim(out), c_imglen(out,1), c_imglen(out,2),
70:                   c_imgtypepix(out));
71:
72:           /* read the image values, print some of them, and write output */
73:           k = dim1 < 5 ? dim1 : 5;
74:           for (i = 1; i <= dim2; ++i) {
75:               in_data = c_imgl2l(in,i);
76:               printf("%d: ", i);
77:               for (j = 0; j < k; ++j)
78:                   printf("%d ", in_data[j]);
79:               if (dim1 > k) {
80:                   printf("~~ ");
81:                   for (j = k; j > 0; --j)
82:                       printf("%d ", in_data[dim1 - j]);
83:               }
84:               printf("\n");
85:               out_data = c_impl2r(out,i);
86:               for (j = 0; j < dim1; ++j)
87:                   out_data[j] = in_data[j];
88:
89:           }
90:           printf("Error code is %d\n",c_iraferr());
91:
92:           /* close both images */
93:           c_imunmap(in);
94:           c_imunmap(out);
95:           printf("Error code is %d\n",c_iraferr());
96:
97:           return 0;
98:   }
```

The program must include the relevant header files, lines 3 - 5 . One only needs to include the header files pertaining to the IRAF libraries actually used, in this case the clio and imio libraries. The "c_iraf.h" file is needed by all programs using the IRAF interface functions. If it is not explicitly included, as is done here, it is automatically included by whatever IRAF header libraries are included. The error handling function at lines 7 - 11 catches any IRAF errors and displays them (more on the error handling later).

Within the main routine, the declaration at line 14 declares the variables "in" and "out" to be IRAF pointers. The declaration at line 20 declares "in_type" to be the name of an IRAF data type. One of the first things that must be done in the program is to call the function that initializes the IRAF libraries (line 24). This function must be called prior to executing any IRAF library functions. Failure to call this initialization function will almost certainly result in a core dump. This function is declared in the "c_iraf.h" file.

After executing any IRAF function one may access the IRAF error status by calling the routines: "c_iraferr()" or "c_iraferrmsg()". The first merely returns the IRAF error number. The second returns a text string with the error message itself. These functions can be examined after any operation. "c_iraferr()" will return 0 if there was no error.

Rather than check the error status after each operation one can install a global error handler. Line 27 installs the error handling function previously mentioned. The C interface functions contain a global error handling stack. If you have a function of the form declared at lines 7 - 11 , at any point in the program you can push it onto the error handling stack by calling the “c_pusherr” function. (The “c_pusherr” takes one argument, a pointer to a function that takes no arguments and returns void.) There is a stack of function pointers (limited to a nesting depth of 32, which is arbitrary). If there is a non-zero pointer on the stack, that function is automatically called when an error occurs. You are free to do anything in the error handling function: print a message, abort, or whatever. Execution continues after executing the error handling function. Of course, there is also a pop function: “c_poperr()”. So, you can manipulate the error handling stack as you see fit, e. g. by installing a new error handling function at the beginning of some complex procedure and popping it at the end. At any point, if you want to override the action of the global handler and give special handling after executing some function, you can merely push 0 onto the stack before executing the operation. That prevents the global error handler from being called. Of course, the pop function then restores the global error handler.

Lines 32 - 33 use IRAF clio functions to prompt the user for the input filename and echo the choice back to the user. If the C program is run as a host operating system task (not a native IRAF task), then the program can also use the C standard I/O library. The image is opened by making a call to the IRAF immap function using the read-only I/O mode (line 38). Following the call to immap, the error code is displayed. Lines 42 - 54 get the dimensions and other attributes of the image and display this information to the user. Lines 57 - 58 prompt the user for the output filename and line 63 opens the output image as a new-copy of the input image. Line 66 changes the output pixel type to single-precision float data. Lines 73 - 89 read the input data using the “c_imgl2l” function and write it to the output file using the “c_impl2r” function, while displaying portions of the input lines in the process. Finally, lines 93 - 94 close both the input and output images using calls to “c_imunmap”.

When the program copyimg is run using the file timage.fit (located in the test directory) as input, the result is given below.

Output of Program copyimg

```
Test program test1 -- basic test of clio and imio
Enter the name of the input image: name="timage.fit"
Reading image file timage.fit
Error code is 0
Input:  2 dimensions -- dim1 = 5  dim2 = 4 of type 5
Error code is 0
Enter the name of the output image: name="timageout.fit"
Copying the input image to timageout.fit
Output:  2 dimensions -- dim1 = 5  dim2 = 4 of type 6
1: 50 75 55 30 15
2: 58 138 160 124 48
3: 72 160 2200 140 45
4: 40 65 75 65 38
Error code is 0
Error code is 0
```

3.2 Constructing a Table using the STSDAS Tables Package

Program wrtable.c

```
# include <stdio.h>
# include <xtables.h>

void iraferr() {
        printf("IRAF error code %d\nIRAF message: %s\n",
                c_iraferr(),c_iraferrmsg());
        exit(0);
}

int main(int argc, char **argv) {

# define NCOLS  7                               /* column numbers: */
        const int REGION =      0;      /* region plate is located in */
        const int PLATE =       1;      /* plate id */
        const int RA =          2;      /* right ascension */
        const int DEC =         3;      /* declination */
        const int EPOCH =       4;      /* epoch of observation */
        const int SURVEY =      5;      /* type of survey */
        const int DISK =        6;      /* CD-ROM disk plate is on */
        int     row;
        char    *region;
        char    *plate;
        double  ra;
        double  dec;
        double  epoch;
        char    *survey;
        int     disk;
        IRAFPointer tab;                /* pointer to table struct */
        IRAFPointer col[NCOLS];         /* pointers to column info */

        c_irafinit(argc, argv);

        c_pusherr(iraferr); /* install a global error handler */

        printf("Test program test3 -- basic test of tables\n");

        tab = c_tbtopn("test3tab",IRAF_NEW_FILE,0); /* Open output table */

        /* Define columns.  The "Name" column is a string up to 20 char long. */
        c_tbcdef1(tab,&col[REGION],"REGION", "", "", -6, 1);
        c_tbcdef1(tab,&col[PLATE], "PLATE",  "", "", -4, 1);
        c_tbcdef1(tab,&col[RA],    "RA", "hours", "%14.3h", IRAF_DOUBLE, 1);
        c_tbcdef1(tab,&col[DEC],   "DEC", "degrees","%14.3h",IRAF_DOUBLE, 1);
        c_tbcdef1(tab,&col[EPOCH], "EPOCH", "years","%10.3f",IRAF_DOUBLE, 1);
        c_tbcdef1(tab,&col[SURVEY],"SURVEY", "", "", -3, 1);
        c_tbcdef1(tab,&col[DISK],  "DISK",   "",       "%3d",IRAF_INT, 1);

        /* Create the output table file. */
        c_tbtcre(tab);

        /* Add a history record. */
        c_tbhadt(tab,"history","Created from GetImage/headers/lo_comp.lis");

        row = 0;

        /* We'll just add a few rows, in a not very sophisticated manner. */

        row++;
        region = "S256";
        plate = "000Y";
        ra = 105.37155;
        dec = -45.07291;
```

```
 63:            epoch = 1980.122;
 64:            survey = "UK";
 65:            disk = 18;
 66:            c_tbeptt(tab,col[REGION],row,region);
 67:            c_tbeptt(tab,col[PLATE],row,plate);
 68:            c_tbeptd(tab,col[RA],row,ra);
 69:            c_tbeptd(tab,col[DEC],row,dec);
 70:            c_tbeptd(tab,col[EPOCH],row,epoch);
 71:            c_tbeptt(tab,col[SURVEY],row,survey);
 72:            c_tbepti(tab,col[DISK],row,disk);
 73:
 74:            row++;
 75:            region = "S734";
 76:            plate = "006Q";
 77:            ra = 275.68950;
 78:            dec = -9.97406;
 79:            epoch = 1978.647;
 80:            survey = "UK";
 81:            disk = 50;
 82:            c_tbeptt(tab,col[REGION],row,region);
 83:            c_tbeptt(tab,col[PLATE],row,plate);
 84:            c_tbeptd(tab,col[RA],row,ra);
 85:            c_tbeptd(tab,col[DEC],row,dec);
 86:            c_tbeptd(tab,col[EPOCH],row,epoch);
 87:            c_tbeptt(tab,col[SURVEY],row,survey);
 88:            c_tbepti(tab,col[DISK],row,disk);
 89:
 90:            row++;
 91:            region = "S888";
 92:            plate = "006R";
 93:            ra = 325.64040;
 94:            dec = 0.22890;
 95:            epoch = 1982.563;
 96:            survey = "UK";
 97:            disk = 60;
 98:            c_tbeptt(tab,col[REGION],row,region);
 99:            c_tbeptt(tab,col[PLATE],row,plate);
100:            c_tbeptd(tab,col[RA],row,ra);
101:            c_tbeptd(tab,col[DEC],row,dec);
102:            c_tbeptd(tab,col[EPOCH],row,epoch);
103:            c_tbeptt(tab,col[SURVEY],row,survey);
104:            c_tbepti(tab,col[DISK],row,disk);
105:
106:
107:            /* Close the table */
108:            c_tbtclo(tab);
109:
110:            printf("Created and closed the tables\n");
111:
112:            return 0;
113:    }
```

The program “wrtable” uses the STSDAS tables package to create a table. The program is similar in structure to the “copyimg” program described previously. Lines 4 - 8 is the global error handling function, as in the previous example. Lines 28 - 29 declare “tab” and “col” to be IRAF pointers. Lines 31 - 33 initialize the IRAF libraries and install the error handling function. Line 37 opens the table and lines 40 - 46 define the columns within the table. The output table file is then created (line 49) and a history record added (line 52). Several rows are then added to the table, using the “c_tbeptX” functions. The table is then closed at line 108 , using the “c_tbtclo” function.

4 The General Structure of the Interface Functions

A few comments on the general structure of the interface functions are in order. Many of these functions are very simple, but some of them are surprisingly complex. Keep in mind that both the function declarations in the ".h" files and the function themselves in the ".c" files are automatically generated. In general, an interface function has four major things that have to be done: first, declare the external routine in the IRAF library that is to be called; second, map the parameters in the IRAF routine to appropriate C parameters; third, perform any necessary data conversions on the parameters and the data item returned; and fourth, provide for handling any IRAF errors that occur. The data conversions that must be performed between SPP and C are indicated below.

SPP	ANSI C
char (2 bytes)	char (1 byte)
bool	Bool (implemented with enum and typedef)
complex	IRAFComplex (implemented with struct and typedef)
pointer	IRAFPointer (implemented with typedef or converted to C pointer)

These points are illustrated below with the interface routine for the "immap" function from the IRAF imio library.

Extract from Generated File ximio.h

```
IRAFPointer c_immap(char *imspec, int acmode, IRAFPointer hdr_arg);
/* char imspec[ARB]            #I image specification */
/* int  acmode                 #I image access mode */
/* int  hdr_arg                #I length of user fields, or header pointer */
```

Within the ".h" file, the code generation process places the declaration of the particular function. This is followed by comments which are made from lines, in the corresponding SPP function, that declare the parameters. Frequently, these lines contain useful comments on the meaning of the parameters.

Extract from Generated File ximio.c

```
 1:            extern IRAFPointer immap_(short *, int *, IRAFPointer *);
 2:    IRAFPointer c_immap(char *imspec, int acmode, IRAFPointer hdr_arg) {
 3:            IRAFPointer rtn;
 4:            clear_cvoserr();
 5:            xerpsh_();
 6:            rtn = immap_(char2iraf(imspec,1), &acmode, &hdr_arg);
 7:            if (xerpoi_())
 8:                set_cvoserr();
 9:            return rtn;
10:    }
```

I will just point out a couple of interesting features from the interface routine. Line 1 declares the SPP procedure, after which the function begins. Any returned data item is then declared. Line

4 clears the error status and error message from any previous function and line 6 calls the SPP function. In the calling sequence, the C character string is converted to an array of shorts, the form of SPP character strings. The calls to the SPP functions “xerpsh” and “xerpoi” set up the proper environment to retrieve the IRAF error code, in the event of an error. Finally, the data item is returned and the function ended.

5 The Process of Generating the Interface Functions

One of the driving principles behind this design was that no modifications to the SPP source code were to be required. This means that in order to automatically generate the interface functions, one must gather as much information as possible from the SPP source code. Unfortunately, at least from a practical point of view, if you do this there is still some information that needs to be supplied to the generation process. The approach that is taken here is to construct an “interface definition” file that drives the code generation process. This file contains simple (usually) one-line entries that supply the added information that is needed to generate the interface functions. This process is depicted in Figure 1.

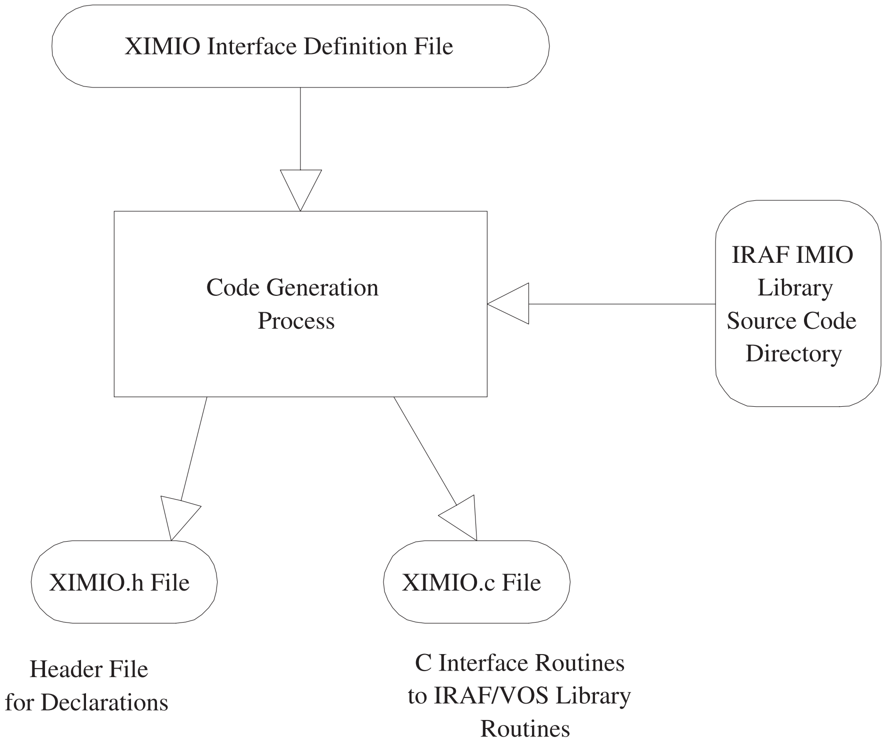

Figure 1: Generating the C Interface Functions

The process of constructing the “interface definition” file is not really as difficult as it might first appear. The majority of cases are very straightforward and easy to deal with. However, inevitably, there are a few pathological cases that make life difficult.

The program that performs the code generation process is “gencbind”, implemented as a C program. It should really be rewritten and probably implemented using Lex and Yacc. It started out as a fairly simple program and evolved as more of these “pathological” cases were encountered. The result is that the parser is not as robust as one would like and the syntax to handle some of the special cases is a little awkward. Nevertheless, the basic mechanisms do work. A description of how to run the program is given below.

The process of generating the interface functions will first be illustrated using the IRAF clio library, a fairly simple case. The steps are the following, where $iraf represents the IRAF source code directory tree.

(1) For a given IRAF directory, begin by doing a grep on “procedure” and save the results in a file. If there are also relevant functions in subdirectories of the IRAF source directory the grep should include the subdirectories.

```
>cd $iraf/sys/clio
>grep procedure *.x >XCLIO.def

>cd $iraf/sys/imio
>grep procedure *.x */*.x >XIMIO.def
```

Then edit the file in the following manner.

(2) Add as the first line of the file XCLIO.def , the name of the interface library (which in this case is “xclio”). This name will be used to construct ".h" and ".c" files.

(3) Add any lines that should be written directly to the “.h” or “.c” files, for example include statements or defines. Lines beginning with “/h/” are written directly into the “.h” file, with the “/h/” at the beginning removed. Likewise, lines beginning with “/c/” are written directly into the “.c” file.

(4) Remove all entries from XCLIO.def except those routines that will be part of the public interface to the clio library.

(5) With the remaining entries in XCLIO.def, mark the “input-only” parameters with an ‘:i’ appended to the name of the parameter. Do this for character strings also. If a parameter is “output-only”, it should be marked with an ‘:o’ (this really only has an affect on character strings, where it helps to optimize things a bit). If nothing is appended, the parameter is assumed to be “input/output”. In the case of an “output-only” character string, make sure that the parameter following the character string is the maximum length of the string.

(6) If the C function name is to be different from the SPP procedure name, it should be appended to the SPP procedure name separated by a ‘|’. Likewise, if the C parameter name should be different from the SPP parameter name it should be appended to the SPP parameter name (i. e., following the ‘:i’ or ‘:o’ if any) separated by a ‘|’.

(7) Within the XCLIO.def file you may add comments. Lines within the XCLIO.def file beginning with ‘#’ and lines with a ‘#’ following the name of the SPP file from the grep operation are assumed to be comments and are ignored.

(8) Within the XCLIO.def file lines may be continued. Lines ending with the ‘\’ character immediately followed by a newline character are assumed to be continued onto the next line.

The result of following this procedure is the file XCLIO.def, which is given below.

XCLIO.def Interface Definition File

```
xclio
# This is the input file for the IRAF/VOS clio library.

/h/# include <c_iraf.h>

# CLIO defines
/h/
/h/# define CL_PRTYPE   1
/h/# define CL_PCACHE   2
/h/
/h/# define PR_CONNECTED 1
/h/# define PR_DETACHED  2
/h/# define PR_HOST      3
/h/
/h/# define PR_NOEXIT   0
/h/# define PR_EXIT     1
/h/

# CLIO includes and private routines
/c/# include <xclio.h>
/c/# include <c_iraf_priv.h>
/c/# include <stdlib.h>
/c/

clseti.x:procedure clseti (parameter:i, value:i)
clstati.x:int procedure clstati (parameter:i)
clgetb.x:bool procedure clgetb (param:i)
clgetc.x:char procedure clgetc (param:i)
clgetd.x:double procedure clgetd (param:i)
clgeti.x:int procedure clgeti (param:i)
clgetl.x:long procedure clgetl (param:i)
clgetr.x:real procedure clgetr|clgetf (param:i)
clgets.x:short procedure clgets (param:i)
clgetx.x:complex procedure clgetx (param:i)
clputb.x:procedure clputb (param:i, bval:i)
clputc.x:procedure clputc (param:i, cval:i)
clputd.x:procedure clputd (param:i, dval:i)
clputi.x:procedure clputi (param:i, value:i)
clputi.x:procedure clputs (param:i, value:i)
clputi.x:procedure clputl (param:i, lval:i)
clputr.x:procedure clputr|clputf (param:i, rval:i)
clputx.x:procedure clputx (param:i, xval:i)
clgstr.x:procedure clgstr (param:i, outstr:o, maxch:i)
clpstr.x:procedure clpstr (param:i, value:i)

clglpb.x:int procedure clglpb (param:i, bval)
clglpc.x:int procedure clglpc (param:i, cval)
clglpd.x:int procedure clglpd (param:i, dval)
clglpi.x:int procedure clglpi (param:i, ival)
clglpl.x:int procedure clglpl (param:i, lval)
clglpr.x:int procedure clglpr|clglpf (param:i, rval)
clglps.x:int procedure clglps (param:i, sval)
clglpx.x:int procedure clglpx (param:i, xval)
```

```
clglstr.x:int procedure clglstr (param:i, outstr:o, maxch:i)

clgcur.x:int procedure clgcur (param:i, wx, wy, wcs, key, strval:o, maxch:i)
clgkey.x:int procedure clgkey (param:i, key, strval:o, maxch:i)
clgwrd.x:int procedure clgwrd (param:i, keyword:o, maxchar:i, dictionary:i)

clopset.x:pointer procedure clopset (pset:i)
clcpset.x:procedure clcpset (pp:i)
clgpsetb.x:bool procedure clgpsetb (pp:i, parname:i)
clgpsetc.x:char procedure clgpsetc (pp:i, parname:i)
clgpsetd.x:double procedure clgpsetd (pp:i, parname:i)
clgpseti.x:int procedure clgpseti (pp:i, parname:i)
clgpsetl.x:long procedure clgpsetl (pp:i, parname:i)
clgpsetr.x:real procedure clgpsetr|clgpsetf (pp:i, parname:i)
clgpsets.x:short procedure clgpsets (pp:i, parname:i)
clgpsetx.x:complex procedure clgpsetx (pp:i, parname:i)
clppsetb.x:procedure clppsetb (pp:i, parname:i, bval:i)
clppsetc.x:procedure clppsetc (pp:i, parname:i, cval:i)
clppsetd.x:procedure clppsetd (pp:i, parname:i, dval:i)
clppseti.x:procedure clppseti (pp:i, parname:i, ival:i)
clppsetl.x:procedure clppsetl (pp:i, parname:i, lval:i)
clppsetr.x:procedure clppsetr|clppsetf (pp:i, parname:i, rval:i)
clppsets.x:procedure clppsets (pp:i, parname:i, sval:i)
clppsetx.x:procedure clppsetx (pp:i, parname:i, xval:i)

clgpseta.x:procedure clgpseta (pp:i, pname:i, outstr:o, maxch:i)
clppseta.x:procedure clppseta (pp:i, pname:i, sval:i)
cllpset.x:procedure cllpset (pp:i, fd:i, format:i)
clepset.x:procedure clepset (pp:i)
```

The full syntax for a procedure entry in the interface definition file is given in Figure 2. The notation used is the following: literals are enclosed in single quotes, optional parts are enclosed in square brackets, and alternatives are separated by the ‘|’ character.

spp_procedure_entry :=
    source_name ‘:’ [return_spec] ‘procedure’ proc_spec ‘(’ parm_list ‘)’
source_name := name
return_spec := spp_type [conversion_spec]
conversion_spec := ‘<’ spp_type ‘>’
proc_spec := spp_proc_name [external_name] [c_proc_name]
spp_proc_name := name
external_name := ‘$’ name
c_proc_name := ‘|’ name
parm_list := parm_item [‘,’ parm_item ‘,’ . . .]
parm_item :=
    spp_parm_name [io_spec] [conversion_spec] [string_size_spec] [c_parm_name]
io_spec := ‘:i’ | ‘:o’
string_size_spec := ‘[’ expression [: expression] ‘]’
c_parm_name := ‘|’ name
spp_type := ‘int’ | ‘char’ | ‘real’ | ‘double’ | ‘pointer’ | ‘long’ | ‘bool’ | ‘complex’ | ‘void’
name := any sequence of characters except space, tab, comma, :, $, |, [,], {, }, (,), <, >

Figure 2: Procedure Entry Syntax in Interface Definition

Some of the more complex parts of the syntax (including an explanation of expressions) will be illustrated using some of the special cases from the interface files included in the system.

One of the items of information that must be added to generate interface functions is how to convert SPP pointers. In some cases, the SPP pointer should not be converted to a C pointer because it only serves as a tag representing an address of an internal IRAF data structure. But in other cases the pointer must be converted to a proper C pointer. However, the SPP declaration of a pointer does not specify its type; only the use of the pointer does that. So, if pointers are to be converted, one must specify how to do the conversion. That is the purpose of the "conversion_spec" in the syntax. Its use is illustrated by the following entries from the "XIMIO.def" file.

```
immap.x:pointer procedure immap (imspec:i, acmode:i, hdr_arg:i<p>)
tf/imgnld.x:int procedure imgnld (imdes:i, lineptr:o<d>, v)
tf/imgs2r.x:pointer<r> procedure imgs2r (im:i, x1:i, x2:i, y1:i, y2:i)
```

Within the conversion specification one can specify only the first letter of the SPP data type; you don't have to spell out the entire word. The first entry changes the data type of "hdr_arg" to a pointer; the second specifies that the output pointer, "lineptr", is to be converted to a pointer to a double; and the third specifies that the returned pointer be converted to a pointer to a real.

One of the options on the procedure entry syntax allows one to substitute a different C procedure name for the SPP procedure name (the default C name is "c_"+ the SPP name) or a different C parameter name for the SPP parameter name. This is done with the "c_proc_name" and "c_parm_name" options. Another naming option that one must deal with is external names of SPP routines. Gencbind uses the SPP algorithm for generating external names: take the first 5 letters of the SPP name and the last letter and then append an underscore, for a total of seven letters, at most. Unfortunately, this does not always yield a unique name. So, the "external_name" option allows one to designate the external name. The only case encountered in the libraries generated here is the following entry from the "XQPOE.def" file.

```
qpiostati.x:int procedure qpio_stati$qpiost (io:i, param:i)
```

One of the important data conversions that the C interface functions do is to convert character strings between C and SPP. The string conversion process is internal to the interface functions and utilizes internal buffers to do the conversion. These internal buffers are allocated (and re-allocated if necessary) to handle strings of arbitrary length. The conversion process can handle a string of characters (i. e., a one-dimensional array of characters) and a one-dimensional array of strings (i. e., a two-dimensional array of characters). The conversion process can also handle up to three string parameters in the same procedure call (I haven't encountered more than that and the limit is easy to change).

In the case of an "input-only" string, the case is easy to handle; the length is taken from the length of the C string. In the case of an "output" string, the size of the string must be specified. In most cases within SPP functions, this parameter follows the string parameter, but not always. That is what the string size specification is for. If there is an output string with no string specification present, gencbind assumes the following parameter is the length of the string (in this case, if the data type is not an “int” an error is flagged). The string specification is placed within square brackets and appended to the name. In the case of an array of strings, the size of the arrays must be specified; the colon separates the two sizes. There is sometimes a need to specify expressions for the sizes. This is done by enclosing the expression in ‘{’ and ‘}’; usually the size is a function of other variables in the calling sequence. There must be no spaces in the expression within the braces.

The following examples of string size specifications are from the “XTABLES.def” file. Some of these functions require arrays of strings and output strings of a fixed maximum size.

```
tbagt.x:int procedure tbagtt (tp:i, cp:i, row:i, cbuf:o[maxch:nelem], \
    maxch:i, first:i, nelem:i)

tbcgt.x:procedure tbcgtt (tp:i, cp:i, buffer:o[lenstr:{lastrow-firstrow+1}], \
    nullflag:o, lenstr:i, firstrow:i, lastrow:i)

tbhgnp.x:procedure tbhgnp (tp:i, parnum:i, keyword:o[SZ_KEYWORD], \
    dtype:o, str:o[SZ_PARREC])
```

All single and multi-dimensional data arrays (i.e., non-characters) are mapped from SPP to C in the same manner. If one has a one-dimensional array of type “double” in SPP, the corresponding C variable is declared as “double *”. In addition, for a two-dimensional array of type “double” in SPP, the corresponding C variable is still declared as “double *”. This mapping holds true for all higher dimensions. For variables of type other than characters, this mapping is handled automatically by the gencbind program. An example of passing single and two-dimensional arrays can be found in functions in the mwcs IRAF library.

In order to run the gencbind program, the command is:

```
gencbind OPTIONS

where OPTIONS is one or more of
        -f <interface definition file name>
        -c <code generation option>
        -d <SPP source code directory>
```

The “-f” option is used to specify the name of the interface definition file. The “-c” option is used to choose the implementation language for the interface routines. While bindings were planned for other languages (IDL and C++), these other languages are not currently accommodated in the interface generation code. Interface routines may only be created for the C language, and consequently, C is the default option. The default value for the interface definition file name is standard input, “stdin”. The default value for the SPP source code directory is the current directory. An example of using the gencbind program is:

>gencbind -f XTABLES.def -c C -d /usr/ra/tables/lib/tbtables/

The gencbind program will append an underscore as part of the external name of SPP routines.

6 Installing the Interface Functions and Test Programs

The process of installing the system is straightforward. The makefile for compiling the program "gencbind" and generating the interface functions is given below. It is very simple and only minor editing is needed. Just edit it and type "make". The makefile uses GNU's C compiler (lines 1 - 2). If you don't have "gcc" use your favorite C compiler. You also have to change lines 5 - 7 . These are respectively: (1) the full path name of the IRAF source directory on your system, (2) the full path name of the STSDAS tables package, and (3) the full path name of the STSDAS package (to generate the interface to synphot). If you don't want to generate either the tables or synphot interfaces, just remove them from lines 9 - 10 . Included in the file is a directory called "interface" that contains a copy of the generated source code for the completed libraries.

Makefile for installing the code generation

```
 1:    CC=gcc
 2:    LD=gcc
 3:    CFLAGS=-c -I. -O
 4:    LFLAGS=
 5:    IRAFDIR=/usr/stsci/irafx
 6:    SDASTABDIR=/usr/stsci
 7:    SDASDIR=/usr/stsci/stsdasx
 8:
 9:    all: gencbind c_iraf_priv.o irafinit.o irafsppinit.o xclio.o ximio.o \
10:            xtables.o xqpoe.o xsynphot.o xmwcs.o
11:
12:    clean:
13:            rm gencbind *.o xclio.c xclio.h ximio.c ximio.h \
14:                    xtables.h xqpoe.h xsynphot.h xmwcs.h \
15:                    xtables.c xqpoe.c xsynphot.c xmwcs.c
16:
17:    gencbind: gencbind.o
18:            $(LD) $(LFLAGS) -o gencbind gencbind.o
19:
20:    gencbind.o: gencbind.c
21:            $(CC) $(CFLAGS) gencbind.c
22:
23:    c_iraf_priv.o: c_iraf_priv.c
24:            $(CC) $(CFLAGS) c_iraf_priv.c
25:
26:    irafinit.o: irafinit.c
27:            $(CC) $(CFLAGS) -I$(IRAFDIR) irafinit.c
28:
29:    irafsppinit.o: irafsppinit.x
30:            xc -c irafsppinit.x
31:
32:    xclio.o: XCLIO.def c_iraf_priv.h
33:            gencbind -f XCLIO.def -c c -d $(IRAFDIR)/sys/clio
34:            $(CC) $(CFLAGS) xclio.c
35:
```

```
ximio.o: XIMIO.def c_iraf_priv.h imspp.x
        gencbind -f XIMIO.def -c c -d $(IRAFDIR)/sys/imio
        gencbind -f XIMIOa.def -c c
        cat ximio.h ximioa.h >Tximio.h
        cat ximio.c ximioa.c >Tximio.c
        rm ximioa.h ximioa.c
        mv Tximio.h ximio.h
        mv Tximio.c ximio.c
        $(CC) $(CFLAGS) ximio.c
        xc -c imspp.x

xsynphot.o: XSYNPHOT.def
        gencbind -f XSYNPHOT.def -c c -d $(SDASDIR)/lib/synphot
        $(CC) $(CFLAGS) xsynphot.c

xqpoe.o: XQPOE.def
        gencbind -f XQPOE.def -c c -d $(IRAFDIR)/sys/qpoe
        $(CC) $(CFLAGS) xqpoe.c

xtables.o: XTABLES.def
        gencbind -f XTABLES.def -c c -d $(SDASTABDIR)/tables/lib/tbtables
        $(CC) $(CFLAGS) xtables.c

xmwcs.o: XMWCS.def
        gencbind -f XMWCS.def -c c -d $(IRAFDIR)/sys/mwcs
        $(CC) $(CFLAGS) xmwcs.c
```

The test directory contains two test programs and a FITS file that may be used for testing. Again, the makefile will have to be edited similarly to the previous makefile. Lines 1 - 2 define the C compiler. Lines 5 - 6 define path names for the IRAF source directory and the STSDAS tables package. You may also have to modify, delete, or add to the library names in lines 8 - 19 . This list of libraries works for Solaris 2.4, but some of the files (F77, math, or other system files needed by the IRAF libraries) may be in different places on your system. So, some experimentation may be required to get the list of libraries right. One final note: when you run the linker you will get a flood of warning messages about "xercom_" having different sizes in "libsys.a" and other modules. This is a known situation within IRAF and isn't really a problem. The linker makes the right choice (namely the larger one); so, you can just ignore the warnings.

Makefile for compiling the test programs

```
 1:  CC=gcc
 2:  LD=gcc
 3:  CFLAGS=-c -I. -I.. -O
 4:  LFLAGS=-t
 5:  IRAFDIR=/usr/stsci/irafx
 6:  SDASTABDIR=/usr/stsci
 7:  OBJ=../c_iraf_priv.o ../irafinit.o ../irafsppinit.o
 8:  LIBS=   $(SDASTABDIR)/tables/lib/libtbtables.a \
 9:          $(IRAFDIR)/lib/libex.a \
10:          $(IRAFDIR)/lib/libsys.a \
11:          $(IRAFDIR)/lib/libvops.a \
12:          $(IRAFDIR)/unix/hlib/libos.a \
13:          $(IRAFDIR)/unix/hlib/libboot.a \
14:          /opt/SUNWspro/SC3*/lib/libF77.a \
15:          /opt/SUNWspro/SC3*/lib/libM77.a \
16:          /opt/SUNWspro/SC3*/lib/libsunmath.a \
17:          /usr/lib/libdl.so.1 /usr/lib/libelf.so.1 \
18:          /usr/lib/libnsl.so.1 \
19:          -lm -lsocket
20:
```

```
all: copyimg wrtable

clean:
	rm *.o copyimg wrtable

copyimg.o:	copyimg.c
	$(CC) $(CFLAGS) copyimg.c

copyimg:	copyimg.o
	$(LD) -o copyimg $(LFLAGS) copyimg.o $(OBJ) \
		../xclio.o ../ximio.o ../imspp.o $(LIBS)

wrtable.o:	wrtable.c
	$(CC) $(CFLAGS) wrtable.c

wrtable:	wrtable.o
	$(LD) -o wrtable $(LFLAGS) wrtable.o $(OBJ) ../xtables.o $(LIBS)

```