Version 1.0
December 1999

CVOS User's Guide and Reference Manual

SPACE
TELESCOPE
SCIENCE
INSTITUTE

Science Software Group
Science Support Division
3700 San Martin Drive
Baltimore, Maryland 21218

The Science Software Group

Rick White	Group Lead
Perry Greenfield	Programming Supervisor
Howard Bushouse	NICMOS calibration and pipeline, Synthetic Photometry
Ivo Busko	Isophote, Fitting, specview (Java), Graphics support
Ed Colbert	System Administration and Distribution
Michele De La Peña	Python/Tkinter Programming, CVOS, HSTIO, GHRS and FOS pipeline and analysis
Warren Hack	ACS and FOC calibration and analysis, igi, Graphics, Paper Products
Phil Hodge	Table System, FOC and STIS calibration and analysis, Fourier analysis
J.C. Hsu	HSP, FGS, WF/PC, WFPC2 calibration and analysis, Paper Products, Dither,file format conversion tools
Dick Shaw	Exposure Time Calculators, nebular
Hemant Shukla	SEA ETC, WFPC2, JPython
Bernie Simon	Calibration Database, Synthetic Photometry, Table Editor, FITSIO, IRAF System Support
Eric Wyckoff	User Support,Software Testing, STSDAS Web pages, CL Scripts, Python Scripts

Version 1: Written by Michele D. De La Peña

Table of Contents

Preface: Introduction to the CVOS

In This Preface...

About the CVOS...

The Image Reduction and Analysis Facility (IRAF) software environment developed at the National Optical Astronomy Observatory (NOAO) consists of a broad range of functionality organized into libraries. The IRAF system interface, colloquially known as the Virtual Operating System (VOS), provides access to the platform-independent Input/Output libraries which comprise the programming interface. The C Interface to the IRAF Virtual Operating System libraries (CVOS), developed by the Science Software Group at Space Telescope Science Institute (STScI), not only provides a mechanism to generate C interfaces to the VOS libraries, giving access to IRAF functionality from C programs, but also is comprised of pre-built interfaces to the major IRAF libraries.

Using This Manual

This manual is designed to be a comprehensive document not only on the use of the pre-built C interface functions of the CVOS, but also a detailed description on how to build interfaces (also referred to as C bindings in this manual) for the IRAF system. This manual is divided into two main chapters: *CVOS User’s Guide* and *CVOS Reference Manual*. The

CVOS User's Guide provides a general overview of the contents of the CVOS, how to use the pre-built interfaces which have been provided to build C tasks, and shows examples of C tasks. The *CVOS Reference Manual* describes in detail how to build additional C interfaces to libraries or packages.

Typographic Conventions

To help one understand the material in the *CVOS User's Guide and Reference Manual*, a few consistent typographic and design conventions have been employed.

Visual Cues

The following typographic cues are used:

- Figures and examples are often labelled with annotations and arrows to help explain their meaning. These annotations are displayed in **bold sans serif** type.
- **Bold** words identify a STSDAS or IRAF task or package name, a UNIX utility, or to emphasize particular CVOS keywords.
- `Typewriter-like` words identify a file name, directory pathname, system command, or response that is typed or displayed as shown.
- *italic* type indicates a new term or an important point.

Comments

Occasional side comments point out three types of information, each identified by an icon in the left margin.

Tip: No problems...just another way to do something or a suggestion that might make life a bit easier.

Heads Up: Indicates something that is often done incorrectly or that is not obvious.

Warning: You could corrupt data, produce incorrect results, or create some similar problem.

Platform Support

The CVOS is currently supported on all platforms for which the STSDAS system is available. A list of supported platforms can be found on the IRAF web site
`http://iraf.noao.edu.`

An exception to the platform support is that native[1] IRAF C tasks are *not* supported on OpenVMS platforms. Further, it should also be noted that STSDAS/TABLES Version 2.1 (09 October 1999) is the last release to be tested on OpenVMS systems. Approximately, three months after the release of v2.1, all support for OpenVMS will end.

Background Material

In order to use the CVOS effectively, it is necessary to be knowledgeable of the available functionality in the IRAF system. A nice introduction to IRAF and an overview of the functionality available is

- "A Beginner's Guide to Using IRAF, IRAF Version 2.10" by Jeannette Barnes (1993). This document can be obtained on-line from:
  `ftp://iraf.noao.edu/pub/beguide.ps.Z.`

A majority of the IRAF application programs, as well as most of the system, is written in the IRAF Subset PreProcessor (SPP) language which is the native programming language of IRAF. An introduction and overview of SPP can be found in the following references:

- "A Reference Manual for the IRAF Subset Preprocessor Language" by D. Tody (1983):
  `ftp://iraf.noao/edu/iraf/docs/spp.txt.Z`
- "Programmer's Crib Sheet for the IRAF Programming Language" by Douglas Tody (1983):
  `ftp://iraf.noao.edu/iraf/docs/prog_crib.txt.Z.`

A document providing implementation instructions and examples is:

- "An Introductory User's Guide to IRAF SPP Programming" by R. Seaman (1992):
  `ftp://iraf.noao.edu/iraf/docs/sppguide.ps.Z.`

1. See the discussion in the *Building a C Application Task* section of Chapter 1 for details on native versus host-level IRAF C tasks.

The most comprehensive document to date on SPP is:

- “SPP Reference Manual” edited by Zoltan Levay (1992). This document can be obtained on-line from: `http://ra.stsci.edu/Document3.html.`

Obtaining the CVOS

The CVOS is distributed as part of the STSDAS system, which is an IRAF layered package containing the software used for calibrating and analyzing data from the Hubble Space Telescope. The CVOS serves as the underlying support library for the STIS, NICMOS, and ACS calibration pipelines, as well as support for the analysis tasks written in C.

This manual is a comprehensive document describing the CVOS library and its usage. Additional sources of information are:

- ***Web page***: A web page providing links to acquire STSDAS is: `http://ra.stsci.edu/STSDAS.html.`
- ***Help Desk***: Questions about using CVOS or STSDAS can be sent to the STScI Help Desk via E-mail to: `help@stsci.edu.`

Chapter 1:

CVOS User’s Guide

In This Chapter...

The CVOS is comprised of both pre-built interface functions which the programmer can use “out of the box” to begin writing C applications, as well as a semi-automated interface generation mechanism which can be used to create additional bindings to needed IRAF functionality. The discussion in this chapter concentrates on using the pre-built interface functions and the formalism necessary to create C application tasks which use IRAF functionality.

CVOS Pre-built Interfaces and Conventions

As of CVOS Version 3.2.2 (03-September-1999), the CVOS provides pre-built interface functions to a majority of the public routines contained in the IRAF and STSDAS/TABLES libraries or packages listed in Table 1.

Table 1: List of IRAF and STSDAS/TABLES libraries or packages with pre-built C bindings.

Library/Package	*Description*
clio	IRAF command language interaction
curfit	linear least squares curve fitting
gflib	tools for the manipulation of GEIS files
gsurfit	linear least squares surface fitting
iminterp	image interpolation
imio	image access
mwcs	mini-world coordinate system
nlfit	non-linear least squares fitting
qpoe[1]	interface to position order event files
selector	syntax for access to multi-extension FITS files
surfit	surface fitting
synphot	synthetic photometry
tables	tools for manipulating STSDAS, FITS, and text tables
vops	vector (array) operations
xtools[2]	miscellaneous tools

1. The **qpoe** interface files contain some manually coded wrapper interface fuc-tions needed to support specific data conversions.
2. Only a subset of tasks in the **xtools** package have C interfaces.

File and Interface Naming Conventions

Some discussion of the nomenclature used by the CVOS is necessary in order to understand the contents of the CVOS source directories. The majority of the CVOS source files are located in `stsdas$lib/cvos` and associated subdirectories; the CVOS header files are located in `stsdas$lib`[1]. An in-depth discussion and description of the CVOS files is found in the *CVOS Reference Manual*.

The naming convention adopted for the CVOS files which correspond to IRAF or STSDAS/TABLES libraries and packages is to prepend an "x" to the library or package name. For example, the CVOS header file for the IRAF **imio** library is `ximio.h`. The CVOS header files contain the function prototypes for all of the public functions which have a C interface for a particular library or package. In order to build a C task properly, it is necessary to include the appropriate CVOS header file in the application source code for any C interface functions used. The C source file associated with a CVOS header file for the **imio** library is named `ximio.c`. The source files contain the definitions for the functions; the definitions handle any necessary data type conversions between C and IRAF, call the actual IRAF functions, and check error conditions upon return from IRAF.

The naming convention for the C interface functions themselves which are contained in the CVOS library/package files is to prepend a "c_" to the IRAF function name. For example, in IRAF the function to open an image file and obtain the data contents is **immap**; the corresponding CVOS function is **c_immap**. In an instance where this convention would cause a conflict with any existing IRAF functions, a "c_x" is prepended to the IRAF function name. Thus far, only one conflict has been found such that the IRAF **c_imaccess** becomes **c_ximaccess** in the CVOS.

Data Types and I/O Nomenclature

In order to use the CVOS to create C tasks, the application source code must include the `<c_iraf.h>` file which contains critical declarations and definitions needed by the C interface functions and source programs. In order to provide a better correspondence between C and IRAF data types, `<c_iraf.h>` defines the three additional types of `Bool`, `IRAFPointer`, and `struct IRAFComplex` which can be referred to as simply `Complex`. `<c_iraf.h>` also defines C symbolic names which correspond to the IRAF symbolic names representing data type

1. `stsdaslib` is set to the IRAF environment variable `stsdas$lib` in the **stsdas** `zzsetenv.def` file.

codes as listed in Table 2. The data type codes are an enumerated list of

Table 2: Data Type Code correspondence between C and IRAF.

CVOS Data Type Codes	*IRAF Data Type Codes*
IRAF_BOOL	TY_BOOL
IRAF_CHAR	TY_CHAR
IRAF_SHORT	TY_SHORT
IRAF_INT	TY_INT
IRAF_LONG	TY_LONG
IRAF_REAL	TY_REAL
IRAF_DOUBLE	TY_DOUBLE
IRAF_COMPLEX	TY_COMPLEX
IRAF_POINTER	TY_POINTER
IRAF_STRUCT	TY_STRUCT
IRAF_USHORT	TY_USHORT
IRAF_UBYTE	TY_UBYTE

`IRAFType`. The IRAF symbolic names are defined in the `iraf$unix/hlib/iraf.h` file.

Data type codes are typically used for dynamic memory allocation where it is necessary to know the number of bytes each value occupies. An example of the use of the CVOS data type codes would be in the definition of a new table column such as

```
c_tbcdef1(table->tp, &(table->back), "BACKGROUND",
"Counts/s", "", IRAF_REAL, table->array_size);
```

This line of C code uses the C interface function, `c_tbcdef1`, to define a single column in a table `table->tp` where the new column `&(table->back)` is called BACKGROUND. The BACKGROUND column has units of `Counts/s`, no specified print format, contains data of type IRAF_REAL, and has `table->array_size` number of elements.

In C the number of bytes associated with the `int` and `long` data types is platform dependent. While these types often represent the same number of bytes, this is not always true as is the case for the Compaq Tru64 platform. Although IRAF has both `int` and `long` defined, these data types *always* represent the same number of bytes in the IRAF system. In order to minimize incompatibilities between C and IRAF, the CVOS uses only the `int` data type in all interfaces. The programmer is urged to use only the `int` data type *for all variables in C programs that are to be passed as arguments* to CVOS interface functions.

File input/output access modes are also defined in the `<c_iraf.h>` file as an enumerated list of `IRAFIOMode` and are listed in Table 3.

Table 3: File I/O correspondence between C and IRAF

C File I/O Modes	***IRAF File I/O Modes***
IRAF_NOMODE	
IRAF_READ_ONLY	READ_ONLY
IRAF_READ_WRITE	READ_WRITE
IRAF_WRITE_ONLY	WRITE_ONLY
IRAF_APPEND	APPEND
IRAF_NEW_FILE	NEW_FILE[1]
IRAF_TEMP_FILE	TEMP_FILE
IRAF_NEW_COPY	NEW_COPY

1. NEW_IMAGE, NEW_STRUCT, and NEW_TAPE are all synonyms for NEW_FILE.

An example where a file I/O mode would be used is in the reading of data from an image which would use the CVOS interface function, **c_immap**:

```
IRAFPointer c_immap(char *, int, IRAFPointer);
```

and the actual line of code in a C task would look like

```
fileDescriptor = c_immap(filename, IRAF_READ_ONLY, 0)2;
```

where the file `filename` contains an image which is mapped to an IRAF image structure in read-only access mode.

2. The null value for the IRAFPointer data type should be zero, and not NULL in C codes.

Building a C Application Task

Preprocessor Include Files

In addition to the `<c_iraf.h>` file, any C application source code must also include any library header files (e.g., `<ximio.h>`) which contain the prototypes for functions used in the application code. As a precaution, the `<c_iraf.h>` is automatically included by all of the library/package header files included in the C application source. However, it is best to include the `<c_iraf.h>` file explicitly and *before* any other CVOS header files in the application source. The top of a C source code file should resemble the following:

```
/* Include CVOS header files */
# include <c_iraf.h>
# include <ximio.h>
```

Mkpkg versus Make

Mkpkg is a portable IRAF utility for building or updating a package or library. Since it is implemented as a foreign task, **mkpkg** can be invoked from within the IRAF environment or from the host system. The advantage of using **mkpkg** is the IRAF group has already resolved the portability issues associated with building and maintaining code. In the event the C source code needs special compilation switches or additional external libraries, these can easily be accommodated in the **mkpkg** syntax. Users of the CVOS are strongly urged to use **mkpkg** to maintain CVOS applications. The **mkpkg** utility determines how to build the executable or library from the `mkpkg` file located in the source directory. To obtain documentation on **mkpkg**, one should type `help mkpkg` in the IRAF environment to invoke the IRAF help pages. Alternatively, one can use the Web-based IRAF help system

`http://iraf.noao.edu/iraf-help.html`

or the Web-based help system developed at STScI

`http://ra.stsci.edu/gethelp/HelpSys.html.`

Make is a UNIX-based utility for maintaining and updating programs and files. Since **make** is platform-dependent, the specific capabilities and option switches vary between platforms. If **make** is used to maintain the CVOS application, it is incumbent on the programmer to determine the location of necessary libraries on the system. The programmer should be aware the locations of libraries can (and do) change with upgrades to the system. The **make** utility determines how to build executables or libraries typically from the `makefile`[3] located in the source directory. To obtain documentation on **make**, one should invoke the manual pages on the specific platform.

Host-level versus Native IRAF Tasks

Any additional code which must be incorporated into the C application source, as well as compilation switches which need to be set, are dependent upon the way the executable is to be built -- either as a host-level or as a native IRAF task. While it is possible to set up the source code and `mkpkg` files in such a manner for the code to be compiled in either mode, this has not proven to be very useful in practice.

Host-level C tasks are programs similar to any pure C program which have the added capability to take advantage of the functionality in the IRAF libraries. In this way, the IRAF libraries are no different than any other public library accessible to C. Host-level tasks are designed to be executed at the host-level with command line arguments, and therefore, they cannot be run directly from the IRAF CL. Since host-level tasks are effectively independent from the IRAF environment, they do not have access to IRAF environment variables. Any needed environment variables (e.g., shortcuts for directory pathnames) *must be set at the host-level* by the user.[4]

In contrast, a C program compiled as a native IRAF task can take advantage of all the capabilities of the IRAF system. Not only do native IRAF tasks have access to the IRAF library functionality, but they can be run directly from the CL. Input/output and other information used to customize the functionality of a native task are handled by task parameters as is done with IRAF SPP tasks.

The following contrasts the attributes of a C task compiled as a host-level and as a native IRAF task.

- Host-level tasks
  - use the Standard C library.
  - can use the IRAF libraries and packages.
  - are run from the host-level with command line arguments.
  - lose the convenience of the IRAF CL parameter handling capabilities.
  - do not know about IRAF environment variables. Any environment variables must be set at the host-level.

3. The actual source file (e.g., `makefile`) used to build the final target file can have several system-dependent names.

4. Defining an environment variable at the host-level is dependent upon the specific shell interpreter in use.

- Native IRAF tasks
  - use the IRAF C library.
  - can use the IRAF libraries and packages.
  - are run from the CL.
  - maintain use of the IRAF CL parameter handling capabilities.
  - know about defined IRAF environment variables.
  - can use a wrapper routine which handles error exits gracefully without hanging the IRAF CL.
  - should not use calls to exit() or _exit().
  - should not use the return statement to return a value from the top level routine.

The following two sections present a very simple C program strictly for the purpose of illustrating and contrasting attributes of being written and compiled as either a host-level or native IRAF task. The presentation includes the C source code, the corresponding `mkpkg` file, an example of running the task, and any other files or information needed to execute the task.

Host-level Task

Example 1 is a simple example of a C task which opens a FITS image and obtains some information regarding the image. This example contains all of the critical components needed for the C program to compile as a host-level task. A barebones `mkpkg` file shows how to compile and link the source code in Example 2. CVOS related files and functions and other important items are represented in **bold** characters in both the C source file and the `mkpkg` file. Finally, the program is compiled and executed from the host command line[5] in Example 3.

As one examines Example 1, keep in mind the necessary CVOS components for the C task to be compiled as a host-level task; the components are summarized here.

- The C source code must include:
  - **<c_iraf.h>**,
  - any necessary header files which contain the prototypes for functionality used in the program (e.g., **<ximio.h>**, **<xclio.h>**, etc.),

5. C tasks can be built using the IRAF **mkpkg** mechanism or UNIX **make** files. Since **mkpkg** is a system independent utility, it is strongly encouraged that tasks be built in this manner.

  - and the IRAF VOS must be initialized by calling **c_irafinit ()**.
- The mkpkg file must include:
  - an XFLAGS **-Inolibc** flag,
  - an XFLAGS **-p stsdas**,[6]
  - an LFLAGS **-H** flag,
  - an LFLAGS **-p stsdas**,[6]
  - and the link must include the CVOS library, **-lcvos**.
- Environment setup:
  - If any environment variables are needed, they need to be defined at the host-level before the task can be run correctly.

6. There exists an alternative to having XFLAGS and LFLAGS **-p stsdas** switches present in the `mkpkg` file. An alternative is discussed in the paragraphs following the host-level IRAF C task `mkpkg` example.

Example 1: Host-level IRAF C task source file, openimage.c.

```
/* C standard header files */
# include <stdio.h>
# include <string.h>

/* CVOS header files */
# include <c_iraf.h>
# include <ximio.h>

# define SZ_FNAME 255

/*
** Simple C program written which illustrates the necessary components
** for a host-level task.
*/

int main(int argc, char **argv) {
    IRAFPointer in;
    IRAFType pixtype;
    int ndim, dim1, dim2;
    int linevector[] = {1, 1, 1, 1, 1, 1, 1};
    char input[SZ_FNAME+1];

    /* Need to initialize the IRAF libraries *
     * for host-level tasks                  */
    c_irafinit (argc, argv);

    if (argc < 2) {
        printf (“syntax:  openimage input\n”);
        exit (1);
    }
    strcpy (input, argv[1]);

    /* Open the input image */
    in = c_immap (input, IRAF_READ_ONLY, 0);

    /* Check the input image was opened without error */
    if (c_iraferr()) {
        printf (“IRAF error code%d\nIRAF message: %s\n”,
                c_iraferr(), c_iraferrmsg());
        exit (1);
    }

    /* Get the image dimensions */
    ndim = c_imgndim (in);
    if (ndim != 2) {
        printf (“Sorry! This example only works for two dimensions.\n”);
        exit (1);
    }

    /* Get the size of each dimension */
    dim1 = c_imglen (in, 1);
    dim2 = c_imglen (in, 2);
    /* ...and the pixel type */
    pixtype = (IRAFType) c_imgtypepix (in);
    printf (“Input:  %d dimensions -- dim1 = %d  dim2 = %d of type %d\n”,
            ndim, dim1, dim2, pixtype);

    /* Close the image and end */
    c_imunmap (in);
    return (0);
}
```

All CVOS tasks need this header file.

C bindings from this library are used in this routine (e.g., c_immap).

Data type defined in <c_iraf.h>.

The IRAFType(s) are listed in Table 2.

Critical to include for host-level tasks.

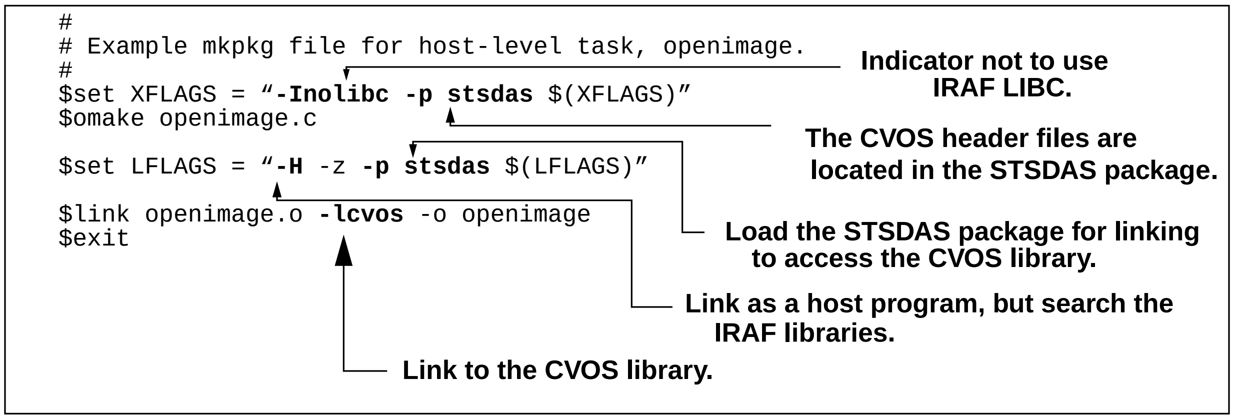
File I/O mode.

Example of a C interface function call.

Basic error handling. Functions are declared in <c_iraf.h>.

As noted, the program in Example 1 is quite simplistic, but it serves to illustrate the basics of how a C program would access IRAF functionality. Previously, it has been briefly noted the C interface definitions are responsible for performing several functions, one of which is to check error conditions upon return from the underlying IRAF function. The lines of code in Example 1 noted as basic error handling show how the error checking can be accessed and used. In theory, this type of checking should be done after *every* call to a C interface routine. In practice, there is a better way to handle error checking. A full discussion of this issue is deferred to the *Error Handling* section of this document.

Example 2: Host-level IRAF C task mkpkg file.



```
#
# Example mkpkg file for host-level task, openimage.
#
$set XFLAGS = "-Inolibc -p stsdas $(XFLAGS)"
$omake openimage.c

$set LFLAGS = "-H -z -p stsdas $(LFLAGS)"

$link openimage.o -lcvos -o openimage
$exit
```

Indicator not to use IRAF LIBC.
The CVOS header files are located in the STSDAS package.
Load the STSDAS package for linking to access the CVOS library.
Link as a host program, but search the IRAF libraries.
Link to the CVOS library.

The `mkpkg` file depicted in Example 2 uses some of the new **xc** compiler command line flags implemented in IRAF 2.11 specifically to improve support for multi-language and host software development. The XFLAGS **-Inolibc** flag formally indicates that one does not want to use the header files located in `iraf$unix/hlib/libc`. This, in effect, disables the use of the IRAF version of the C library. In this example, the host system files will be used instead. The XFLAGS **-p stsdas** is needed in order for the compiler to find the CVOS include files which are located in `stsdaslib$`. The LFLAGS **-H** flag indicates the program should be linked as a host-level program, but the IRAF libraries should still be searched to resolve symbol references. The LFLAGS **-p stsdas** indicates the STSDAS layered package should be loaded; this is necessary since this package includes the definition of the CVOS (i.e., the CVOS library). Finally, the **-lcvos** tells the linker to include the CVOS library, in addition to the standard libraries which are included, to resolve symbol references.

In this example, the **-p stsdas** switch is included with both the XFLAGS and LFLAGS as the safest way to ensure the STSDAS package is searched not only for the CVOS header files during the compilation, but also for the CVOS library during the link stage when building the task. An alternative to including the **-p stsdas** switch with both the XFLAGS and LFLAGS in the `mkpkg` file is to include the **-p stsdas** as a command line argument to the **mkpkg** task,

```
enkidu> mkpkg -p stsdas
```

where the programmer must remember to use the command line argument when invoking **mkpkg**. If the **-p stsdas** is included on the command line as well as in the `mkpkg` file itself, it is not a problem.

Example 3: Compilation and execution of a host-level IRAF C task, openimage.

```
    enkidu> mkpkg
    xc -Inolibc -p stsdas -c -DSYSV -DSOLARIS -/libmil openimage.c
    xc -H -z -p stsdas -/Bstatic openimage.o -lcvos -o openimage
    enkidu> openimage "o3tt03040_raw.fits[1]"
    Input:  2 dimensions -- dim1 = 1062  dim2 = 1044 of type 11
```

Example 3 shows the `openimage` executable being built on the host system with the `mkpkg` file in Example 2 and then being run at the command line. This particular example shows the task being built on a system running Solaris; the **xc** lines echoed here may be different for other platforms. See Appendix A for the Host-Level task checklist.

Native IRAF Task

Using the same algorithm as depicted in Example 1, the actual program has been rewritten as a native IRAF task; the results are shown in Example 4. CVOS related files and functions and other important items are represented in **bold** characters in both the C source file and the `mkpkg` file. The presentation includes the C source code (Example 4), the corresponding `mkpkg` file (Example 5), a simple parameter file (Example 6), and an example of compiling and then running the task from the CL (Example 7). A parameter file is a way to specify the attributes of input/output values which are read/written by a task from the CL. If there are parameters associated with a native IRAF task, it is necessary to have a parameter file associated with the task. Please see the *SPP Reference Manual* for details.

The necessary CVOS components for the C task to be compiled as a native IRAF task are summarized here.

- The C source code must include:
    - **<c_iraf.h>**,
    - any necessary header files which contain the prototypes for functionality used in the program (e.g., **<ximio.h>**, **<xclio.h>**, etc.),
    - instead of main(), use the **IRAFTASK(taskname)** macro. This macro performs several functions and invokes the IRAF initialization routine

for native tasks automatically. The programmer does *not* need to make any explicit calls to an initialization routine.

- The C source code must ***not*** include:
  - any calls to exit() or _exit(),
  - the top level routine should not use the return statement to pass a value to the CL. In the top level routine a return statement with no value can be used, or no return statement needs to be present at all. A return statement can be used to pass values from subroutines to the top level routine.
- The mkpkg file must include:
  - an XFLAGS **-Inolibc** flag,
  - an XFLAGS flag to use the CVOS version of **<stdio.h>** in **stsdaslib$cvos/irafstdio**,
  - an XFLAGS **-p stsdas**,
  - an LFLAGS **-H** flag,
  - an LFLAGS **-p stsdas**,
  - the link must include the CVOS library, **-lcvos**,
  - and the link must include the C library, **-lc** (this is the default on some platforms, but it does not hurt to include the library explicitly.

Example 4: Native IRAF C task source file, openimage.c.

```
/* C standard header files */
# include <stdio.h>

/* CVOS header files */
# include <c_iraf.h>
# include <xclio.h>
# include <ximio.h>

# define SZ_FNAME 255

/* C program which illustrates the necessary components for a native IRAF task. */

/* Wrapper which serves as the main entry/exit routine.  This wrapper is **
** needed to handle error exits which might hang the IRAF CL.            */
IRAFTASK (openimage) {
                                    Special macro which must be used in native IRAF C tasks.
    /* Declare a local variable and the function prototype */
    int i;
    int openIt (void);

    i = openIt ();                     No return(), exit(), or _exit() statement.
}

/* Real work routine */
int openIt (void) {

    IRAFPointer in;
    IRAFType pixtype;
    int ndim, dim1, dim2;
    int linevector[] = {1, 1, 1, 1, 1, 1, 1};
    char input[SZ_FNAME+1];

    c_clgstr (“input”, input, SZ_FNAME);     C interface function call to read a
                                                 string from the CL.
    /* Open the input image */
    in = c_immap (input, IRAF_READ_ONLY, 0);

    /* Check the input image was opened without error */
    if (c_iraferr()) {
        printf (“IRAF error code: %d\nIRAF message: %s\n”,
                c_iraferr(), c_iraferrmsg());
        return (1);
    }                                  Never use exit() or _exit(). OK to use return with a value
                                             here to pass information to the top level.
    /* Get the image dimensions */
    ndim = c_imgndim (in);
    if (ndim != 2) {
        printf (“Sorry! This example only works for two dimensions.\n”);
        return (1);
    }

    /* Get the size of each dimension */
    dim1 = c_imglen (in, 1);
    dim2 = c_imglen (in, 2);

    /* Get the pixel type */
    pixtype = (IRAFType) c_imgtypepix (in);
    printf (“Input:  %d dimensions -- dim1 = %d  dim2 = %d of type %d\n”,
            ndim, dim1, dim2, pixtype);

    /* Close the image and return */
    c_imunmap (in);
    return (0);
}
```

The native IRAF task in Example 4 differs from the host-level task in Example 1 in several fundamental respects. The main entry point is not through a “main” routine

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

but rather through the use of

```
IRAFTASK (taskname).
```

IRAFTASK is a macro defined in `<c_iraf.h>;` **IRAFTASK** itself has a **main()** and includes a call to **irafcmain ()** which is the initialization routine for a native IRAF task. This is in contrast to the initialization routine, **c_irafinit()**, which is used by host-level tasks and *must* be called explicitly by the programmer. The programmer does *not* need to call an IRAF initialization routine explicitly when writing a native IRAF C task.

Note there is no return type for the **IRAFTASK** macro, so there is no **return()** statement in the main entry routine.[7] In this example, the native IRAF task routine obtains a single string input value from the CL using the C interface routine, **c_clgstr()**. Since this routine is reading input from the CL, it is necessary to have a parameter file for this task. The parameter file is described in Example 6.

Perhaps the most fundamental difference between a native IRAF task and a host-level task which are both written in C is the need for the native IRAF task to use a main entry point routine, as defined by the **IRAFTASK(taskname)** macro, as a wrapper for the entire algorithm. Since no return value is expected to be passed from the native task to the IRAF CL, it is critical that the entire algorithm not use **exit()** or **_exit()** to terminate the task upon detection of an error. Both of these functions will cause an immediate termination of the C task, and can (and probably will) cause problems in the CL. The problems are manifested as corruption and/or hanging of the CL. The task should use the **return()** statement when errors are detected in any lower-level subroutines in order to pass control back to the top level routine for a clean termination.

Although Example 4 has been written such that the majority of the task functionality is contained in the **openIt()** subroutine, it is not necessary to create a C native IRAF task in this manner. The important issue is simply that **IRAFTASK(taskname)** replaces **int main (argc, argv)**, and the task should not exit to the CL (via **return()**, **exit()**, or **_exit()**) with a returned value.

The programmer should note **taskname** in **IRAFTASK(taskname)** must not be surrounded by quotes. If quotes are included accidentally, excessive errors will be generated when compiling the task. Also, the

7. Actually, there can be a **return** statement from the main entry routine as long as it does *not* return a value.

length of **taskname** is not restricted to six characters and may include use of the underscore character. *All* the characters of **taskname** are significant. This is in contrast to SPP task names which are mapped to Fortran identifiers that conform to Fortran 66 standards. The SPP to Fortran 66 mapping removes any underscores and only the first five characters and the last character of the task name are significant and are used for the final identifier.

The native task `mkpkg` file shown in Example 5 differs from the host-level `mkpkg` file in two ways: use of **-Istsdaslib$cvos/irafstdio** for the compilation and the **-lc** flag for the link. In this case, the **-Inolibc** flag is working in conjunction with the **-Istsdaslib$cvos/irafstdio** flag. The **-Inolibc** indicates that the host system C header files should be used, except for `<stdio.h>` which is found in the directory as indicated by **-Istsdaslib$cvos/irafstdio**. This is a customized version of `<stdio.h>` which is needed to support native IRAF tasks. The **-lc** indicates the Standard C library must be linked to the compiled object.

Example 5: Native IRAF C task mkpkg file.

```
#
# Example mkpkg file for native IRAF C task, openimage.
#
$set XFLAGS = “-Inolibc ‘-Istsdaslib$cvos/irafstdio’ -p stsdas $(XFLAGS)”
$omake openimage.c
                                                    A special version of <stdio.h> to support
$set LFLAGS = “-H -z -p stsdas $(LFLAGS)”                native tasks is stored here.

$link openimage.o -lcvos -lc -o openimage
$exit                                                 Link in the Standard C library.
```

Running the task from within the IRAF CL requires one to define the new task and set the parameters as would be done with any IRAF task. The parameter file for the task should be located in the same directory as the task executable. The parameter file should also have the same rootname as the associated task, appended with a ".par". In this example the `openimage` executable is located in the directory `/mydir/iraf/`; the associated parameter file is `openimage.par`. Example 6 illustrates a very simple parameter file which only reads a string from the IRAF CL. Please see the *SPP Reference Manual* for details on parameter files.

Example 6: Native IRAF C task parameter file, openimage.par.

```
input,s,a,””,,,”Input image name”
```

Example 7 shows how the task executable is built. The **mkpkg** command can be issued at the host-level or from within the IRAF CL. Once the executable is built, a typical user will run the task from within the CL as depicted. It is necessary first to define the new task via the **task** command. This example then invokes **lpar** on the taskname to verify the parameter file is accessible, and lastly, the task is executed. See Appendix A for the Native IRAF task checklist.

Example 7: Compilation, definition, and execution of a Native IRAF C task.

```
enkidu> mkpkg
xc -Inolibc ‘-Istsdaslib$cvos/irafstdio’ -p stsdas -c -DSYSV -DSOLARIS -/libmil
openimage.c
xc -H -z -p stsdas -/Bstatic openimage.o -lcvos -lc -o openimage
enkidu> cl
cl> task openimage = /mydir/iraf/openimage     ◄── Define a new task.
cl> lpar openimage
        input = ““       Input image name
        (mode = “ql”)
cl> openimage                ◄──────────── Execute the task.
Input image name: o3tt03040_raw.fits[1]
Input:  2 dimensions -- dim1 = 1062  dim2 = 1044 of type 11
cl>
```

Error Handling

As seen in Example 1, after invoking any C interface function, the IRAF error status can be checked by using the **c_iraferr()** and **c_iraferrmsg()** functions; these functions have no parameters. **C_iraferr()** returns the IRAF error number; **c_iraferrmsg()** returns the text string associated with the error number. If no error has occurred, **c_iraferr()** returns zero, and c**_iraferrmsg()** is a null string. An example of these functions in use is illustrated by the following code snippet from Example 1.

Example 8: Code snippet illustrating the use of the basic error handling functions.

```
/* Check the input image was opened without error */
if (c_iraferr()) {
    printf (“IRAF error code%d\nIRAF message: %s\n”,
    c_iraferr(), c_iraferrmsg());
    exit (1);
}
```

If the input image did not exist, the following message would be generated when using the host-level task, openimage:

```
enkidu> openimage “ack.fits”
IRAF error code 827
IRAF error message: Cannot open image (ack.fits)
```

As part of the implementation for each CVOS interface function, any previously set error codes and corresponding error messages are cleared before the underlying SPP functions are invoked. Consequently, it is not safe to call a series of C interface functions and then check **c_iraferr()** at the end of the series. If **c_iraferr()** is to be useful, it must be invoked after every C interface function call.

A practical alternative to inserting error checking code after every invocation of a C interface *in a host-level task* is to use the CVOS **c_pusherr()** function. The **c_pusherr()** is a mechanism for installing global customized error handlers. The advantage of the global error handler is the error status of each C interface function used in the source need not be checked after each invocation. Rather, if an error occurs and a handler function has been installed, the handler function will be called automatically upon detection of the error.

The **c_pusherr()** function works in conjunction with an error handler stack which can accommodate up to thirty-two entries. This gives the programmer the ability to define a series of error functions to handle special situations. The functions are installed by pushing them onto the error handler stack via **c_pusherr()**; **c_pusherr()** takes one parameter which is a pointer to a function. Due to the nature of a stack, only the last function pushed onto the stack is "active". It is this "active" error function that will be called automatically when an error is detected. The programmer can manipulate the action associated with any detected error by pushing and popping, via a stack pop function **c_poperr()**, different handler functions onto the stack; the **c_poperr()** function has no parameters. In order to prevent temporarily any particular error handler function from being called, a zero can be pushed onto the stack. Popping the zero will then restore the previous handler function. Example 9 illustrates the use of an error handler. This illustration is based upon Example 1 which is the source code for a host-level task; the code has been modified to incorporate the use of a global error handler and has been slightly abridged in order to fit the example on a single page. Please note the global error handler can *only be used with host-level tasks.* Since native IRAF tasks can only use the **return()** statement to terminate execution of a subroutine and transfer control back to the top level routine and no value can be passed to the IRAF CL, it is not possible to use the global error handler for native IRAF C tasks.

Example 9: Host-level IRAF C task using a global error handler.

```
/* C standard header files */
# include <stdio.h>
# include <string.h>

/* CVOS header files */
# include <c_iraf.h>
# include <ximio.h>
                                   Define the global error handler function.
# define SZ_FNAME 255
/*
** Define a global error handler for this routine.
*/
static void detect_iraferr () {
    fprintf (stderr, "\nIRAF error %d: %s\n", c_iraferr(), c_iraferrmsg());
    fflush (stderr);
    exit (1);
}
/*
** Simple C program written which illustrates the
** necessary components for a host-level task.
*/
int main(int argc, char **argv) {
    IRAFPointer in;
    IRAFType pixtype;
    int ndim, dim1, dim2;
    int linevector[] = {1, 1, 1, 1, 1, 1, 1};
    char input[SZ_FNAME+1];

    /* Need to initialize the IRAF libraries *
     * for host-level tasks                  */
    c_irafinit (argc, argv);

    /* Push a function onto the error handler stack. */
    c_pusherr (detect_iraferr);
                                         Install the handler function.
    if (argc < 2) {
        printf ("syntax:  globalhandler input\n");
        exit (1);
    }
    strcpy (input, argv[1]);

    /* Open the input image */
    in = c_immap (input, IRAF_READ_ONLY, 0);
                                          No longer need to check c_iraferr().
    /* Get the size of each dimension */
    dim1 = c_imglen (in, 1);
    dim2 = c_imglen (in, 2);
    /* ...and the pixel type */
    pixtype = (IRAFType) c_imgtypepix (in);
    printf ("Input dimensions -- dim1 = %d  dim2 = %d of type %d\n",
            dim1, dim2, pixtype);

    /* Close the image and remove the function from the error handler stack */
    c_imunmap (in);
    c_poperr ();                   Remove the error handler function from the stack.

    return (0);
}
```

Exception Handling

While the handling of exceptions is done automatically for the programmer by the system, a short digression at this time can serve to clear up misconceptions. For this discussion, *exceptions* should be differentiated from *errors* in that exceptions are asynchronous problems which arise from situations in the task that cannot be reasonably anticipated (e.g., ^C, bus errors, segmentation violations, divide by zero) and are detected by the hardware as illegal conditions. Because exceptions are detected by the hardware, the task does not necessarily know its state at the time of the exception. This makes exception handling more complex than error handling. When an exception is detected, the normal execution of the task is disrupted, and the flow of control is transferred out of the task and to system-level procedures which have been specifically designed to handle exceptions. The default exception handling procedure performs several clean-up functions and exits.

In contrast, basic errors represent situations which could be checked and handled by code in the task itself (e.g., cannot find or open a file, no memory to allocate a pointer). Since an error condition is detected by the task, the state of the task is known at the time of the error. This situation is more easily handled by the system or the programmer.

For C programs compiled as native IRAF tasks, the IRAF initialization step which is implicitly invoked by the **IRAFTASK** macro posts a default exception handler. This situation is identical to what would be done for a task written in SPP, the native language of IRAF. In the event an exception occurs, the exception will be handled by the IRAF system default handler. It is possible for the programmer to post alternate exception handlers for native IRAF tasks, but this issue is beyond the scope of this document. Please see the *SPP Reference Manual* for further details.

For C programs compiled as host-level IRAF tasks, exceptions are handled by CVOS routines in a straightforward fashion. When an exception is detected, the associated IRAF error messages are posted, all output buffers are flushed, and the task exits to the host environment.

This discussion is presented so that programmers do not confuse the CVOS error handling mechanism described in the previous section with the handling of exceptions. The CVOS error handler appears to mimic an exception handler by seeming to detect error conditions in an omniscient fashion. However, this is not the situation. The function on the top of the error handler stack is explicitly invoked by the error checking software imbedded in each of the C interface functions.

Summary of CVOS Utility Functions

Most of the CVOS public utility functions have been mentioned throughout this document and used in examples. These functions are declared in the `<c_iraf.h>` file found in the `stsdaslib$` directory; the programmer is encouraged to examine the contents of this file. The utility function prototypes and native task macro are summarized here for convenience.

- Compilation mode functions:
  - **IRAFTASK (taskname) -** This macro is used for source compiled as native IRAF C tasks. Taskname is the logical IRAF task name.
  - **void c_irafinit (int argc, char **argv) -** A call to this function must be included for all host-level IRAF C tasks to invoke IRAF initialization. The argc and argv variables are not actually used.
- Error handler stack manipulation functions:
  - **int c_pusherr (c_IRAFErrHandler) -** This routine pushes the named function onto the top of the error handler stack. The return value is the number of functions currently on the stack or -1 if the maximum number of error functions has already been reached.
  - **typedef void (*c_IRAFErrHandler) (void) -** This defines c_IRAFErrHandler as a pointer to a function which has no parameters and does not return any values.
  - **int c_poperr (void) -** This routine removes the function residing on the top of the error handler stack. The return value is the remaining number of functions on the stack or -1 if there were no functions to remove from the stack.
- Error message functions:
  - **int c_iraferr (void) -** This routine returns the IRAF error code or zero for no error.
  - **char * c_iraferrmsg (void) -** This routine returns the IRAF error string associated with the IRAF error code. If there is no error condition, the empty string is returned.
  - **void clear_cvoserr (void) -** This routine sets the IRAF error code to zero and the IRAF error string to the empty string. The IRAF error code and error string are set, if necessary, by a private CVOS function imbedded in the C binding routines.

An Additional Example

An example of a host-level IRAF C task which creates a simple table is depicted in Example 10. As in the previous examples, CVOS related files and functions, as well as other notable issues, are represented in **bold** characters. The `mkpkg` file is shown in Example 11. The contents of the resultant table, `wrtableTest.tab`, are displayed in Example 12.

Example 10: Host-level IRAF C task to work with tables, wrtable.c.

```
/* C standard header files */
# include <stdlib.h>
# include <stdio.h>

/* CVOS header files */
# include <c_iraf.h>     <--------  All CVOS tasks need this header file.
# include <xtables.h>   <---------  C bindings from this library are used in
                                     this routine.
/* Define the number of columns */
# define NCOLS 7

/*
 * Define the global error handler for this routine
 */
void iraferr () {
    printf (“IRAF error code %d\nIRAF message: %s\n”,     Define the global
            c_iraferr (), c_iraferrmsg ());                   error handler.
    exit (1);
}

/*
** Simple C program which uses tools from the tables package
** in a host-level task.
*/

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

    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       = 0;
    int    disk;
    char   *region;
    char   *plate;
    double ra;
    double dec;
    double epoch;
    char   *survey;
    IRAFPointer tab;           /* pointer to table struct    */
    IRAFPointer col[NCOLS];    /* pointers to column info    */

    /* Need to initialize the IRAF libraries *
     * for host-level tasks                  */
    c_irafinit (argc, argv);  <-----  Critical for host-level tasks.
```

Example 10 (Continued): Continuation of wrtable.c.

```
    /* Install a global error handler */
       c_pusherr (iraferr);       ◄──  Install the error handler on the stack.

       /* Open the output table */                                    Create the
       tab = c_tbtopn ("wrtableTest.tab", IRAF_NEW_FILE, 0);     ◄─┘   table descriptor.

       /* 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);    ◄──────────────  Create the table.

       /* Add a history record */
       c_tbhadt (tab, "history", "Simple test program to create a table.");

       /* Add a few rows in a simple manner */
       row++;
       region = "S256";
       plate  = "000Y";
       ra     = 105.37155;
       dec    = -45.07291;
       epoch  = 1980.122;
       survey = "UK";
       disk   = 18;

       c_tbeptt (tab, col[REGION], row, region);     ◄──────   Write a single element
       c_tbeptt (tab, col[PLATE], row, plate);                   to a table.
       c_tbeptd (tab, col[RA], row, ra);
       c_tbeptd (tab, col[DEC], row, dec);
       c_tbeptd (tab, col[EPOCH], row, epoch);
       c_tbeptt (tab, col[SURVEY], row, survey);
       c_tbepti (tab, col[DISK], row, disk);

       row++;
       region = "S734";
       plate  = "006Q";
       ra     = 275.68950;
       dec    = -9.97406;
       epoch  = 1978.647;
       survey = "UK";
       disk   = 50;

       c_tbeptt (tab, col[REGION], row, region);
       c_tbeptt (tab, col[PLATE], row, plate);
       c_tbeptd (tab, col[RA], row, ra);
       c_tbeptd (tab, col[DEC], row, dec);
       c_tbeptd (tab, col[EPOCH], row, epoch);
       c_tbeptt (tab, col[SURVEY], row, survey);
       c_tbepti (tab, col[DISK], row, disk);

       /* Close the table and clean up */
       c_tbtclo (tab);   ◄─────────  Close the output table.

       printf ("Created and closed the tables\n");

       /* Remove the function from the error handler stack */
       c_poperr ();    ◄────┐
       return (0);          │  Remove the error handler function from the stack.
   }
```

The `mkpkg` file is similar to the file in Example 2. However, this example is also using the TBTABLES library. The LFLAGS **-p stsdas -p tables** indicates both the STSDAS and TABLES layered packages should be loaded. This is necessary since the CVOS library is located in the STSDAS package and the TBTABLES library is located in the TABLES package. The TBTABLES library must be included in the link stage to resolve symbol references.

Example 11: Host-level C task mkpkg file using the **tables** library.

```
#
# Example mkpkg file for host-level task, openimage.
#
$set XFLAGS = “-Inolibc -p stsdas $(XFLAGS)”
$omake wrtable.c

$set LFLAGS = “-H -z -p stsdas -p tables $(LFLAGS)”

$link wrtable.o -lcvos -ltbtables -o wrtable
$exit
```

Load both STSDAS and TABLES packages.

Link to the TBTABLES library.

Link to the CVOS library.

After running the `wrtable` executable, the `wrtableTest.tab` file should be present in the current directory. Using the **tprint** task, the contents of the `wrtableTest.tab` file contains information as illustrated in Example 12.

Example 12: Contents of the wrtableTest.tab file.

```
cl> tprint wrtableTest.tab
#  Table wrtableTest.tab  Mon 16:47:08 25-Oct-1999

#K HISTORY  Simple test program to create a table.

# row REGION PLATE             RA            DEC      EPOCH SURVEY  DISK
#                           hours        degrees      years

    1 S256   000Y   105:22:17.580  -45:04:22.476   1980.122 UK        18
    2 S734   006Q   275:41:22.200   -9:58:26.616   1978.647 UK        50
cl>
```

Odds and Ends

The CVOS defines and uses a version macro, CVOS_VERSION, which is a string indicating the version number of the library and a corresponding date of installation. The macro is defined in the `c_iraf_priv.c` file as

```
const char CVOS_VERSION[] = {“CVOS Version N (DD-MMM-YYYY)”};
```

In order to determine the version of the CVOS library or the version of the library used by an executable, the following can be done.

```
> strings libcvos.a | grep “CVOS V”
```

```
> strings myprog.e | grep “CVOS V”
```

In both of these instances, the output looks like the following for the latest version of the CVOS:

```
CVOS Version 3.2.2 (03-September-1999).
```

Chapter 2: CVOS Reference Manual

In the works!

In the meantime, *A C Interface to IRAF’s VOS Library* written by A. Farris (1996) can be used as a reference guide in order to generate new C interface routines.

This page intentionally blank.

Appendix A: Checklists

For the convenience of programmers new to building C tasks in conjunction with the IRAF system, this appendix contains two checklists which can be used to make sure that one has implemented all the necessary syntax and procedures to build successful host-level and native IRAF C tasks.

Host-Level Task Checklist

C source code

Did you remember to...

- include <c_iraf.h> before any other CVOS header file?
- include all CVOS library/package header files needed to support functionality used in the source (e.g., <ximio.h>, etc.)?
- call the c_irafinit() function to initialize the IRAF libraries?

Mkpkg file

Did you remember to...

- use the XFLAGS, -Inolibc?
- use the XFLAGS, -p -stsdas?
- use the LFLAGS, -H?
- use the LFLAGS, -p stsdas?
- use an LFLAGS, -p tables, *if* the program requires linking libraries which are part of the TABLES package?
- link the CVOS library, -lcvos?
- link any other needed IRAF or package libraries?

Host System settings

Did you remember to...

- set any needed system environment variables?

Native IRAF Task Checklist

C source code

Did you remember to...

- include <c_iraf.h> before any other CVOS header file?
- include all CVOS library/package header files needed to support functionality used in the source (e.g., <ximio.h>, etc.)?
- use the IRAFTASK(taskname) macro instead of main()?
- not make any calls to exit() or _exit()?
- only use the return statement with a value to transfer flow of control from subroutines to the top level routine? It is not necessary in the top level routine to have a return statement as no values should be passed to the IRAF CL.

Mkpkg file

Did you remember to...

- use the XFLAGS, -Inolibc?
- use the XFLAGS, -p -stsdas?
- use the XFLAGS, -Istsdaslib$cvos/irafstdio to access the special <stdio.h>?
- use the LFLAGS, -H?
- use the LFLAGS, -p stsdas?
- use an LFLAGS, -p tables, *if* the program requires linking libraries which are part of the TABLES package?
- link the CVOS library, -lcvos?
- link the C library, -lc?
- link any other needed IRAF or package libraries?