Volume II

Khoros Programmer's Manual

Chapter 1

WRITING PROGRAMS / VIFF FORMAT

Writing Programs Under the Khoros System

Primary Author(s):

Danielle Argiro & Charlie Gage

Printed: May 9, 1991

WRITING PROGRAMS / VIFF FORMAT

A. OVERVIEW OF THE KHOROS PROGRAMMER'S MANUAL

Volume II of the *Khoros Manuals* contains documentation for the program developer who wishes to create new applications using the Khoros user interface development tools.

In simplest terms, a Khoros application consists of application-specific algorithms which are accessed by an automatically generated user interface based on a User Interface Specification (UIS). The Khoros environment supports the creation of three types of user interfaces: graphical, textual, and the visual language.

Chapter 2 provides an in-depth explanation of the use of the UIS file. The first task for the application designer who wishes to develop a Khoros program is to create a UIS file, so a careful study of this chapter is important for anyone who wishes to develop a program under the Khoros system. Information concerning the physical appearance and functionality of an application interface is contained here. Designed as a high level description, the specification is used directly to generate the code for both the graphical and the command line user interfaces.

Chapter 3 gives instructions for the use of **preview**, a program that allows you to prototype the graphical user interface defined by a UIS.

Chapter 4 gives instructions for the use of **composer**, a program that allows you to interactively edit a UIS file and a Program Specification (PS) file.

Chapter 5 details the use of the **ghost routines** for automatic generation and maintenance of code for the command line user interface (CLUI) of a Khoros program. Use of the **ghost routines** is required for creation of any Khoros program.

Chapter 6 explains **conductor**, an automatic code generation program for xvroutines only. **Conductor** generates the C code necessary to mediate between the *xvforms* library and the application program's graphical user interface.

Chapter 7 gives examples of xvroutines; there is no need for additional examples of vroutines, as more than enough are available in the *vipl* and *dspl* libraries.

Chapter 8 introduces the libraries which form the foundation on which the Khoros system is based. They provide utilities so that the program developer may create applications under X Windows, Version 11, Release 4, while knowing only a little about X programming. For those who have already developed programs using the Khoros libraries, changes to library routines are listed in the first section.

Chapter 9 contains the documentation for the layout of of the Khoros Source Tree. In addition, it gives details on the use of *Makefiles* and *Imakefiles*.

B. WRITING PROGRAMS IN THE KHOROS ENVIRONMENT

This chapter provides information necessary to writing a new program and its integration into the Khoros system. An introduction is provided outlining the differences between the two main types of programs in Khoros (*vroutines* and *xvroutines*). Section B specifically outlines the writing of a *vroutine*, while section C outlines how to write an *xvroutine*. Each of these sections gives appropriate instructions on creating the UIS file, using the **ghost routines** for automatic program generation, and the installation process. Notes on use of the Program Specification file are given in section D, which is followed by section E, on integrating the new program into **cantata**. Finally, section F presents information on maintenance of new programs in the Khoros system.

In addition, this document also contains general information about the Khoros system, including argument and file naming conventions, and important style and coding conventions. It is *strongly* recommended that you follow these guidelines for consistency and maintainability of your programs. Usage of the provided information and recommendations will result in a high degree of standardization in the source code and an increased understanding of the Khoros system.

B.1 INTRODUCTION TO KHOROS VROUTINES & XVROUTINES

First of all, it is necessary for the reader to distinguish between the two types of programs in the Khoros system: programs that are referred to as *xvroutines*, and those that are referred to as *vroutines*. There are also some programs in the Khoros system that are a cross between the two. These are referred to as *hybrid* routines.

Usually, an *xvroutine* is an application which has a graphical user interface, *regardless of whether or not it is executed from the command line*; there are only a few notable exceptions to this rule. An xvroutine is characterized by the fact that it must link against at least one of the *xvforms, xvutils, xvdisplay,* or *xvgraphics* libraries. In general, an xvroutine must also use the **conductor** code generation program to initialize and extract information from the graphical user interface. An example of an xvroutine is **editimage**, which is used to interactively display and manipulate an image.

Khoros *vroutines* also have a graphical user interface, but this graphical user interface is provided entirely through their inclusion into the **cantata** visual language, or via interpretation and display by the **composer** or **preview** programs - it is NOT supported by the programs themselves. When a *vroutine* is run from the command line, there is no graphical display; data is input, processed, and output. An example of a vroutine is **vadd**, which is one of the many image processing routines within Khoros.

The first task of a programmer working in the Khoros environment is to decide which of the above types of programs they will be writing, let us explain the difference in greater detail. Xvroutines will always involve some sort of graphic display on a screen supporting X Windows, *independently of any accessibility they may have through* **cantata**. Xvroutines always use the X Toolkit and the Athena widget set indirectly (since they will link against one of the Khoros libraries that uses X11.4). Xlib, the X Toolkit and/or the Athena Widget Set may also be used directly by the xvroutine if desired. Examples of *xvroutines* include all the major applications of Khoros: **animate, cantata, composer, editimage, preview, viewimage, warpimage, xprism2,** and **xprism3.**

All these programs meet the first criterion for an xvroutine: they *always* have a graphical user interface, regardless of whether they are run from the command line, or executed via **cantata**. With one notable exception, the Khoros programmer for each of these programs had to use the **conductor** code generation program in the development and maintenance of the program. The only exception to this rule is **cantata**, which, as a necessarily extensible visual language, is *interpretive* -- that is, it interprets the graphical user interface on its own, without the aid of automatically generated code, at runtime.

While the Khoros xvroutines are highly visible, the vast majority of the programs that make up the Khoros system are vroutines rather than xvroutines. These include all the programs in the image processing library (vipl), all the programs in the digital signal processing library (dspl), all the file-formatting programs, all the classification algorithms and morphology routines ... in general, almost all the Khoros programs besides those major applications listed above.

Some programs, however, cannot be classified as either a vroutine or an xvroutine. For instance, some programs may involve graphic display (like an xvroutine), but may NOT have an interactive graphical user interface when run from the command line (like a vroutine). These *graphic hybrid* routines must be located with the xvroutines in the Khoros source tree, as they will usually need the utilities provided by the *xvdisplay* or *xvgraphics* libraries. However, they are NOT xvroutines, per se, as they are not interactive, and do not have a graphical user interface when run from the command line; thus, they do NOT use the **conductor** code generation program, as they have no graphical user interface from which to extract information; they simply provide non-interactive graphic display. Examples of these programs are: **iconimage, putimage, xverror,** and **xvviewer**. The reader is highly encouraged to write this type of application; however, be aware that it falls into a "grey" area of the Khoros domain - it should be installed in the source tree with the xvroutines, but will follow the same procedures as that of a vroutine.

On the other hand, some existing non-graphic Khoros programs must be located in the Khoros source tree with the xvroutines since they must link against the *xvforms* library. In particular, **getimage, ghostreader, ghostwriter,** and **conductor** take advantage of these libraries to parse and deparse lines out of UIS files, and perform other necessary operations. Since they do not display graphics of any sort (user interface or otherwise), they are not xvroutines, and do not use **conductor**; the procedure followed to write them was, like the graphic hybrid routines, the same as the procedure that will be outlined for vroutines. These routines are classified as *non-graphic hybrid* xvroutines. In contrast to graphic hybrid routines, however, it is NOT RECOMMENDED to attempt to write a non-graphic hybrid routine! The reason for this is (identical to the reason that the reader is not encouraged to write an interpretive application such as **cantata**) that a detailed & thorough knowledge of the internal workings of the *xvforms* and *xvutils* libraries would be necessary for such a task. While we are confident that Khoros users may be quite capable of acquiring such knowledge, any use of the non-public, undocumented regions of these libraries is strongly discouraged, as this code is subject to change at any time without notice.

To summarize briefly, simply remember this simple rule of thumb: if your program will have a graphical user interface when it is run from the command line, it will be an *xvroutine*; if it doesn't, it will be a *vroutine*. In the case that you would like to write an application that is graphical but non-interactive, it will be a *graphic hybrid* routine; follow the instructions for writing a vroutine, but installing an xvroutine. The only real difference between writing an xvroutine and writing an vroutine (aside from adding the code that will provide the actual functionality of the program), is that when writing an xvroutine, you will be required to perform the extra step of running the **conductor** code generation program in addition to using the **ghostwriter** code generation program, and you will have to do a little work in order to integrate the code generated by these two programs. The final difference is that vroutines and xvroutines are located in different sections of the Khoros source tree.

Regardless of the type of program, adding a new program to the Khoros system involves integrating it into the **cantata** graphical user interface. This is easily achieved by modifying the **cantata** interface to allow access to any new program added to the system. Once a new program is written, it is necessary to follow a few guidelines to ensure that the program is integrated correctly within the graphical user interface and to provide consistent and maintainable code.

Throughout the course of writing a new program, several utilities are available to aid the programmer in writing the source code and designing a form for the graphical user interface; two of these have already been mentioned. A complete description of each utility, with usage information, can be obtained by reading the appropriate sections of the Khoros manual; online manual pages will provide a quick reference for those already familiar with the utilities. A listing of these utilities is provided below.

ghost routines : ghostwriter, ghostreader, and ghostcheck-

These are a set of utilities (used with ALL Khoros programs) which are used to facilitate the writing of programs in the Khoros environment. The user interface specification file (UIS file) provides the information which **ghostwriter** uses to generate code which automatically parses the command line, print manual pages, and generate usage statements. **Ghostwriter** ensures that certain conventions are followed for all new code added to the Khoros system. This is to ensure that the command line and graphical user interface are correctly applied and installed. In addition, with the use of a program specification file (PS file) **ghostwriter** guarantees that the source code can be easily modified and maintained within the framework of the Khoros system. **Ghostreader** automatically updates the PS file according to changes made in source code and man pages, while **ghostcheck** attempts to provide advice when, in the course of program maintenance, the user is not sure whether to begin by running **ghostwriter** or **ghostreader**. In-depth explanations of the **ghost routines** are provided in Chapter 5 of the Khoros Programmer's Manual.

conductor -

This is a utility (used ONLY with xvroutines) which automatically generates code to initialize and extract information from the graphical user interface. A complete explanation of **conductor** is provided in Chapter 6 of the Khoros Programmer's Manual.

composer -

This is a top level tool (used with ANY Khoros program, if desired) that provides interactive creation/modification of the UIS pane file and PS file. In addition, it presents a graphical user interface to the **ghost routines**. Complete documentation on **composer** is provided in Chapter 4 of the Khoros Programmer's Manual.

preview -

This is a utility (used with ALL Khoros programs) that allows the user to view and edit an existing graphical user interface specification (UIS) file. **Preview** is covered in Chapter 3 of the Khoros Programmer's Manual.

B.2 OVERVIEW OF PROGRAMMING IN THE KHOROS ENVIRONMENT

If you are unfamiliar with the details of the *User Interface Specification (UIS)*, it amy be helpful to review Chapter 2 of the Khoros Programmer's Manual. All of the programs in Khoros are based on the UIS. Reviewing the documentation on the code generators in Chapter 5 (Ghost Routines) may be helpful, if you have not used these routines before. If your program is to be an xvroutine, it is suggested that you read Chapter 6 (Conductor). If you are going to write a vroutine, it is suggested that you first try to recreate a existing program that is similar to the one you will be attempting to write, using it as a model or example (this is not really recommended for writers of xvroutines, as existing programs are simply too long to attempt to re-create. However, it is advisable to become familiar with the graphical user interface and operation of other xvroutines).

If your new program is an xvroutine, it is recommended that prior to designing your xvroutine, you become familiar with the layout and operation of existing xvroutines. Chapter 7 provides example xvroutines -- simple programs that follow the Khoros standards for xvroutines, and that use the *xvforms*, *xvutils*, *xvgraphics* and/or *xvdisplay* libraries. After you have used **ghostwriter** to generate your command-line program, you will need to use **conductor** to generate the code that will initialize and extract information from the graphical user interface. Next, you will need to integrate the main driver generated by **conductor** with the one generated by **ghostwriter**. There are instructions for doing this in Chapter 6 of the Khoros Programmer's Manual.

After getting the program started under the Khoros system, you will, of course, need to add the code that will give your new program its functionality. Finally, you will need to install your new routine in the Khoros source tree, and add it into the **cantata** visual language. To help you do this with your vroutine is a utility program called **kinstall**. Unfortunately, **kinstall** does not work with xvroutines, which must be installed and added to **cantata** by hand. Sections B and C of this chapter describe each of the procedures that should be followed when writing new programs under the Khoros system. If your intention is to write a vroutine, then follow section B on writing a vroutine, otherwise proceed to section C on writing an xvroutine.

C. WRITING A VROUTINE

The information provided in this section concerns the process of writing a vroutine. This is prefaced by an outline of the steps involved in writing a vroutine, followed by more detailed information concerning each step in the process. It is *strongly* suggested that you install any new programs in the KHOROS_CONTRIB (or *contrib*) source tree. This will allow you to easily update or install patches to the Khoros system as they become available. Note that if you install new programs in the KHOROS_HOME (or *Khoros home*) source tree, then it will be much more difficult to maintain the Khoros system and install patches. New programs installed in KHOROS_CONTRIB will still be accessable under **cantata**, from the contrib pulldown menu.

One of the first things to do when writing a new vroutine is to decide where to put your program within the contrib source tree. In the Khoros home source tree, we have grouped routines with a similar theme into a common category. For example, routines that have two input images and perform an arithmetic operation on the images, are grouped into a category called *arith_binary*. You may want to create a category for your own routines in contrib, or follow a grouping similar to that in the Khoros home source tree. More information about integrating your program into **cantata** is provided in Section E of this chapter.

1) Creating the UIS file - First, create your User Interface Specification (UIS) file (ie, *.pane file) which will contain the definitions and specifications for both the Command Line User Interface (CLUI) and the Graphical User Interface (GUI) of your program. This necessitates that all input and output parameters be defined and specified prior to writing of the source code. You may do this by using an existing UIS file as a template, and modifying it as desired using **preview**; alternatively, you may use **composer** to interactively generate your UIS file.

2) Using the ghost routines - The **ghost routines** help to automate writing of the source code and ensure consistency throughout the Khoros system. **Ghostwriter** is a code generator that generates your main driver, include file, library file, and man pages. Attention must be paid to the [-type] flag, since it indicates whether the routine links against FORTRAN code. Specify [-type fprog] if your routine links against FORTRAN, otherwise use [-type prog], which is also the default. The first time **ghostwriter** is run, a template Program Specification (PS) file is created for you, which contains key fields for the documentation and source code. DO NOT DELETE any of the *tags* that **ghostwriter** will insert into your source code and man pages! The **ghost routines** facilitate the writing and maintenance of new programs via this PS file, which, once completed, contains all of the necessary information to re-generate the manual pages and associated source code files at any time, thus making the future addition or deletion of command line arguments a trivial process.

3) Writing the new program - As you add your code and documentation to the files generated by **ghostwriter**, run **ghostreader** occasionally to update your PS file. Documentation and source code will be inserted into the program specification (PS) file between the appropriate keys. Several important points are discussed later in this chapter concerning arguments, error handling, data structures, and memory allocation.

4) Installing the new program & integrating it into cantata- Once the new program is completed to your satisfaction, you must read it in as a UIS pane file and verify it, then you can install it in the Khoros Source Tree, and integrate it into the **cantata** visual language. To ensure that your vroutine gets installed and added to **cantata** correctly, the **kinstall** utility program should be used.

The following sections describe each of the above steps in greater detail.

C.1 CREATING THE UIS FILE

The first step in creating a new program is to create a UIS (*.pane) file for the new program. A thorough discussion of UIS files and the methods for creating a UIS file is provided in Chapters 2 through 4 of the Khoros Programmer's Manual. The UIS file can be thought of as a program. That is, each line of the UIS file has a strictly defined syntax and meaning. Therefore, particular attention must be paid to order of occurrence and syntax.

The *.pane file for your vroutine will be used for two purposes: it will be used as input to **ghostwriter** to generate the command line user interface for your vroutine, and it will be used to integrate your vroutine into **cantata**. For the purpose of using **ghostwriter**, it is important to make sure that the *.pane file provides a complete description of all desired program arguments. On the other hand, for the purpose of using the *.pane file to integrate your vroutine into **cantata**, you must make sure that the *.pane file describes a visually coherent I/0 pane that will fit into the appropriate subform of **cantata**, that it includes a Help Button, and that it has a Routine button so that the user will be able to access your new vroutine through **cantata**. No extraneous UIS lines should appear in your *.pane file.

There are currently 28 different types of UIS lines, some of which are required to appear in every UIS file. Certain UIS lines contain information that is used by both the command- line and graphical user interfaces, whereas others contain information that is used only by the graphical user interface.

There are several **required lines** for all UIS (*.pane) files. The **required lines** include:

- **StartForm (-F) -** The (-F) line begins the user interface specification. This line contains several fields which contain information concerning the version number of the *xvforms* library, the geometry of the form, and the title of the form among others (the -F line is required in every UIS file).
- **StartSubForm (-M) -** The (-M) line contains information describing the size, position, and title of the subform (the -M line is required in every UIS file).
- **StartPane (-P) -** The (-P) line describes the graphical user interface pane that provides a backplane for selections and action buttons to allow input to the application program (the -P line is required in every UIS file).
- **Help (-H) -** The (-H) line describes a specialized button on the user interface which will bring up an on-line help file when selected. The -H line is required to appear on all **cantata** panes; therefore your *.pane file should contain a -H line that references the man1 page for your vroutine, for when it is integrated into **cantata**). Note that the -H line is ignored by **ghostwriter**.
- **End (-E) -** The (-E) line is used to end a set of definitions in a UIS file. The (-E) line closes a corresponding definition line in the UIS file. For example, a (-F) line must have a corresponding (-E) line to complete the form definition, a (-M) line must have a corresponding (-E) line to complete the subform definition, and a (-P) line must have a (-E) line to complete the pane definition.

Optional lines for structuring the user interface include:

- **Toggle (-T) -** The (-T) line defines a set of one or more *selection* items of the same type. The value of the toggle group will take on the default value of the selected item (except in the case of a Logical toggle, which takes on the number of the selected item, where logical members of the toggle are numbered starting at 1 with the first item).
- **MutualExclusion (-C) -** The (-C) line defines a mutually exclusive group of *selections*. Members of the mutually exclusive group may be of the same or different types. The value of the mutually exclusive group will take on the current value of the selected item.

Other lines that may appear in the UIS pane file pertain to *selections* and *actions*, which are used to identify the inputs, outputs, and program arguments. These include the following:

- **InputFile (-I) -** The (-I) line specifies an *input file selection*.
- **OutputFile (-O) -** The (-O) line specifies an *output file selection*.
- **Integer (-i) -** The (-i) line specifies an *integer selection*.
- **Float (-f) -** The (-f) line specifies a *float selection*.
- **String (-s) -** The (-s) line specifies a *string selection*.
- **Logical (-l) -** The (-l) line specifies a *logical selection*.
- **Routine (-R) -** The (-R) line describes a specialized action button which will appear on the pane when it is integrated into **cantata**. When the user clicks on this

button, the vroutine will be executed (this line is ignored by **ghostwriter**).

- **Blank (-b) -** The (-b) line is used for adding comments and extra titles into the pane, for extra clarity on the pane when it is integrated into **cantata**. Note that the -b line is ignored by **ghostwriter**.

A complete description of each of the UIS lines can be found in Chapter 2, section B.3 of the Khoros Programmer's Manual.

An example of a UIS pane file is presented below. This illustrates the use and placement of the various lines in a UIS pane file.

```
-F 4.2 1 0 170x7+10+20 +35+1 'CANTATA Visual Programming Environment for the KHOROS System' cantata
  -M 1 0 100x40+10+20 +34+1 'Histogram' histogram
    -P 1 0 80x38+22+2 +0+0 'Enhancement Using Local Standard Deviation and Mean' venhance
      -I 1 0 0 1 0 1 50x1+2+2 +0+0 ' ' 'Input Image   ' 'input image filename' i
      -O 1 0 0 1 0 1 50x1+2+3 +0+0 ' ' 'Output Image  ' 'output image filename' o
      -i 1 0 0 1 0 50x1+2+5 +0+0 1 100 3 'Window Width ' 'window width of local area' w
      -i 1 0 0 1 0 50x1+2+6 +0+0 1 100 3 'Window Height' 'window height of local area' h
      -f 1 0 0 1 0 50x1+2+8 +0+0 0 1 0.5 'Tuning factor' 'specifies scale factor' k
      -f 1 0 0 1 0 50x1+2+9 +0+0 0 2000 1 'Min Std Dev  ' 'specifies minimum standard deviation' v
      -R 1 0 1 13x2+1+13 'Execute' 'do operation' venhance
      -H 1  13x2+39+13 'Help' 'man page for venhance' KHOROS_HOME/doc/manpages/venhance.1
    -E
  -E
-E
```

In this example, there are three lines for structuring the user interface. These include the **StartForm (-F)** line for starting a user interface specification, a **StartSubForm (-M)** line beginning the definition of a subform, and a **StartPane (-P)** line beginning the definition of an I/O pane. Note that each of these lines has a corresponding **End (-E)** line to terminate the respective set of definitions. Chapter 2 discusses other types of UIS lines that are used to structure that graphical user interface into multiple subforms, and multiple panes; however, these lines are generally not used by vroutines, and are always ignored by **ghostwriter**. This particular example has both an **InputFile (-I)** line and an **OutputFile (-O)** line to specify the input and output file selections. There are also lines for **Integer (-i)** and **Float (-f)** selections, as well as a **Routine (-R)** action button line and a **Help (-H)** line for access to the on-line man page.

Next we will provide a brief discussion of some of the more common fields in the UIS file to get the reader started in constructing one of their own.

The **StartForm (-F)** line always has the version number as its first field. This provides a way of keeping track of outdated UIS files. The next field in the (-F) line, and the first field in most other lines, is the activation field. This field allows various parts of the form to be disabled, thus is should always be set to 1 (TRUE), unless that part of the form is to be deactivated. The next field that is found on most lines is the selected field, which is used by the *xvforms* library to indicate if an item has recently had its value changed by the user. The *selected* field should always be set to 0 (FALSE) with a few exceptions as noted in Chapter 2.

Lines that appear only in the pane definition contain either three or four additional fields prior to the geometry string field. The *optional* field, which is the third field, determines whether a selection is optional. The *option selected* or fourth field specifies whether the default value is to be used, and the *live* or fifth field specifies if the selection is to return immediately when the user hits <cr>. The I/O lines contain a sixth field called *file type* that will be used in the future.

A geometry string appears in each line associated with a form, button or selection. The geometry string, which is of the form [width x height + xoffset + yoffset], specifies the overall geometry of an item, where the offsets are referenced from the upper left hand corner of the parent widget. The geometry string determines the size of the master form for the StartForm (-F) line, the subform for the StartSubform (-M) line, and the pane for the StartPane (-P) line. Default values for the geometry string are produced for these lines when **composer** is used to design the pane file. If the decision was made to design the pane file yourself, then it is suggested that you use an existing UIS pane file as a template. **Preview** can be used to display the UIS pane and make editing changes once it has been created.

The next field specifies the offset for the *title*. These are generally set to +0+0 for I/O and other selections within the pane definition. Other fields that are found in the pane definition lines include selections for the upper and lower bounds of the input, the default value or file, the title, and the description.

This brief introduction to the common fields associated with the lines of a pane definition is provided to familiarize the user with a UIS pane file. The reader is urged to consult Section B.5 of Chapter 2 for a detailed description on the use of each field.

There are three other lines that may be used to simplify selections on UIS panes. These include *Logicals*, *Toggles*, and *Mutual Excusions*. A **Logical (-l)** line allows the user to specify a boolean value as a response to a question or selection. An example of a *logical* selection is provided below:

```
-l 1 0 1 1 0 20x1+1+5 +0+0 1 'Display Grid?' 'No' 'Yes' 'request grid display?' grid
```

This line illustrates the use of an optional logical selection. This means that the selection will appear with an optional box in front of the title. Since the *option selected* field is set to TRUE, the optional box will be initially highlighted, indicating that the default action will be to use the value of the logical. This value, according to the value of the *default* field, will be 1 (TRUE), interpreted in this case as a response of "Yes" to the question, "Display Grid?"

A **Toggle (-T)** line allows the user to specify one of a predefined number of choices. The choices may be of any type of argument, as long as they are all of the same type. For example, it is possible to define a toggle of integers, a toggle of floats, or a toggle of strings. The (-T) line begins the definition of a set of toggles. The following example illustrates the use of a *toggle of logicals*:

```
-T 1 0 0 1 0 20x1+1+5 +0+0 4 'Plot Type' 'type of plot' plot_type
  -l 1 0 1 0 0 10x1+0+1 +3+0 0 '2D' '2D plot' dummy
  -l 1 0 1 0 0 10x1+14+1 +3+0 0 'Discrete' 'discrete plot' dummy
  -l 1 0 1 0 0 10x1+28+1 +3+0 0 'Histogram' 'histogram plot' dummy
  -l 1 0 1 1 0 10x1+7+2 +3+0 0 'Polymarker' 'polymarker' dummy
  -l 1 0 1 0 0 10x1+21+2 +3+0 0 'Linemarker' 'linemarker' dummy
-E
```

This example illustrates one way in which the (-T) line might be used to create a toggle which yields a value for the plot_type variable. This example demonstrates the use of a required logical toggle. The toggle (ie. the integer variable "plot_type") will take on values of 1, 2, 3, 4, or 5, depending on which one of the logicals is selected.

In contrast to integer toggles and string toggles, the *defaults* of the logicals in a toggle do not matter - they are all set to 0. In all UIS lines that are members of a toggle, the *variable* fields on the logicals will not be used - we put in "dummy" to remind us of this fact. Check to make sure that the default of the toggle is set correctly - the *default* field on the (-T) line is set to 4, and the fourth logical is the only one to have its *option selected* field set to 1. Thus the default of the toggle will be the value 4, corresponding to the selection 'Polymarker'. Finally, check the geometry, to make sure that you have specified the *y* field relative to the (-T) line; the geometry of the logicals will specify 2 rows of toggle items, the top row with three selections, and the bottom row with two selections.

A **MutExcl (-C)** line defines a group of two or more arguments that need not have predefined values and may be of different types. The only field on the (-C) line specifies whether a selection is required or is optional (ie. a value of 0 indicates that the selection is optional). If the selection is required (ie. a value other than 0 is provided), then the user will be able to provide and use a value for one and only one of the selections in the group. Unlike the toggle (-T) group, which is limited to a single type of selection, the mutually exclusive group may contain a mixed group of Integer, Float, String, Logical, InputFile, or OutputFile selections. All selections in a mutually exclusive group MUST be specified as optional. The user will only be able to "turn on" one of the mutually exclusive selections (by highlighting the optional box) at any one time; this is the only one of the selections' values that is guaranteed to be valid at any one time.

The following example illustrates the use of a *mutually exclusive* group:

```
-C  1
   -I  1  0  1  1  0  1  45x1+1+13 +0+0  './'  'Input Path'  'input path selection'  input_path
   -O  1  0  1  0  0  1  45x1+1+14 +0+0  './'  'Output Path'  'output path selection'  output_path
   -s  1  0  1  0  0   45x1+1+15 +0+0  ' '  'String Sel'  'string selection'  string1
-E
```

This example demonstrates the use of a required argument for a mutually exclusive group. The only field on the (-C) line requires a value specifying one and only one of the items within the mutually exclusive group. In this example, the first item is selected as the default. Note that the *option selected* field (fourth field) is selected with a 1 in this location.

The operation of a mutually exclusive group in the command line user interface is slightly different from the operation of a mutually exclusive group in the graphical user interface. Note that the default values for the input_path and output_path parameters are both set to "./". When using this mutually exclusive group on the command line, these defaults are unused -- the user will be forced to specify a value for one of the selections, and if they choose the input_path or output_path parameters, will still have to explicitly enter a value, even if they want the "./" directory specified. However, for the graphical user interface, it is good to specify these defaults - the selections will come up in the order listed, and the input_path and output_path selections' parameter boxes will have "./" appearing inside them. The optional box in front of the input_path selection will be highlighted -- if the user wishes to specify the input_path selection as their choice from the group (the default of the group) and they also wish to have "./" as the value of the input_path selection (the default of the input_path selection), they need not do anything.

The reader is once again urged to consult the detailed discussions provided in in Chapter 2 of the Khoros Programmer's Manual for a complete description of the proper syntax and use of each line in a UIS pane file.

Once the UIS pane file has been created, the next step is to create a driver program for the routine, using all of the arguments specified in the UIS pane file. This is easily accomplished using the **ghost routines** provided for this purpose. The next section describes how to use the

ghost routines to create a driver program and generate the framework for the library routine.

C.2 USING THE GHOST ROUTINES

To help speed up the process of writing a program and to ensure consistency in the code, Khoros contains code generators that help to automate part of the procedure of programming. The code generators belong to the **ghost routines**, which includes **ghostwriter** and **ghostreader**.

To get started, create a new directory that can be used as the development directory for the new routine. Create a UIS (*.pane) file using either **composer** or a modification of an existing UIS (*.pane) file, as detailed in Chapters 2-4.

The first tool you will use is **ghostwriter**. However, keep in mind that **ghostwriter** requires a UIS file as input (discussed in the previous sections). **Ghostwriter** can be run with the UIS file to create a template for the PS file, the include file, the source code files for the main driver, the library file, and the manual pages for the program. Alternatively, you can begin with an empty template PS file, fill in the appropriate fields, and run **ghostwriter** to generate the source code and manual page files. We have found that a hybrid method is most popular with the Khoros group; however, after writing one or more programs under the Khoros system, you will find your own preference.

The following example demonstrates the use of **ghostwriter** to generate a vroutine:

```
% ghostwriter -name vroutine
```

This will generate the files "vroutine.c", "lvroutine.c", "vroutine.1", "lvroutine.3", "vroutine.h". Note that in order to use **ghostwriter**, a valid UIS file (in this case, "vroutine.pane") must exist in the current working directory. **Ghostwriter** will look for a valid PS file in the current working directory; if one does not exist, a template PS file ("vroutine.prog") will be created. In addition, **ghostwriter** will also look for a configuration file in the current working directory; if one does not exist, a template configuration file ("vroutine.conf") will be created as well. The configuration file simply specifies the paths to each of the files associated with the new routine. An example of a template configuration file is located in KHOROS_HOME/repos/config/src_conf and is called "TEMPLATE.conf". The use of the configuration file is explained later in this chapter. For more detailed information about the configuration file, see Chapter 5 of the Khoros Programmer's Manual.

There are several options with **ghostwriter** which allow use of a configuration file, formatting, debug statements, and generation of certain files. The various options may be listed using the [-U 1] option; or, you may wish to run the program with prompts for each command line option using the [-P 1] option. An answer file, which specifies the desired arguments to **ghostwriter**, may be created by using the [-A ghost.ans] and [-P 1] options together, making it easier to perform successive runs of **ghostwriter** (a certainty!).

In this example, if we had used the [-A ghost.ans] option to **ghostwriter**, for the vroutine, its answer file, "ghost.ans", for future executions of **ghostwriter** would look like:

```
-name vroutine
-config f
-lib t
-man3 t
```

-install f
-format 1
-debug f
-tag t
-force f
-prog t
-type prog

Now, for successive runs of **ghostwriter**, we need not provide arguments on the command line, or interactively with the [-P 1] option. To execute **ghostwriter** with the exact same arguments as before, all that is necessary is to run:

```
% ghostwriter -a ghost.ans
```

If your code links against FORTRAN (note that the *linpack* and *eispack* libraries are in FORTRAN), you must provide the [-type fprog] flag, or your code will not compile. Notice that executing **ghostwriter -name vroutine** generated the files: "vroutine.c", "vroutine.1", "vroutine.h", "lvroutine.c", and "lvroutine.3". **Ghostwriter** will also make calls to **imkmf** and **makemake** for you, thus generating (respectively) the appropriate "Imakefile" and "Makefile" files for this program. The vroutine is now ready to be compiled; of course, lvroutine() will simply return to the main program (initially, it is generated with only a *return(TRUE)* statement). You may wish to compile at this point, checking for any errors in the driver and include files. To compile, simply type:

```
% make
```

No errors should occur, unless you modify these files in some way. However, if compile errors do occur in any of the source files, you can fix the problems in the source files, but remember to follow any modifications with a call to **ghostreader**. Note that **ghostreader** has a subset of the complete set of **ghostwriter** arguments - those arguments that are identical for the two programs have the same meaning, and neither program checks for invalid arguments, so that you may provide the same answer file, "ghost.ans" for both programs. To execute **ghostreader**, then:

```
% ghostreader -name vroutine
```

or

```
% ghostreader -a ghost.ans
```

The Program Specification (PS) file contains key fields for the insertion of Khoros system documentation, man page documentation, and source code. A template "vroutine.prog" file is created when **ghostwriter** is run for the first time, containing all of the key fields for constructing a complete PS file. Each key field in the "vroutine.prog" file is delineated by *begin* and *end keys*. All comments, code, and documentation is placed between the appropriate *begin* and *end tags* for a particular field. An example of a template PS file is presented in Chapter 5. Chapter 5 should be consulted for specific descriptions and guidelines concerning the information to be included in each of the key fields of the PS file. An example illustrating the use and format of each key field of a *.prog file is included in Chapter 5. This format should be adhered to when creating a PS file to ensure maintainability of the Khoros System.

Ghostwriter creates the source code, include files, and man pages with pre-defined **tags**, that are recognized by **ghostreader**. Thus, segments of code and/or documentation between these tags may be modified by the programmer and then reinserted into the PS file by running **ghostreader**. **Ghostreader** takes as input the 5 files: "vroutine.c","vroutine.h", "vroutine.1", and "lvroutine.3", as well as the old PS file "vroutine.prog". It looks for the tags, and pulls any text between a pair of begin and end tags into the corresponding place in the PS file. In this way, a new PS file may be generated containing the new information.

Running **ghostreader** insures that all modifications are correctly updated in the "vroutine.prog" PS file; thus, when **ghostwriter** is run again (perhaps with the [-A 1] flag), the updates will be written to the appropriate files. BE WARNED that if you modify the source or man page files, and forget to run **ghostreader** before running **ghostwriter** again, *all your changes to source code and man pages will disappear!*

C.3 WRITING THE NEW PROGRAM

All functionality for your program should go into the "lvroutine.c" file, often referred to here as the library routine. The "vroutine.c" is referred to as the driver program, which calls the "lvroutine.c" file. Note that it is possible to write a driver that simply makes calls to other library routines, in which case there would not be an "lvroutine.c" file for this particular routine. The driver is generated by **ghostwriter** from the information contained in the UIS (vroutine.pane) file.

It is STRONGLY recommended that you read the man pages on the *verror, vgparm, vmath, vrast,* and *vutils* libraries. These are public Khoros libraries, and contain many useful routines for I/O, math, and reading/writing/checking of images that help the programmer avoid "re-inventing the wheel".

When the source code for the library routine is completed, you are reminded (again!) to use **ghostreader** to update your PS file in preparation for any future runs of **ghostwriter**. The program should then be carefully tested for proper operation, using all possible options both separately and in different combinations.

If your routine does not have a library file (lvroutine.c), then there is no need to complete a man3 page for your routine. If your routine has a library file, then you will need to complete the man3 page information for your new library routine. The purpose of the man3 page is to provide a detailed description of the library routine. This should include a discussion of the algorithm used and any parameters that are passed to or from the routine. If the algorithm was taken from another source, be sure to provide references to the source. No examples of the library call are necessary unless they clarify the call to the library routine.

Recall that **ghostwriter** created a skeleton "lvroutine.3" file for your program. If you are writing a vroutine, you should update this file with careful documentation of your library routine, following modifications with calls to **ghostreader**.

The "lvroutine.3" file may be edited directly. Modifications should be followed by a call to **ghostreader**; however, when updating man pages the programmer often prefers to edit the PS file. If you like, you may edit the PS file ("lvroutine.prog"), looking for keys that mark the man3 information. WARNING: you should run **ghostreader** immediately before editing the "lvroutine.3" file, and run **ghostwriter** immediately afterwards. It is best to look at some of the man3 files for other library routines to get the feel of how to write one. Man3 files can be found in `KHOROS_HOME/man/man3`. The man3 file should include usage information about all of the parameters passed into the library routine including any restrictions.

Finally, you will be required to complete the man1 page information for your new program. The purpose of the man1 page is to provide a detailed description of the function of your new program. This should include descriptions of the input files, output files, and arguments to the program. Try to be as complete as possible in your descriptions of the inputs, outputs, and arguments to the program. This information will be of valuable assistance to anyone using your program. Man1 pages are very similar to man3 pages, except that they discuss the program in general. The procedure for modification of man1 pages is identical to that for modification of man3 pages. You may want to look at some of the man1 files for other Khoros programs to get the feel of how to write one. Man1 files can be found in `KHOROS_HOME/man/man1`.

Formatting for both types of man pages should be done in *nroff* format. This is to ensure that the on-line manual pages are generated correctly and to maintain consistent formatting throughout the documentation. Certain fields found in the PS file need no formatting since they must consist of only a one line description (notes on the PS file will follow later). A few of the more common *nroff* formatting commands that you may find useful include:

.IP/.LP - for new block paragraphs
.RS/.RE - for indenting
.DS/.DE - for non-formatted sections
.SH - for section headers
.sp # - for adding # blank lines

C.3.1 Conventions used for Image Processing Algorithms

It is common to have multiband images in an image processing system, and Khoros has provisions for handling multiband images. The convention for addressing multiband images within Khoros is to refer to the first band as band number 0, and the second band as band number 1, etc. Therefore when writing a new routine that works with multiband images, you should follow this convention. (For details on VIFF see section H of this chapter.)

Classification routines often deal with multiband images, as they frequently work with multiband cluster and class images. The cluster number and class images should output as data storage type *integer*. This is to ensure that there are plenty of available numbers for clusters. The numbering convention should start with 0 and increase. Cluster number 0 should correspond with cluster center 0, etc. In most cases the numbering scheme will be from 0 to N-1, where N is the number of cluster centers. The convention is to make cluster center vectors of data type float. Also, variances of the centers should be of type float.

During the course of processing an image for classification, a border is often created around the image due to windowing algorithms. Therefore, all clustering/classification and labeling/shape algorithms should have the provision of ignoring image borders. The default output pixel value for borders should be 0, but this should not matter if it is ignored.

Classification routines should not assign NULL to any cluster center. There may be cases when a cluster number does not have a corresponding center, and in cases like this the cluster number should be 0, and the corresponding center vector will be 0's. The program *viso2* can produce a situation like this, and in this case there will be N+1 centers (0 ... N). This will still actually provide valid input to other algorithms, but will produce strange results. Therefore the programmer must be aware of cases such as this, and thoroughly document it in the manual page for the routine.

C.3.2 Conventions used for Digital Signal Processing Algorithms

There are two representations of signals recognized in the DSPL library. The first is vector signals, which will be called *vector oriented*. In this data representation, adjacent elements of a signal can be found in the same location in the previous or next data band. In this way, data is processed *through* the bands. The second representation of signals is what is referred to as *band oriented*. That is, each band contains a single, complete signal that is arbitrarily related, or not related at all, to signals occurring on other bands. In this way, data is processed *on* the bands.

Two constants are defined for use by dspl routines: DSP_BAND and DSP_VECTOR. These are used when referring to these data types in a program. These representations allow much flexibility because the band orientation is more common, while the vector representation allows the band row-column dimensions to encode spatial, time, frequency gradient or other information. Signals that are organized on a band should generally be 1 column by n rows, where n is the length of the data sequence. This follows established conventions and simplifies numerical analysis when implementing linear equations.

Two programs have been written to simplify the use of these two formats. They are *dload_vector()* and *dunload_vector()*. *Dload_vector()* is used to reorganize the data into an array of pointers to data vectors, (i.e, float **). This allows a simple for loop to be used to process all data sets in a VIFF file. *dunload_vector()* returns the data to its original organization. The UIS file should have an entry using a logical variable named 'd' to select which organization the data is assumed to be organized in. There is no information on data organization inside the VIFF header structure. For more information on these routines, please refer to the section on the vutils library in Chapter 8 in the Khoros Programmer's Manual.

Occasionally there are times when it is desirable to handle the real and imaginary components of a complex data set separately. In this case, they are treated as independent, real vectors of data. Where appropriate, dspl programs written for Khoros will have a processing option referred to in the pane file as 'j'. All programs written for the Khoros dspl library will handle real and complex data.

C.4 INSTALLATION OF NEW PROGRAM

The installation process of a new routine consists of moving all the associated files of a new program into their proper places in the Khoros contrib source tree, and compiling the program. As explained previously, you are not prevented from installing a new routine into the Khoros home source tree; however, it will be much easier to maintain Khoros and install patches if new user routines are installed in the Khoros contrib source tree. The Khoros contrib directory structure mimics that of Khoros home. A source tree is provided in Khoros contrib for the purpose of installing new user routines. You might want to examine the structure of the Khoros home source tree to get a feel for the organization of the Khoros system. The layout of the Khoros home source tree and contrib is provided in Chapter 9 of the Khoros Programmer's Manual.

The following sections give detailed explanations on the use of Khoros configuration files, and the step-by-step process for installing your vroutine.

C.4.1 Configuration Files

Before installing a program, you must ensure that a correct configuration file exists for your program. The configuration file is simply an ascii file that specifies the location within the Khoros source tree for all of the source code files, document files, UIS files and PS files associated with your program. Once your program is installed, the configuration file is used by by **kinstall** to find the appropriate locations for installing files. Later, **ghostreader** will use the configuration file to locate the appropriate files for input before creating a new PS file, and by **ghostwriter** to specify the appropriate directory paths for writing the new files.

A template configuration file is created automatically for you the first time that **ghostwriter** is run; the default locations for files in the template file is the local "." directory. As long as you have your program in your working directory, this is sufficient; however, once you are to install your routine, this configuration file must be manually updated to specify the pertinent locations of the files associated with your program. There are ten *file keys* in the configuration file that identify a particular file associated with your program; each file key is followed by the path that specifies the directory in the Khoros source tree where that file is to be located. An example configuration file is presented below for the vroutine, "vadd". It has each of the 10 file keys filled out with the appropriate path information.

```
cfile: vipl/arith_binary/vadd
hfile: vipl/arith_binary/vadd
lfile: vipl/Lib
progfile: vipl/arith_binary/vadd
man1file: $KHOROS_HOME/man/man1
man3file: $KHOROS_HOME/man/man3
panefile: $KHOROS_HOME/repos/cantata/subforms/arith_binary
helpfile: $KHOROS_HOME/doc/manpages
subhelpfile: $KHOROS_HOME/doc/cantata/subforms/arith_binary
topsrc: $KHOROS_HOME/src
```

Each file key specifies the path where a specific file will be installed. All of the keys except "helpfile", "subhelpfile", and "topsrc" are also used by the **ghost** routines, and are explained in Chapter 5 of the Khoros Programmer's Manual. The **kinstall** program uses all of the keys, the remainder of which are used as follows:

helpfile

All man1 (*.1) pages are formatted slightly differently for use in the **cantata** online help pages. Online help pages for **cantata** are always located either in `$KHOROS_HOME/doc/manpages` or `$KHOROS_CONTRIB/doc/manpages`. The

kinstall program will format your man1 file for **cantata**'s online help, and install a copy of it here.

subhelpfile
This is the path to the `Overview.doc` file that provides the online help for the subform of **cantata** to which the pane associated with your program will be added. You will edit this file and add a short description of your new program to the short descriptions of other programs that already exist on the subform in question.

topsrcThe "cfile", "hfile", "lfile", and "progfile", will be called the *source* file keys, and can be specified using either full paths (ie. cfile: `$KHOROS_HOME/src/vipl/arith_binary/vadd`) or relative paths (ie. cfile: `vipl/arith_binary/vadd`). If you decide to use relative paths, then you MUST have the "topsrc" file key filled in with the path to the *top level* of the source tree; that is, either `$KHOROS_HOME/src` or `$KHOROS_CONTRIB/src`. Of course, you may choose not to bother with the "topsrc" key, and simply provide the full path in the source keys. In this case, you may delete the "topsrc" line from the configuration file.

Further information on the configuration file can be found in Chapter 5 of the Khoros Programmer's Manual.

C.4.2 Installing Your Vroutine

There are several tools available to aid the programmer in maintaining and installing new programs in the Khoros system. In addition, the tools also include several "low-level" programs that manage the Khoros source tree configuration, which are designed to be used by managers of the Khoros system. From the programmer's perspective, the only tool that is required for installing a new vroutine is **kinstall**. To use **kinstall**, you must be familiar with the configuration file and the **ghostwriter** and **ghostreader** programs. The configuration file was discussed previously in this document, as well as in Chapter 5 of the Khoros Programmer's Manual. Chapter 5 also describes the **ghostwriter** and **ghostreader** programs.

One other file that is used by the **kinstall** program is a *mach* file. The mach file specifies the paths to the source directories of your program and to the corresponding *srcmach directories* within the *srcmach tree*. A srcmach tree is established if a particular site is supporting different machine architectures. For example, if your site is supporting both Decmips and SUN4 architectures, the Khoros system for your site would include a srcmach tree whose layout and contents mimic the "real" Khoros source tree. The srcmach tree consists of symbolic links to files in the original source tree, along with object files compiled for that particular architecture. The mach file specifies the machine names for a site, as well as the path for the srcmach directories. Normally the mach files for a particular site would be configured by the manager of the Khoros system; however, to completely understand the installation process, it helps to understand how the mach file ties in with the installation process.

The "low-level" tools are called by the installation program which takes care of creating the Makefiles and establishing any symbolic links to other *srcmach* directories if they exist.

The syntax for the **kinstall** program is as follows:

```
% kinstall -name xxxx -conf xxxx.conf [-mach yyyy] [-force z]
[type wwww]
```

where,

xxxx is the name of the vroutine that is being installed.

yyyy is the path and filename of the mach file that is being used for the current install.

z is a boolean that specifies whether to use the force option which bypasses the overwrite prompts and edit sessions. The default is FALSE, which does not bypass the overwrite prompts and edit sessions. Note that the force option of TRUE is not allowed if the vroutine has never been installed, or has been removed from the system.

wwwwis the type of program to be installed. If you have a vroutine, use "prog". If you have a vroutine that links against FORTRAN, use "fprog".

The following scenario is provided to illustrate the installation of a routine into the Khoros system using the **kinstall** program. It is assumed that you are installing a vroutine from a directory in your work area, and that all the necessary files for the routine are present in the work directory. The necessary files include: a *.pane file, *.prog file, *.c file, *.h file, *.1 file, and a *.conf file. Optional files include: l*.c file, l*.h file, and a *.3 file. The optional files would be necessary if a vroutine includes library source files. It is further assumed that the programmer has run **ghostwriter** to generate the necessary files from the *.pane and *.prog file. The step-by-step procedure for installing the program "vroutine" in the Khoros contrib source tree appears below.

1) From your work directory, initiate the **kinstall** program as follows:

```
% kinstall -name vroutine -conf vroutine.conf -mach
contrib_mach
```

Note that the optional argument [-mach] is not used in this example. This assumes that the manager of the Khoros system has configured the mach files for your site. Information on mach files is found in Chapter 9 of Volume II, on Configuration Management. The [-force] argument would not be allowed if the vroutine you are installing is a new routine.

2) The **kinstall** program will prompt you for a destination path to an additional library include file. This is only necessary if your program requires additional defines and/or includes, as in the case for many of the file format vroutines. If no path is required, simply respond with "n" to the prompt.

3) The **kinstall** program will check to make sure that the supplied paths are correct (ie. that they exist, and have the correct permissions). **Kinstall** will also provide you with some information about the mach file that is being used and will indicate which program is about to be installed. The paths for the library and driver source will also be provided and **kinstall** will prompt you to continue with the install. Answer "y" to continue with the installation, or "n" to abort the installation. If you continue with the installation, **kinstall** will create a lock file in the Khoros home directory which will prevent other programmers from installing vroutines simultaneously. If someone else is installing a vroutine, a message will be displayed indicating that another user is installing a vroutine, and to try again in a few minutes.

4) Once the installation has begun, you will be provided with some information along the way to indicate what file is being installed and the location where it will be installed. Do not be concerned if the information scrolls off the screen, since all of the information will be mailed to you indicating the status of the installation. Basically, **kinstall** copies the files

associated with your particular routine to the destinations indicated in the configuration file. Symbolic links are created in srcmach directories if your site supports multiple machine architectures. Imakefile & Makefiles are created and updated in each associated directory, and the source files are compiled for each architecture.

5) The **kinstall** program will put you into a session with the editor so that the new vroutine can be added to the appropriate **cantata** UIS file. This file is located in the directory specified by the "panefile" entry in the configuration file. In the same directory will be another file with the extension ".sub". This UIS file contains the specification for the subform within **cantata** to which you will be adding the pane that defines the GUI for your program. Programs with a common theme are grouped together in one subdirectory and are accessed under the same menu selection within cantata. This step is explained in greater detail in the section labeled, "INTEGRATING THE NEW PROGRAM INTO CANTATA".

6) The **kinstall** program will again put the programmer into an editor so that the **cantata** help file can be updated with a short description of the new vroutine.

7) The **kinstall** program will then update the Makefiles and compiling the program for each machine architecture. Finally, **kinstall** will remove the lock file, so that another vroutine may be installed. The last thing you must do is to read the mail that **kinstall** sent to you regarding the status of the installation. The number of mail messages sent to you will depend on the number of machines for your particular site, and whether library source files were installed in addition to the driver program.

8) Now, test your program to make sure that all is well. Don't forget to first execute:

```
% rehash
```

Since you are installing your program for the first time, forgetting to run *rehash* will result in the operating system error, "Command not found". Extensive testing is usually not necessary on installation, as you should have already completed the testing and debugging process in your work area, before installation. However, *never* neglect to do at least cursory testing on your installed program! Sometimes a compile looks successful, but in fact is corrupted due to system problems or corrupted library files. When this is the case, the result is usually an immediate segmentation violation and/or core dump. If this happens, repeat step (7).

D. WRITING AN XVROUTINE

The information provided in this section concerns the process of writing an xvroutine. This is prefaced by an outline of the steps involved in writing an xvroutine, followed by more detailed information concerning each step in the process. It is *strongly*suggested that you install any new programs in the KHOROS_CONTRIB (or *contrib*) source tree. This will allow you to easily update or install patches to the Khoros system as they become available. Note that if you install new programs in the KHOROS_HOME (or *Khoros home*) source tree, then it will be much more difficult to maintain the Khoros system and install patches. New programs installed in KHOROS_CONTRIB will still be accessable under **cantata**, from the contrib pulldown menu.

1) Creating the *.pane file - Create the UIS file (ie, *.pane file) which will contain the definitions and specifications for the Command Line User Interface (CLUI).

2) Using the ghost routines - The **ghost routines** help to automate writing of the source code and ensure consistency throughout the Khoros system. **Ghostwriter** is a code generator that generates your main driver, include file, library file, and man pages. Note that if your program must link against FORTRAN code, you must specify the [-type fxprog]. The first time **ghostwriter** is run, a template Program Specification (PS) file is created for you, which contains key fields for the documentation and source code. DO NOT DELETE any of the *tags* that **ghostwriter** will insert into your source code and man pages! The **ghost routines** facilitate the writing and maintenance of new programs via this PS file, which, once completed, contains all of the necessary information to re-generate the manual pages and associated source code files at any time, thus making the future addition or deletion of command line arguments a trivial process.

3) Creating the *.form file - Most xvroutines have a separate UIS file (ie, *.form file) that describes their Graphical User Interface (GUI). While in theory, the same UIS file can describe both the CLUI and the GUI of an xvroutine, we have found that the GUI of an xvroutine is generally much more comprehensive than the CLUI, thus prompting the creation of two UIS files: one for use with the **ghost routines** (mentioned in step 1 above), and this one, for use with **conductor**.

4) Using conductor- This step involves running **conductor**, a code generation program. **Conductor** should be run using the **[-b 1]** flag immediately after running **ghostwriter** for the first time. Instructions are given in Chapter 6 of the Khoros Programmer's Manual to integrate the two drivers. **Conductor** is driven by a UIS file (ie, the *.form file) which is usually *not* the same as the *.pane file that is used by the **ghost routines** -- more will be said about this later.

5) Writing the new program - As you add your code and documentation to the files generated by **ghostwriter**, run **ghostreader** occasionally to update your PS file. Documentation and source code will be inserted into the program specification (PS) file between the appropriate keys. Several important points are discussed later in this chapter concerning arguments, error handling, data structures, and memory allocation.

6) Installing the new program & integrating it into cantata- Once the new program is completed to your satisfaction, you must install it in the Khoros Source Tree, and integrate it into the **cantata** visual language. There is currently no program to install xvroutines. They must be installed and integrated by hand. Details on installing your xvroutine are provided later in the following sections.

D.1 CREATING THE *.pane FILE FOR YOUR XVROUTINE

The *.pane file for your xvroutine will be used for two purposes: it will be used as input to **ghostwriter** to generate the command line user interface for your xvroutine, and it will be used to integrate your xvroutine into **cantata**. For the purpose of using **ghostwriter**, it is important to make sure that the *.pane file provides a complete description of all desired program arguments. On the other hand, for the purpose of using the *.pane file to integrate your xvroutine into **cantata**, you must make sure that the *.pane file describes an I/0 pane that will fit into the appropriate subform of **cantata**, that it includes a Help Button, and that it has a Routine button so that the user will be able to access your new xvroutine through **cantata**. No extraneous UIS lines should appear in your *.pane file. Furthermore, it should also present a graphical user interface that is visually pleasing, since the *.pane file is what will be used to integrate your xvroutine into **cantata**.

There are currently 28 different types of UIS lines, some of which are required to appear in every UIS file, others of which are used only by the command line user interface, and others which are used only by the graphical user interface.

There are several **required lines** for all *.pane (and *.form) files. The **required lines** include:

- **StartForm (-F) -** The (-F) line begins the user interface specification. This line contains several fields which contain information concerning the version number of the *xvforms* library, the geometry of the form, and the title of the form among others (the -F line is required in every UIS file).
- **StartSubForm (-M) -** The (-M) line contains information describing the size, position, and title of the subform (the -M line is required in every UIS file).
- **StartPane (-P) -** The (-P) line describes the graphical user interface pane that provides a backplane for selections and action buttons to allow input to the application program (the -P line is required in every UIS file).
- **Help (-H) -** The (-H) line describes a specialized button on the user interface which will bring up an on-line help file when selected. The -H line is required to appear on all **cantata** panes; therefore your *.pane file should contain a -H line that references the man1 page for your xvroutine, for when it is integrated into **cantata**). Note that the -H line is ignored by **ghostwriter**.
- **End (-E) -** The (-E) line is used to end a set of definitions in a UIS file. The (-E) line closes a corresponding definition line in the UIS file. For example, a (-F) line must have a corresponding (-E) line to complete the form definition, a (-M) line must have a corresponding (-E) line to complete the subform definition, and a (-P) line must have a (-E) line to complete the pane definition.

Optional lines for structuring the command line user interface in the *.pane file include:

- **Toggle (-T) -** The (-T) line defines a set of one or more *selection* items of the same type. The value of the toggle group will take on the default value of the selected item (except in the case of a Logical toggle, which takes on the number of the selected item, where logical members of the toggle are numbered starting at 1 with the first item).
- **MutExcl (-C) -** The (-C) line defines a mutually exclusive group of *selections*. Members of the mutually exclusive group may be of the same or different types. The value of the mutually exclusive group will take on the current value of the selected item.

Other lines that may appear in the UIS pane file pertain to *selections*, which are used to identify the inputs, outputs, and program arguments. These include the following:

- **InputFile (-I) -** The (-I) line specifies an *input file selection*.
- **OutputFile (-O) -** The (-O) line specifies an *output file selection*.
- **Integer (-i) -** The (-i) line specifies an *integer selection*.

- **Float (-f) -** The (-f) line specifies a *float selection*.
- **String (-s) -** The (-s) line specifies a *string selection*.
- **Logical (-l) -** The (-l) line specifies a *logical selection*.
- **Routine (-R) -** The (-R) line describes a specialized action button which will appear on the pane when it is integrated into **cantata**. When the user clicks on this button, the xvroutine will be executed (this line is ignored by **ghostwriter**).
- **Blank (-b) -** The (-b) line is used for adding comments and extra titles into the pane, for extra clarity on the pane when it is integrated into **cantata**. Note that the -b line is ignored by **ghostwriter**.
- **Help (-H) -** The (-H) line is used to add a "Help" button to the **cantata** pane that represents your vroutine. The Khoros convention is to provide a "Help" button that accesses the man1 page for your vroutine.

A complete description of each of the UIS lines can be found in Chapter 2, section B.3 of the Khoros Programmer's Manual.

An example of the *.pane file for the xvroutine **warpimage** is presented below. This illustrates the use and placement of the various lines in a UIS pane file.

```
-F 4.2 1 0 170x7+10+20 +35+1 'CANTATA Visual Programming Environment for the KHO
ROS System' cantata
   -M 1 0 100x40+10+20 +28+1 'Warp Images' gis_warp
      -P 1 1 80x38+22+2 +11+0 'Interactive Image Warp' warpimage
      -I 1 0 1 0 0 1 50x1+1+2 +0+0 ' ' 'Input Source Image' 'input source image' src
      -I 1 0 1 0 0 1 50x1+1+3 +0+0 ' ' 'Input Target Image' 'input target image' target
         -T 1 0 1 1 0 40x1+1+5 +0+0 1 'Tiepoint Mode' 'selection type ' tp_mode
            -l 1 0 1 1 0 40x1+2+1 +2+0 0 'Source & Target' 'False' 'True' 'Select Source & Destination' dummy
            -l 1 0 1 0 0 40x1+2+2 +2+0 0 'Source Only' 'False' 'True' 'Select Source Only'dummy
         -E
         -R 1 0 1 13x2+1+13 'Execute' 'do operation in foreground' warpimage
         -H 1  13x2+39+13 'Help' 'man page for editimage' KHOROS_HOME/doc/manpages/warpimage.1
      -E
   -E
-E
```

As in every *.pane file, there are three required lines for the *.pane file. These include the **StartForm (-F)** line for starting a UIS, a **StartSubForm (-M)** line beginning the definition of a subform, and a **StartPane (-P)** line beginning the definition of an I/O pane. Note that each of these lines has a corresponding **End (-E)** line to terminate the respective set of definitions. These lines are all ignored by **ghostwriter**.

Note that the (-F) line is the standard **cantata** (-F) line for all *.pane files, and that the (-M) line is the same as all other *.pane files that will be included in the chosen **cantata** subform, in this case, the "Warp Images" subform. If you would like to look up this example, it can be found in `KHOROS_HOME/repos/cantata/subforms/gis_warp`.

The reader is once again urged to study the detailed discussions provided in in Chapter 2 of the Khoros Programmer's Manual for a complete description of the proper syntax and use of each line in a *.pane file, as they will not be repeated here. An experienced Khoros programmer should be able to look at the *.pane file above, and note that it is a *.pane file for a program named **warpimage**. A quick glance confirms that **warpimage** has two optional input files that will be specified on the command line as [-src {viff image}] and [-target {viff image}], respectively. There is also an optional [-tp_mode] flag that may be specified as 0 (for will be accessed from the **cantata** subform labelled, "Warp Images", on the pane labelled "Interactive Image Warp".

The "warpimage.pane" file also includes a (-H) line describing a help button so that the user may look at its man1 page when they are using **cantata**, and a (-R) line so that the user may enter command line arguments into **cantata**'s pane for **warpimage**, before clicking on the Routine button described by this (-R) line, in order to execute the **warpimage** program. Other info contained in the UIS lines may be looked up when necessary in Chapter 2.

Once the UIS pane file has been created, the next step is to create a driver program for the xvroutine, using all of the arguments specified in the UIS pane file. This is easily accomplished using the **ghost routines** provided for this purpose. The next section describes how to use the **ghost routines** to create a driver program and generate the framework for the library routine.

D.2 USING THE GHOST ROUTINES

Programming is a difficult task that involves many steps between the analysis of the problem and its efficient solution. A huge gap must be bridged between a formal (descriptive) specification of a problem and its accommodation to a given programming environment on a particular machine. There is widespread agreement that the difficulties involved in constructing correct programs can only be overcome if the whole task is broken into sufficient small and formally justified steps following well structured standards. Because of the importance of this, Khoros contains code generators that help to automate part of the procedure of programming.

To get started, create a new directory that can be used as the development directory for the new routine. Create a UIS (*.pane) file using either **composer** or a modification of an existing UIS (*.pane) file, as detailed in Chapters 2-4.

The first tool you will use is **ghostwriter**. However, keep in mind that **ghostwriter** requires a UIS file as input (discussed in the previous sections). **Ghostwriter** can be run with the UIS file to create a template for the PS file, the include file, the source code files for the main driver, the library file, and the manual pages for the program. Alternatively, you can begin with an empty template PS file, fill in the appropriate fields, and run **ghostwriter** to generate the source code and manual page files. We have found that a hybrid method is most popular with the Khoros group; however, after writing one or more programs under the Khoros system, you will find your own preference.

The following example demonstrates the use of **ghostwriter** to generate an xvroutine:

```
% ghostwriter -name xvroutine -type xprog
```

This will generate the files "xvroutine.c", "lxvroutine.c", "xvroutine.1", "lxvroutine.3", "xvroutine.h". Note that in order for **ghostwriter** to run, a valid UIS file (in this case, "xvroutine.pane") must exist in the current working directory. **Ghostwriter** will look for a valid PS file in the current working directory; if one does not exist, a template PS file ("xvroutine.prog") will be created. In addition, **ghostwriter** will also look for a configuration file in the current working directory; if one does not exist, a template configuration file ("xvroutine.conf") will be created as well. The use of the configuration file is explained later in this chapter; for more detailed information about the configuration file, see Chapter 5 of the Khoros Programmer's Manual.

When writing an xvroutine, you must remember to provide the [-type xprog] argument! If your xvroutine links against FORTRAN (note that the *linpack* and *eispack* libraries are in FORTRAN), you should instead provide the [-type fxprog]. If you do not provide the correct [-type] flag to **ghostwriter** for your xvroutine, your code will not compile.

There are several options with **ghostwriter** which allow use of a configuration file, formatting, debug statements, and generation of certain files. The various options may be listed using the [-U 1] option; or, you may wish to run the program with prompts for each command line option using the [-P 1] option. An answer file, which specifies the desired arguments to **ghostwriter**, may be created by using the [-A ghost.ans] and [-P 1] options together, making it easier to perform successive runs of **ghostwriter** (a certainty!).

In this example, if we had used the [-A ghost.ans] option to **ghostwriter**, for the xvroutine, its answer file, "xvroutine.ans", for future executions of **ghostwriter** would look like:

```
-name xvroutine
-config f
-lib t
-man3 t
-install f
-format 1
-debug f
-tag t
-force f
-prog t
-type xprog
```

Now, for successive runs of **ghostwriter**, we need not provide arguments on the command line, or interactively with the [-P 1] option. To execute **ghostwriter** with the exact same arguments as before, all that is necessary is to run:

```
% ghostwriter -a ghost.ans
```

In addition to creating the source files and man files for your xvroutine, **ghostwriter** will also make calls to **imkmf** and **makemake** for you, thus generating (respectively) the appropriate "Imakefile" and "Makefile" files for this program. The xvroutine is now ready to be compiled; of course, lxvroutine() will simply return to the main program (initially, it is generated with only a *return(TRUE)* statement). You may wish to compile at this point, checking for any errors in the driver and include files. To compile, simply type:

```
% make
```

No errors should occur, unless you modify these files in some way; however, if compile errors do occur in any of the source files, you can fix the problems in the problem source files, but remember to follow any modifications with a call to **ghostreader**. Note that **ghostreader** has a subset of the complete set of **ghostwriter** arguments - those arguments that are identical for the two programs have the same meaning, and neither program checks for invalid arguments, so that you may provide the same answer file, "ghost.ans" for both programs. To execute **ghostreader**, then:

```
% ghostreader -name xvroutine
```

or

```
% ghostreader -a ghost.ans
```

The Program Specification (PS) file contains key fields for the insertion of Khoros system documentation, man page documentation, and source code. A template "xvroutine.prog" file is created when **ghostwriter** is run for the first time, containing all of the key fields for constructing a complete PS file. Each key field in the "xvroutine.prog" file is delineated by *begin* and *end keys*. All comments, code, and documentation is placed between the appropriate *begin* and *end tags* for a particular field. An example of a template PS file is presented in Chapter 5. Chapter 5 should be consulted for specific descriptions and guidelines concerning the information to be included in each of the key fields of the PS file. An example illustrating the use and format of each key field of a *.prog file is included in Chapter 5. This format should be adhered to when creating a PS file to ensure maintainability of the Khoros System.

Ghostwriter creates the source code, include files, and man pages with pre-defined **tags**, that are recognized by **ghostreader**. Thus, segments of code and/or documentation between these tags may be modified by the programmer and then reinserted into the PS file by running **ghostreader**. **Ghostreader** takes as input the 5 files: "xvroutine.c","xvroutine.h", "xvroutine.1", and "lxvroutine.3", as well as the old PS file "xvroutine.prog". First it loads in the information provided in the old "xvroutine.prog" file. It then looks for the tags in the source and man files, and pulls any text between a pair of begin and end tags into the corresponding place in the new "xvroutine.prog" file, thus over-writing the old information. In this way, a new PS file may be generated containing the new information.

Running **ghostreader** insures that all modifications are correctly updated in the "xvroutine.prog" PS file; thus, when **ghostwriter** is run again (perhaps with the [-a ghost.ans] flag), the updates will be written to the appropriate files. BE WARNED that if you modify source code or man pages, and forget to run **ghostreader** before running **ghostwriter** again, *all your changes to source code and man pages will disappear!*

D.3 CREATING THE *.form FILE FOR YOUR XVROUTINE

First of all, a short discussion of the use of UIS files is in order. While in theory, the same UIS file can be used for both the command line user interface (CLUI) and the graphical user interface (GUI) of an xvroutine (as it ALWAYS is for vroutines), we in the Khoros group have found that most often, the CLUI requires fewer arguments than the GUI. That is, it is usually appropriate to offer options to the user in the graphical user interface that would not be appropriate on the command line. For example, **editimage** offers only 13 command line arguments. Anyone that has used the **editimage** program knows that the graphical user interface offers dozens more options than that!

To resolve this conflict of interest, xvroutines almost always have *two* UIS files. The first, named "xvroutine.pane", is located with the source code of the xvroutine; it is short, and used with **ghostwriter** as detailed above; in addition, it is used to integrate the xvroutine into **cantata**. The second, named "xvroutine.form", is located in `KHOROS_HOME/repos/xvroutine/`; it is generally much longer and more comprehensive, and is used with the code generated by **conductor** to present the user with a complete graphical user interface.

If you have not already done so, examine your "xvroutine.pane" file. It should have ONLY those arguments that will be input to your program from the command line. However, does it really have everything it needs to make your xvroutine complete? If not, copy your "xvroutine.pane" file to "xvroutine.form". Use your imagination, future plans for the program, and **preview** to extend the "xvroutine.form" UIS file to a point where it offers the user with all the options they will need (that you are willing to implement)! Don't worry if you won't have time to put in all the functionality right away; portions of the GUI can be de-activated (see Chapter 2 of the Khoros Programmer's Manual), or simply not used until later.

While we will not detail all the lines you may decide to have in your *.form file to structure your graphical user interface, some lines you might consider using for the *.form file that you would *not* add to a *.pane file include:

- **SubMenu (-D) -** The (-D) line specifies a *pull-down menu*.
- **StartMaster (-S) -** The (-S) line specifies a *master form*.
- **SubFormButton (-d) -** The (-d) line specifies a *subform button*.
- **MasterAction (-n) -** The (-n) line specifies a *master action button*.
- **StartGuide (-G) -** The (-G) line specifies a *guide pane*.
- **GuideButton (-g) -** The (-g) line specifies a *guide button*.
- **SubformAction (-m) -** The (-m) line specifies a *subform action button*.
- **Workspace (-w) -** The (-w) line specifies a *workspace*.

A complete description of each of the UIS lines can be found in Chapter 2, section B.3 of the Khoros Programmer's Manual.

An example of the *.form file for the xvroutine **warpimage** is presented below. This illustrates the use and placement of the various lines in a UIS form file. Note the difference between the *.form file for **warpimage**, below, and the *.pane file given earlier.

```
-F 4.2 1 1 70x3+10+15 +60+1 'Warp Image' Master
   -M 1 1 70x60+10+5 +60+0  'WARP IMAGE' WarpImage
      -G 1 20x60+0+0 +1+0 ' '
     -w 1000x690+0+5 +25+1 ' ' 'source image' workspace
         -g 1 1 17x1+1+0 'Operations'
     -g 1 0 17x1+1+1 'Input'
     -g 1 0 17x1+1+2 'Output'
     -m 1 0 17x1+1+3 'VIEW TIEPOINTS' 'view tiepoints' view
     -m 1 0 17x1+1+4 'CLEAR TIEPOINTS' 'view tiepoints' clear
     -m 1 0 17x1+1+5 'WARP/VIEW IMAGE' 'warp image ' warp
     -H 1   10x1+125+0 'HELP' 'Master Form Help' $KHOROS_HOME/doc/warpimage/help
     -H 1   10x1+125+2 'Copyright' 'Copyright' $KHOROS_HOME/doc/copyright/copyright
     -Q 1 0 10x1+125+4 'QUIT'
      -E
      -P 1 1 40x60+19+0 +2+0 'WARPIMAGE OPERATIONS' Options
     -T 1 0 0 1 1 20x1+2+1 +0+0 1 'Tiepoint Operation' 'operation' operation
            -l 1 0 1 1 0 40x1+1+1 +0+0 0 'Add' 'False' 'True' ' ' dummy
            -l 1 0 1 0 0 40x1+1+2 +0+0 0 'Delete' 'False' 'True' ' ' dummy
         -E
     -b +27+1 'Zoom Type'
     -a 1 0 14x1+25+2 'Rubberband' 'rubber band zoom' rb_zoom
     -a 1 0 14x1+25+3 'Point & Click' 'point & click' pc_zoom

     -b +53+1 'Zoom Factor'
     -f 1 0 0 1 1 20x1+48+2 +0+0 2.0 2.0 2.0 'Source      ' 'zoom factor' src_zoom_factor
     -f 1 0 0 1 1 20x1+48+3 +0+0 2.0 2.0 2.0 'Destination:' 'zoom factor' dest_zoom_factor
      -E
      -P 1 0 40x60+19+0 +0+0 'WARPIMAGE INPUT' Input
     -I 1 0 0 1 1 1 85x1+0+1 +0+0 ' ' 'Input Source Image ' 'input file' in_src_img
     -I 1 0 0 1 1 1 85x1+0+2 +0+0 ' ' 'Input Target Image ' 'output file' in_dest_img
     -O 1 0 0 1 1 1 85x1+0+3 +0+0 ' ' 'Input Tiepoints    ' 'input tiepoints file' in_tp
      -E
      -P 1 0 40x60+19+0 +0+0 'WARPIMAGE OUTPUT' Output
     -O 1 0 0 1 1 1 85x1+0+1 +0+0 ' ' 'Output Tiepoints       ' 'output file' out_tp
     -O 1 0 0 1 1 1 85x1+0+2 +0+0 ' ' 'Output Coefficients    ' 'output file' out_coeff
      -E
   -E
-E
```

Compare the *.form file for **warpimage**, above, with the *.pane file for **warpimage**, given earlier. See how the *.form file is much more complete, and contains UIS lines describing options not available through the command line. Furthermore, there are many lines in the *.form file that are used to structure the graphical user interface of **warpimage** - examination reveals that the GUI for **warpimage** has no master form; it has a single subform with three guide buttons and three subform buttons on the guide pane, which will appear on the upper left hand corner of the GUI. The guide buttons bring up a "WARPIMAGE OPERATIONS" pane, a "WARPIMAGE INPUT" pane, and a "WARPIMAGE OUTPUT" pane, thus separating the functionality of **warpimage** into obvious categories for the user. These panes will appear to the right of the guide pane. Below the guide pane and the panes will be a workspace on which the source and target images can be displayed.

Note that the (-H) lines in the *.form file for **warpimage** are different than the single (-H) line in its *.pane file. While every *.pane file has a single (-H) line that references the man1 page of the program, a *.form file usually has several (-H) lines - one on the master form, if there is one, one on each subform, and one on each pane. These (-H) lines reference on-line help that is found in `KHOROS_HOME/doc/warpimage/help`; in addition, (-H) lines are used to access the Khoros copyright.

you should have online documentation for your new xvroutine, and you should put it in `KHOROS_CONTRIB/doc/xvroutine/help`, to keep the Khoros contrib source tree parallel with the Khoros home source tree. **Warpimage** is an exception to the many-help-button xvroutine rule: since the panes are small and few, a single Help button to the far right produces an online help page with little buttons across the top that access the entire **warpimage** help directory - by selecting the button labeled with the name of the help file they are interested in, users may read the on-line help for all aspects of **warpimage**.

Use **preview** often when developing your *.form file to make sure that it looks good, and presents options in a readable fashion. Asking co-workers or other potential users of your program for feedback is usually a good way to get an idea of how well you are designing your graphical user interface, and to obtain suggestions and constructive criticism.

D.4 USING CONDUCTOR

When your "xvroutine.form" file is satisfactory, go on to run **conductor** to generate the files "form_drv.c", "form_info.h", "form_info.c", and "form_init.c". If you have never used **conductor** before, you should read Chapter 6 of the Khoros Programmer's Manual, if you have not already done so. The rest of this discussion will assume that you understand the use of **conductor**, that you have already used it with the [-b 1] flag, and that you have integrated the "form_drv.c" file generated by **conductor** with the "lxvroutine.c" file generated by **ghostwriter**, according to the step-by-step instructions that can be found in Chapter 6 of the Khoros Programmer's Manual, on **conductor**.

D.5 WRITING THE NEW XVROUTINE

Writers of xvroutines may want to do some initializations in the "lxvroutine.c" file, but most of the functionality for the xvroutine will go into other files (created by the programmer with appropriate names). Calls to the subroutines and/or functions created in these other files should be made in the "run_xvroutine.c" file, replacing those comments created by **conductor** that say, "PUT YOUR CODE HERE".

It is STRONGLY recommended that you read the man pages on the *verror, vgparm, vmath, vrast,* and *vutils* libraries. These are public Khoros libraries, and contain many useful routines for I/O, math, and reading/writing/checking of images that help the programmer avoid "re-inventing the wheel". Writers of xvroutines are also referred to the man pages on the *xvforms, xvutils, xvgraphics,* and *xvdisplay* libraries. The man pages on *xvforms* will augment Chapter 6 in helping you understand the main user interface loop that was generated by conductor, while the man pages on *xvutils* will inform you about utilities for pop-up error messages, warning messages, and the like. *Xvgraphics* contains an extensive collection of graphics routines, while *xvdisplay* provides routines for displaying VIFF images.

When the source code for library routine is completed, you are reminded (again!) to use **ghostreader** to update your PS files in preparation for any future runs of **ghostwriter**. The program should then be carefully tested for proper operation, using all possible options both separately and in different combinations.

Finally, you will be required to complete the man1 page information for your new program. The purpose of the man1 page is to provide a detailed description of the function of your new program. This should include descriptions of the input files, output files, and arguments to the program. Try to be as complete as possible in your descriptions of the inputs, outputs, and arguments to the program. This information will be of valuable assistance to anyone using your program. All Khoros programs must have a man1 page (after all, you want to let users know of your new program)!

Many people prefer to fill in the appropriate fields in the PS file for the man1 page, and then run **ghostwriter** to update their man1 page properly. Alternatively, you may fill in the man1 page directly, and run **ghostreader** when you are done to update the PS file. You may want to look at some of the man1 files for other Khoros programs to get the feel of how to write one. Man1 files can be found in `KHOROS_HOME/man/man1`. They can be accessed by executing: `% vman {program}`.

Formatting for man1 pages should be done in *nroff* format. This is to ensure that the on-line man page is generated correctly and to maintain consistent formatting throughout the documentation. Certain fields found in the PS file need no formatting since they must consist of only a one line description (notes on the PS file will follow later). Some of the more common *nroff* formatting commands that you may find useful include:

.IP/.LP - for new block paragraphs
.RS/.RE - for indenting
.DS/.DE - for non-formatted sections
.SH - for section headers
.sp # - for adding # blank lines

For an xvroutine, you need not worry about the man3 file at all - remember that "lxvroutine.c" does not really contain a "library" routine - however, you should not delete it, as **ghostreader** will complain if it doesn't exist.

Finally, you should read Section D of this document, labeled, "CONVENTIONS & GUIDELINES FOR ALL KHOROS ROUTINES" for valuable information that may apply to you. In addition, a table of file naming conventions and a listing of variable naming conventions are given there.

D.6 JOURNAL RECORD / JOURNAL PLAYBACK

When you write an xvroutine, it will automatically have the ability to have journal record and journal playback, due to its use of the *xvforms* library; by the same token, it will also be a viable target program for the **concert** distributed user interface collaboration tool. If you would like to use the journal record / playback capability for demos or other purposes, you must first make a journal recording of your program. This is done by executing:

```
% xvroutine -jr xvroutine.jp
```

If your xvroutine has any required arguments, you must provide them as well on the command line; optional arguments may be provided as desired. Remember the command line arguments that you used when recording, as you MUST provide these same arguments when playing back. The program will start up as usual, with no apparent difference. Go through a normal session with your xvroutine, remembering that the journal recording mechanism records your XEvents *in real time*. That is, if you stop in the middle of your recording session to answer the phone, a pause of the same length will be played back at the time when you paused. For the same reason, it is recommended that you make a mental plan of the session and follow it through smoothly while making the recording, especially when making journal recordings for demo purposes; errors, typos, mistakes and pauses will all be recorded.

When you have created your journal recording file, you may then play it back whenever desired. Note that the journal recording/playback mechanism is machine independant, so that you need not be on the same architecture when playing back a journal session as you were when you recorded it. To play back the recorded session of your xvroutine, simply execute:

```
% xvroutine -jp xvroutine.jp
```

Remember that if you provided any command line arguments (required or optional) when you recorded the session, you *must provide the exact same arguments* when playing back the session. If you forget arguments, or substitute others for the original ones, the actions produced by journal playback will be unpredictable. In this case, the playback may or may not work; it may or may not exhibit bizarre behavior, and it may or may not crash your program.

As of Khoros 1.0, when using a window manager that requires interactive placement of windows, you are unfortunately required to map widgets as they appear during a journal playback session. That is, you will have to map the widgets during the playback session as you did when they were recorded. Positioning is irrelevant; however, timing is important. Pay attention to the journal playback session, and map items of the graphical user interface as soon as they appear. Failing to map widgets in a short amount of time may result in events piling up behind you, and happening all at once the moment the widget is mapped. In many instances, this proves quite confusing to the observer. We will attempt to address this problem before the next release of Khoros. Note that if your window manager automatically maps windowsm, this issue should not be a problem for you.

You may also wish to use the **concert** collaboration program to "share" the execution of your program with one or more other people. You may do this by executing:

```
% concert -command xvroutine -d2 display2 [-d3 display3 ... ]
```

For more information on the **concert** distributed interface, see the man page on **concert**, or read Chapter 9 of the Khoros User's Manual.

D.7 INSTALLATION OF THE XVROUTINE

The installation process of a new routine is the process of moving all the files involved in the new program into their proper places in the Khoros source tree, and compiling the program. The layout of the Khoros source tree is explained in detail in Chapter 9 of the Khoros Programmer's Manual - you are encouraged to read this chapter if you have not already done so. The installation process for an xvroutine is not animated like the process for a vroutine; unfortunately, there are too many variations to be easily addressed by a high-level installation program like **kinstall** to handle. Therefore, it must be done by hand.

1) Go to the location of xvroutines in the Khoros Source Tree. This should be `KHOROS_HOME/src/xvroutines` or `KHOROS_CONTRIB/src/xvroutines`. Create a new directory with the same name as your program. Move into the new directory. Copy all source code into the new directory, including the Imakefile, *.c file, *.h file, l*.c file, *.prog file, *.conf file, files generated by conductor, and any other source files you may have created to use with your xvroutine.

2) Execute `% makemake` to create a clean Makefile.

3) Now, if your site supports different computer architectures, you must create a "shadow" of this directory in the srcmach trees (if your site supports only one architecture, skip this step). First, make sure that the directory created in Step (1) is free of all extraneous files. This is important, because once you run **ksrcconf**, any incorrect symbolic links must be removed by hand. Once the directory is clean, you run **ksrcconf** to create the machine-defendant representation of your program. If you are adding your xvroutine to KHOROS_HOME (note that this is NOT recommended, as it will make installation of future Khoros patches difficult), run:

```
% ksrcconf
```

If you are adding your xvroutine to KHOROS_CONTRIB, run:

```
% ksrcconf -mach contrib_mach
```

ksrcconf should respond by listing each location in the srcmach tree that it is making symbolic links. There will be one message for each architecture that is supported by your site. If you go to the srcmach tree, to the location that mimics the location in the source tree where you just copied your source, you should find a matching directory. In this directory you should find a symbolic link to your Imakefile, a Makefile that is particular to the specific machine architecture, and symbolic links to all other files that you copied into the Khoros source tree in Step 1.

4) Now, you are ready to compile your program. If your site supports different architectures, and you are adding to KHOROS_HOME (again, a reminder that this is not generally recommended), execute:

```
% kmakeall install
```

If you are installing in KHOROS_CONTRIB, you must supply the [-mach contrib_mach] flag to **kmakeall**:

```
% kmakeall -mach contrib_mach install
```

Kmakeall will mail you (one mail message for each architecture) with the results of your compile. If there are any errors, they will appear in the mail.

On the other hand, if your site supports only one architecture, you simply execute:

```
% make install
```

and watch the compile. Correct any errors as they occur. In either case, a successful compile should result in the program executable being located in either KHOROS_HOME or

KHOROS_CONTRIB.

5) Now, test your program to make sure that all is well. Don't forget to first execute:

```
% rehash
```

Since you are installing your program for the first time, forgetting to run *rehash* will result in the operating system error, "Command not found". Extensive testing is usually not necessary on installation, as you should have already completed the testing and debugging process in your work area, before installation. However, *never* neglect to do at least cursory testing on your installed program! Sometimes a compile looks successful, but in fact is corrupted due to system problems or corrupted library files. When this is the case, the result is usually an immediate segmentation violation and/or core dump, even before your graphical user interface is mapped to the screen. If this happens, repeat step (4).

6) Move your man1 page (*.1 file) into its appropriate location in either `KHOROS_CONTRIB/man/man1` or `KHOROS_HOME/man/man1`. Make sure that its appearance is correct by executing:

```
% vman xvroutine
```

7) Finally, you may want to integrate your new program into **cantata**. If so, that procedure is detailed in the next chapter.

E. CONVENTIONS & GUIDELINES FOR ALL KHOROS ROUTINES

This section contains information that is common to both vroutines and xvroutines. It details conventions that must be adhered for the sake of portability and machine independance when programming in the Khoros environment. Variable naming conventions that should be followed when creating a UIS file are given here, as are file naming conventions that are followed throughout the Khoros system.

E.1 GUIDELINES FOR WRITING CODE

There are several rules that you should follow when developing new programs under the Khoros environment. Following these guidelines will help to ensure that your new program will function properly and consistently as a new part of the Khoros system.

When coding in the PS file, all *fprintf();* error message statements should use the file descriptor **stderr**, NOT **stdout**.

The *check_args()* function exists in the vgparms library. Its purpose is to make sure the command line did not contain any invalid arguments/options. If the command line did contain invalid arguments/options, *check_args()* will print them out and your program will exit. This call is not mandatory, but it is a good idea to include it in your code.

All input file error checking should be performed in the -include_macros field using functions from the **verror** library. As many #defines may be included as necessary.

The motivation for breaking the program into a main *.c file and a l*.c library file is mainly to separate the functionality into its two component phases. The preliminary work is done in the main *.c file, which includes argument and file checking. The library file includes the source code for executing the vroutine on the data. This enables the library routine to be called from different programs as well as from the original.

In xvroutines, the l*.c file is used to integrate the main driver generated by **conductor** with the files generated by **ghostwriter**. It is not truly a "library" file in that it will not become part of a library, will not be called by any other xvroutines, and will not have a man3 page installed. However, the library file for an xvroutine should still retain its name and the tags generated by **ghostwriter** in order to stay maintainable and consistent under the Khoros system.

An important point which needs to be stated here is that the library routine should be always be declared to return an integer. Consequently it will return zero (0) upon failure or one (1) upon success. The library should be very robust; it should not not crash for any reason. In order to obtain this objective, all the necessary error checking should be done in the libraries as well as their drivers. The library should always return to the main program; either it completed its work successfully or unsuccessfully. That is, the library should never contain an *exit()* statement within its instructions.

Whenever memory is allocated for one or more pointers, they should be freed as soon as they no longer needed. This is to minimize the risk of running out of memory. Also, for those working with VIFF files, the data contained in the memory location starting with the address given by "image->imagedata", should be released with a call to *freeimage()* when it is changed or replaced by another one; for instance, in the following example, "image->imagedata" is replaced by a new pointer.

```
freeimage(image);
image->imagedata = (char *) fptr;
```

If the library routine that is being written needs to call any of the already existing ones, the calling format should be the same as that of the main program; that is, the calling routine should check if the subordinate routine returns one (1) or zero (0) upon success or failure.

Also, one should make an effort not to introduce inter-dependencies into library routines. However, you are welcome (and highly encouraged!) to make calls from your vroutine to the utilities found in the *verror, vgparm, vmath, vrast,* and *vutils* libraries. On the other hand, you are not encouraged to add dspl routines that depend on ipl routines and vice versa.

Due to the necessity to perform the same operation for different image types, one should try to use flow control that provides fast performance. For instance, in programs like vconvert, it is better to have *for* loops inside the *switch* statement rather than having the *switch* inside the *for* loops. The reason for this is that the latter requires more time due to the continuous breaks(jumps).

Also, if the algorithm being implemented requires a large amount of code, modularity is one of the important issues that one should observe. Some kind of compromise with the efficiency issues mentioned above should be considered. When writing a vroutine, one should make sure that all functionality of the program is provided by the code in the library file.

If you make a system call to *signal(sig, function)*, you must define *function* to be of type *vsignal*, and you must have the line: `#include "vsignal.h"` in order for your code to be portable.

If you make a system call to *wait(status)* or *vwait3(status,options,rusage)*, you must define *status* to be of type *vstatus*, and you must have the line: `#include "vsignal.h"` in order for your code to be portable.

If you make a system call to the directory routine *readdir()*, or any other system call that uses the type *dirent* or *direct*, you must re-define the routine that you are calling to be of type *vdirect*. The *vdirect* type is definedin `KHOROS_HOME/include/vinclude.h`. Again, this is to allow your code to be portable to other architectures.

If you need to use the system calls *putenv(), tempnam(), random(), rand(), srandom(), srand(),* or *wait3(),* you must instead use the Khoros versions of these routines that may be found in either the *vgparm* or*vmath* libraries. Once more, this is to ensure portability to other architectures.

If you need to use the system call *vfork()*, then to make your code portable, you must follow the approach in the example below. This is because not all machines support the *vfork()* system call.

```
#ifdef VFORK
        if ((pid = vfork()) == 0)
#else
        if ((pid = fork()) == 0)
#endif
        {
            (void) execl("/bin/csh", "csh", "-cf",
                         "lpr writing_progs.ps", (char *)0);
            exit(1);
        }
```

Since every program is different, it is very difficult to explain exactly how to write one. The best approach for writing a new program is to look at examples. There are plenty of vroutines in the `KHOROS_HOME/src/vipl/Lib/` directory. Spend some time browsing through these routines, looking for one that is similar to the one that you would like to write in order to get an idea of how to approach the problem at hand. Experience with the C programming language will help tremendously, as all Khoros programs are written in C.

Additional information concerning the Program Specification file is provided in Section D of this chapter. Certain segments of the PS file that elaborate on the programming aspects are detailed in that section.

E.2 CONVENTIONS FOR VARIABLE NAMES

These are conventions for the most common variable names used in the Khoros system. Note that the *variable* field on a particular line of the *.pane file becomes the name of the [-flag] to the program on the command line, as well as the name of the corresponding field in the C structure that **ghostwriter** generates in the *.h file.

i single input viff file
i# multiple input viff files
(NOTE: do not use the word "image" casually -
try to use "viff file" when you mean an image
formatted specifically for the Khoros system,
not "image file".

o single output viff file
o# multiple viff output files
f output ascii file

t data type (string) This flag requires a call to *vget_type()*
d direction of processing for 1d data
w,h width and height
x,y coordinates of a pixel
r,c row and column

b bands

a, A, P, U, V reserved command line arguments

E.3 KHOROS FILE NAMING CONVENTIONS

The table below is a list of the file naming conventions used in Khoros.

File suffix	Description
*.a	library
*.ans	Command line answer file.
*.awk	awk script files.
*.c	C source
*.cmdlog	Khoros command execution log file.
*.conf	Configuration file used by **ghost** routines.
*.csh	csh script files.
*.doc	Documentation files formatted for on line help in X applications.
*.f	fortran source
*.form	UIS file describing GUI for xvroutines - used with **conductor**.
*.jp	Journal Playback files.
*.l	lex source
*.man	Files used for preparing documentation for printing.
*.ms	Manual files containing ms macros.
*.o	object file produced by cc
*.pane	UIS file describing CLUI & GUI - used with **cantata** & **ghostwriter**.
*.prog	PS file for **ghostwriter** and **ghostreader**.
*.ps	Postscript files ready to be printed.
*.sh	bourne script files.
*.sub	UIS file describing subforms for **cantata** (has ref's to *.pane files).
*.txt	Text file (README(s) are the exception).
*.vec	Vector font files.
*.xv	Files that are in VIFF format.
*.y	yacc source
*.1	Section one manual page.
*.3	Section three manual page.

F. NOTES ON THE PS FILE

As the Program Specification file is described in Chapter 5 of the Khoros Programmer's Manual, we will not repeat it all here. However, a few extra notes may help the reader. The following notes deal with those segments of the PS file that deal with C programming; descriptions of all the different segments of the PS file may be found in Chapter 5 of the Khoros Programmer's Manual. Note that it is NOT required for you to program in the PS file; it is for this very reason that **ghostreader** is provided - so that the *.c, *.h, l*.c, *.1, and *.3 files may be modified directly, and then **ghostreader** executed, to update the PS file. However, sometimes (especially when updating man pages) it is more convenient to modify the PS file, and run **ghostwriter** to update the abovementioned files. Therefore, selected segments of the PS file are explained in detail below. Again, if you have not already read Chapter 5 of the Khoros Programming Manual as of yet, we urge you to do so now.

-USAGE_ADDITIONS

Ghostwriter will place this code in the *.c file, in the *gw_usage()* routine. **Ghostreader** will take this code from the *.c file, from between the */* -usage_additions */* and */* -usage_additions_end */* keys. This field may or may not be filled in. It is used when the programmer wishes to add more comments to the usage statement. For example, *vbandcomb* uses this field to describe which integer maps to which color space model. If the USAGE_ADDITIONS field had not been used, the usage statement generated by **ghostwriter** would not contain any information as to the legal values for the option that specifies the resulting color space model. If it is not clear as to what to add, it can always be added later. The usage addition statements MUST be in C *fprintf* statements because these additions will be added directly to the main *.c file. After the code is compiled, the programmer can take a look at the usage statement, and modify it if necessary.

-INCLUDE_INCLUDES

Ghostwriter will place this code in the *.h file. **Ghostreader** will take this code from the *.h file, from between the */* -include_include */* and */* -include_include_end */* keys. This field will contain other #include statements that need to appear in the *.h file that are not automatically generated. Note that "unmcopyright.h" and "vinclude.h" are automatically included for vroutines; "unmcopyright.h" and "xvinclude.h" are automatically included for xvroutines (remember that when you are writing an xvroutine, you must provide the [-xprog 1] flag! If you forget, "vinclude.h" will be included incorrectly, and you will not be able to compile your program.

-INCLUDE_ADDITIONS

Ghostwriter will place this code in the *.h file. **Ghostreader** will take this code from the *.h file, from between the */* -include_additions */* and */* -include_additions_end */* keys. This field will contain any additional statements that the programmer may want to insert in the include file. Any #defines, global declarations, structure definitions, etc. should be inserted here.

-INCLUDE_MACROS

Ghostwriter will place this code in the *.h file. **Ghostreader** will take this code from the *.h file, from between the */* -include_macros */* and */* -include_macros_end */* keys. All code inside this field will be within #define statements. In many vroutines, the first step is to read in the input image. To do this it is necessary to use the function called *readimage()*

whose synopsis is :

```
struct xvimage *readimage(filename)
char   *filename;
```

filename is the name of the file; in general, it corresponds to a [-i] flag for the new program, as specified in the UIS file by a [-I] UIS line. *readimage()* will return a pointer to the address where the image is located upon success, or NULL upon failure. Therefore, the pertinent error checking must be performed by the user. Whenever the program exits, it should do so with an appropriate comment. Thus, calls to *readimage()* are a good subject for a macro to be included in the PS file under -INCLUDE_MACROS. For example:

```
#define READINPUT(image) -
image = readimage(vroutine->i_file); -
if (image == NULL) { -
  (void) fprintf(stderr, "vroutine: cannot read input image"); -
  (void) fprintf(stderr, " %s0, vroutine->i_file); -
  exit(1) -
}
```

If there is more than one input image, then each input image should have its own corresponding READINPUT #define statement.

Other error checking can be also performed via macros. Many error checking routines are provided in the *verror* library (the user is encouraged to read the online manual pages for *verror*!). For example, the data type of the image can be tested to see if it is one of the supported by the program. A check for the type of encode scheme may also be needed to make sure that the program will work properly. Whenever two images are read, often the size of both images should match. As an example, in the following macro, the image is be checked to make sure that it has a data storage type of float, only one image is contained in the file, that the image contains 3 data bands and that the map enable is optional.

```
#define CHECKINPUT(program, img1) -
    propertype(program,img1,VFF_TYP_FLOAT,TRUE);  -
    proper_num_images(program,img1,1,TRUE);  -
    proper_num_bands(program,img1,3,TRUE);  -
    proper_map_enable(program,img1,VFF_MAP_OPTIONAL,TRUE);
```

All image error checking should be performed in this manner. As many #defines may be included as necessary.

-MAIN_VARIABLE_LIST

Ghostwriter will place this code in the *.c file, at the beginning of the main() program. **Ghostreader** will take this code from the *.c file, from between the */* -main_variable_list */* and */* -main_variable_list_end */* keys. This field will contain the variable declarations used by the main program. In vroutines that deal with image processing, for example, one necessary declaration is for all images used in the routine. An example declaration might look like:

```
struct xvimage *image;
```

The xvimage structure is explained elsewhere in this document (it is the internal representation of a VIFF file). Any other variables that will be used in the main() must also be defined in this field.

-MAIN_BEFORE_LIB_CALL

Ghostwriter will place this code in the *.c file, in the main() program, after the variable list, but before the library call. **Ghostreader** will take this code from the *.c file, from between the */* -main_before_lib_call */* and */* -main_before_lib_call_end */* keys. This field should contain the C code to get the all necessary input for the program. In addition, any other code that the programmer wishes to appear in the main before the call to the library routine. All vroutines must include a call to *check_args()* here. The call to *check_args()* should have the following syntax.

```
if (check_args()) exit(1);
```

The *check_args()* function exists in the *vgparm* library. Its purpose is to make sure the command line did not contain any invalid arguments. If the command line did contain some invalid arguments, *check_args()* will print them out and your program will exit. REMEMBER: xvroutines MUST NOT make this call!

-MAIN_LIBRARY_CALL

Ghostwriter will place this code in the *.c file, in the main() program. **Ghostreader** will take this code from the *.c file, from between the */* -main_library_call */* and */* -main_library_call_end */* keys. This field should contain the C code to get the all necessary input for It is necessary to pass all the appropriate information to the l*.c file; for example, in a vroutine that was also an image processing routine, it will probably be necessary to pass the pointer of the original image as well as any other parameters that the library might need in order to process the image. In situations where it is not necessary to preserve the input image, one need not pass to the library a new pointer for the resulting image, since the resulting image will be returned through the same pointer as the input image was passed in with. In this scenario, the original image will be lost. The library will return the resulting image of same data type that was originally specified by the user.

If the input image needs to be preserved, a pointer to the resulting image is necessary. Thus, according to the type of problem the user is trying to solve, it will be necessary to build the data structure of a new image. This task should be done prior to making the library call, and then the library routine should be sent the pointer of the new image with all the header information set. To create a new image, see the man page on *vutils*. An example follows:

```
if(! lvroutine(image, vroutine->w_int, vroutine->h_int)) {
   (void) fprintf(stderr, "lvroutine failed");
```

```
    exit(1); }
```

When writing xvroutines, the call to the "library" routine will vary somewhat (see Chapter 6 of the Khoros Programmer's Manual). The main driver that is generated by **conductor** is what actually becomes the l*.c file (ie, the user moves the code generated in "form_drv.c" into the l*.c file). Thus, the library call for an xvroutine might look more like:

```
if (!(lxvroutine(argv, argc, program, xvroutine->i1,
xvroutine->i2)))
    exit(1);
```

Note the lack of *fprintf()* statement when the library routine fails. This is because there is really no need for such a statement in an xvroutine - the appropriate error statement should already have been printed out by this point.

-MAIN_AFTER_LIB_CALL

Ghostwriter will place this code in the *.c file, in the main() program, after the library call. **Ghostreader** will take this code from the *.c file, from between the */* -main_after_lib_call */* and */* -main_after_lib_call_end */* keys. This field should contain the With reference to image processing vroutines: now that all the data processing has been performed, the resulting image needs to be written to the specified file. The function *writeimage()* will perform this task. Its synopsis is:

```
writeimage(filename, imageptr)
struct xvimage *imageptr;
char *filename;
```

where *filename* is the full pathname of the VIFF file, and *imageptr* is a pointer to the structure xvimage of the image.

-LIBRARY_DEF

Ghostwriter will place this code in the l*.c file; **Ghostreader** will take this code from the l*.c file, from between the */* -library_def */* and */* -library_def_end */* keys. it is the definition of your library routine. Depending on whether you are writing a vroutine or an xvroutine, this segment will vary. In either case, you will need to include all the pertinent information that was gathered from the command line. A vroutine library definition might look like:

```
int lvroutine(image, type)
struct  xvimage *image;
int  type;
```

On the other hand, the library definition for an xvroutine will look more like:

```
int lxvroutine(argc, argv, program, type)
int argc;
char *argv[];
char *program;
int type;
```

-LIBRARY_INPUT

Ghostwriter will place this code in the header of the l*.c file, and in the *.3 file. This field should contain a short description of each input to the library routine, with types included. This field may be formatted for nroff. Remember that library inputs should ALWAYS be documented in either the PS file or the *.3 file! If you document library modifications directly into the l*.c file, they will be LOST the next time you run **ghostwriter**.

-LIBRARY_OUTPUT

Ghostwriter will place this code in the header of the l*.c file, and in the *.3 file. This field should contain a short description of each output from the library routine, with types included. This field may be formatted for nroff. Remember that library outputs should ALWAYS be documented in either the PS file or the *.3 file! If you document library modifications directly into the l*.c file, they will be LOST the next time you run **ghostwriter**.

-LIBRARY_MODS

Ghostwriter will place this code in the header of the l*.c file. This field should contain a short description of each new modification to the library routine in the form: name(date) - change. This field may be formatted for nroff. When using the *vi* editor, it is often convenient to use !!date to include the current date. Remember that library modifications should ALWAYS be documented in either the PS file or the *.3 file! If you document library modifications directly into the l*.c file, they will be LOST the next time you run **ghostwriter**.

-LIBRARY_INCLUDES

Ghostwriter will place this code at the very top of the l*.c file. **Ghostreader** will take this code from the l*.c file, from between the */* -library_includes */* and */* -library_includes_end */* keys. This field should contain any #include statements that need to be included in the l*.c file that are not already included by *vinclude.h* (for vroutines) or *xvinclude.h* (for xvroutines).

-LIBRARY_CODE

Ghostwriter will place this code in the body of the l*.c file. **Ghostreader** will take this code from the l*.c file, from between the */* -library_code */* and */* -library_code_end */* keys. This segment contains all the C code for the library routine. If you are like most programmers, and prefer to modify code directly in the lvroutine.c file, then **ghostreader** should be used to pick out the code from the l*.c file and place it in the PS (*.prog) file, after you have finished editing the library code.

G. INTEGRATING THE NEW PROGRAM INTO CANTATA

The final task that remains after installing the new program is to incorporate the program into **cantata**. Prior to doing this, the new program should be tested from both the command line and from the GUI. Simply running the new program from the command line using each option will test the functionality of the program. To examine the GUI for the new program, start up **cantata** and use the 'Workspace' pulldown menu to bring up the 'File Attributes' pane. On the 'File Attributes' pane, enter the name of your *.pane file in the "UIS Filename" parameter box. Once the UIS file is read in, a glyph representing your program will be created you may then be connected up with other glyphs for testing of your routine.

Now, you must add a reference to your new program in the UIS file describing the **cantata** subform on which you want your new program to be accessible from. First, look through the existing directories that may be found in `KHOROS_CONTRIB/repos/cantata/subforms` and decide which one is most appropriate for your program (if desired, you may create a new subform - see Chapter 2 for more details). Move the program's UIS file (*.pane file) to the directory that you have chosen in `KHOROS_CONTRIB/repos/cantata/subforms`.

You must then modify the UIS file in that directory that defines the subform for the main category to which you will be adding your program; there will be only one of these files per directory, and it will be named *.sub, where "*" is the name of the main category of the programs on that subform. To actually incorporate your program into cantata, you must add two lines to this *.sub file; one (-g) UIS line to put a button on the subform that will bring up the pane associated with your program, and one matching (-p) line that will include your *.pane file within the subform definition specified in the *.sub file. Adding these two lines to the *.sub file will cause **cantata** to include your installed and compiled program. Chapter 3 provides a complete discussion on the specific requirements for pane files to be included into **cantata**.

Following is an example of the modifications that would be made to `analysis.sub` to add the routine **dostuff**. In particular, look at the (-g) and (-p) lines that reference the program, "DoStuff".

```
-F 4.2 1 0 170x7+10+20 +35+1 'CANTATA Visual Programming Environment for the KHOROS System' cantata
-M 1 0 100x40+10+20 +23+1 'Data Analysis' analysis
-G 1 20x38+1+2 +2+0 'Choose Selection'
-g 1 1 18x1+1+1 'Remove Interlace'
-g 1 0 18x1+1+2 'Ring Intergral'
-g 1 0 18x1+1+3 'Total Harmonic Distortion'
-g 1 0 18x1+1+4 'Row and Col Sums'
-g 1 0 18x1+1+5 'Boundary Fill'
-g 1 0 18x1+1+6 'Usound Class'
-g 1 0 18x1+1+7 'Do Stuff'
-H 1  18x2+1+14 'HELP' 'guide help' KHOROS_CONTRIB/doc/cantata/subforms/analysis/help
-Q 1 0 18x2+1+16 'QUIT'
-E
-p KHOROS_CONTRIB/repos/cantata/subforms/analysis/vblurlace.pane
-p KHOROS_CONTRIB/repos/cantata/subforms/analysis/vringint.pane
-p KHOROS_CONTRIB/repos/cantata/subforms/analysis/vthd.pane
-p KHOROS_CONTRIB/repos/cantata/subforms/analysis/vrcsum.pane
-p KHOROS_CONTRIB/repos/cantata/subforms/analysis/vfill.pane
-p KHOROS_CONTRIB/repos/cantata/subforms/analysis/vuclass.pane
-p KHOROS_CONTRIB/repos/cantata/subforms/analysis/dostuff.pane
-E
-E
```

Notice the seventh (-g) line, specifying a new guide button called "Do Stuff", and the seventh (-p) line, referencing the new "dostuff.pane" UIS file. These are the two lines that must be added to the `analysis.sub` UIS file in order to make **cantata** include the new **dostuff** program. Once the (-g) and (-p) lines referencing the new routine have been added to the appropriate subform, **cantata** can now access the new routine. It is imperative that you add the (-g) line and the (-p) lines consistently; that is, if you add a 7th (-g) line to the subform definition, you take care to add your (-p) line as the 7th (-p) line in the subform definition, and *not* the 6th or 5th. This is because the *xvforms* library determines which subform button will map which pane *strictly by order of appearance*. Furthermore, make sure that your (-g) line has a geometry string that is NOT identical to any of the others. Geometry strings that are identical cause subform buttons to be mapped directly on top of each other, and in this case your subform button will not appear on the **cantata** subform; even though it will exist, it will be underneath another button, and therefore invisible.

The above procedure is the simplest way to add new programs to **cantata**. As you become more familiar with the tools in Khoros, you will discover that it is not that difficult to completely reconfigure **cantata** for different application areas.

H. MAINTENANCE OF NEW PROGRAMS

To fix bugs in new and existing programs, it is wise to create a work directory where bug fixes may be performed. A tool called **kdinstall** is available to help copy all the files associated with a routine to your work directory. The **kdinstall** routine does not remove files from the source tree directories, it just copies them to your work directory. The way to use **kdinstall** is to change directories to the location of the current config file for the routine to de-install, and execute the **kdinstall** program. The syntax for the **kdinstall** program is as follows:

```
% kdinstall -name xxxx -dest yyyy [-mach zzzz] [-force t]
```

where,

xxxx is the name of the routine to de-install.

yyyy is the destination path for the files. This is typically a directory in your work area. Note that if the directory does not exist, then the program will prompt you to create the directory.

zzzz is the path and filename of the mach file that is being used for the current de-install. The default mach file will be used if none is provided.

t is a boolean that specifies whether to use the force option which bypasses the overwrite prompts and edit sessions. The default is FALSE, which does not bypass the overwrite prompts and edit sessions. Note that the force option of TRUE is not allowed if the vroutine has never been installed, or has been removed from the system.

All bug fixes in new and existing programs can be performed using either the source code (*.c) files or the program specification (*.prog) file. If modifications are made to the source code files, then **ghostreader** should be run to incorporate the changes into the program specification file. On the other hand, if changes are made to the program specification file, then **ghostwriter** should be run to generate new source code files. If you are unsure which of the ghost routines to run, simply run **ghostcheck** to provide you with some assistance. Remember to copy the library source and manual pages to the appropriate directories if you did not use the "-config" option of **ghostwriter**.

Once the bug fixes have been made, you must then re-install the program. This is most easily accomplished by running the **kinstall** program, which will correctly install the bug fixed routine over the existing routine in source and recompile the routine. If multiple architectures are supported, then it will rsh to the other machines and recompile for that architecture. To run **kinstall** refer to the instructions given in Section B.4.2 of this chapter.

Volume II

Khoros Programmer's Manual

Chapter 1

VIFF

Khoros Visualization/Image File Format

Primary Author(s):

John Rasure, Scott Wilson, Tom Sauer

Printed: May 9, 1991

VIFF

I. VISUALIZATION/IMAGE FILE FORMAT (VIFF)

The Khoros Visualization/Image File format and internal data structure format (the *xvimage structure* - see `KHOROS_HOME/include/viff.h`) was designed to facilitate the interchange of data, not only between data processing functions within Khoros, but also between researchers that want to exchange data and information. VIFF was designed to be comprehensive; image processing is an extremely diverse field with different applications requiring different information about an image. The VIFF header includes information necessary for an application program to properly interpret the data and perform basic error checking. The VIFF data structure has been applied to both 1D and 2D data processing applications and 3D visualization. The VIFF format was also designed to be extensible; since new applications and new methods of processing image data are sure to be developed in the future, the file format can be extended by adding new fields. There is currently plenty of reserve space in the 1024 byte header, or the spare fields can be used.

I.1 VIFF FILE ORGANIZATION

The Khoros Image File Format is organized as 1024 bytes of header followed by map(s), location data and then image data. The header also contains information identifying the header format used. This file format (while on disk) or data structure (struct *xvimage, while in memory) has evolved from originally supporting only *images* to supporting multi-dimensional data. The data may be an image in the normal sense, or it may be sets of vectors. The difference is indicated by carefully examining the values of the various fields. Information on an existing file can be obtained by using **vfileinfo**.

The fields can be grouped into four categories:

- administration - file management information,
- data storage - describes how the data is stored but not how it is to be interpreted,
- location data - describes the spatial location of the data (this is optional),
- data mapping - describes how the stored data should be mapped or interpreted (no mapping is also valid),
- color space - in the case of images, indicates what coordinate space and model is being used.

Please refer to the table in section C.4 and the examples that follow for specific information about all fields in the header.

I.2 FILE FORMAT TERMINOLOGY

I.2.1 Image & Imagedata

There are several working terms that must be defined for the context of the VIFF file format discussion. An *image* is a two-dimensional, multiband array of data with either implied or explicit data locations. Note that while the pointer to the image data (*imagedata*) is a single dimensional array, *conceptually,* the image should be thought of as two dimensional. It is important to remember that the VIFF image is oriented so that (0,0) is at the upper left hand corner of the image. The column index, X, proceeds across the image to the right; the row index, Y, proceeds from the top of the image to the bottom.

I.2.2 Pixels, Bands & Vectors

At each location, implied or explicit, there is a *pixel* that may have one or more values. If there is more than one value at a pixel location, for example in an RGB image, these data values are separated into *bands*. This means that an image can have one or more bands; these bands are stored sequentially. For instance, in an RGB image, the Red band is followed by the Green band, which is followed by Blue band. Taken as a group, the three pixel values describe a single color value.

For multidimensional data, the concept of a *vector* may be more appropriate. Vectors are stored exactly like images, except that the number of bands is interpreted to mean the *vector dimension*. Vectors can have either implied or explicit location data, just like the pixels of an image.

I.2.3 Implicit VS. Explicit Locations

When we refer to the *implied location* of a pixel, we refer to the fact that imagedata[x * row_size + y] = pixel intensity value, where row_size is the width of the image (ie, number of columns); here, x is the horizontal index of the pixel, and y is the vertical index of the pixel; because the x and y values in this image are implied, we call this type of image *implicit*. When an image is implicit, there is no need for location data, as this would be redundant. A one dimensional image with *implied locations* data can be represented by setting either the column size or the row size to one (1).

A VIFF image can have *explicit location* data, where each pixel can have an N dimensional location in space. This concept can be explained as the imagedata value being an attribute at some location in N-dimensional space. To specify an attribute in two dimensional space, for example, we explicitly store the values of x and y, as opposed to allowing their values to be implied. To access the x value, we look up location[i * row_size + j]; to access the y value, we would up location[(i * row_size + j) + row_size * col_size]. Here, i is the horizontal index of the image and location data, j is the vertical index of the image and location data; row_size is again the width of the image (ie, number of columns) and col_size is the height of the image (ie, number of rows). As in the *implicit* image, the imagedata value is located at imagedata[i * row_size + j]. For an image, the location dimension would be two. A typical use of this interpretation is digital elevation data, where the image consists of elevation values; the two dimensional location data provides the X and Y spatial coordinates.

The *explicit* N dimensional location information is completely independent of the vector dimension or the number of bands. For multiband data with explicit locations, the concept can be understood as the multiband imagedata values being a set of attributes at some location in N-dimensional space.

I.2.4 An Aside on One Dimensional Data

Given the above description, it is unclear how best to represent sets of 1D data. There are actually two interpretations that make sense and are supported in the digital signal processing library (dspl). A single one dimensional implicitly located data set can be stored as a single band with the product of row size and column size giving the number of elements. Alternatively, it may be represented as a single pixel with the number of bands giving the number of elements (row size=col size=1 in this case). A group of 1D data sets can then be stored as a set of bands, where the band number can indicate either the length of the data set or the number of data sets. Similarly, the row_size times the col_size can indicate the length of the data set or the number of data sets. These two interpretations lead to the idea of processing *down vectors* or processing *across bands*.

A multiband image may have several spatial organization interpretations while it is being processed in Khoros. A multiband image may first be processed down 1D vectors and later may be processed as a multiband 2D image.

I.2.5 Image Maps, Map Types, Map Enable and Map Schemes

Any image may contain *maps*. A map provides additional information associated with the imagedata. Most often, the map field is used to provide color information, but is not limited to the storage of color information only. The size of a map is defined by *map_row_size* (the number of columns in the map) and *map_col_size* (the number of rows in the map). Any pixel value found in the imagedata serves as an index into one of the map rows. The number of columns in that map row determines the number of pieces of information that are described by the map, and accessed via the pixel value. For example, when the map is used to store RGB colors, a pixel value of 5 found in the imagedata will index row 5 of the map. At row 5, three values will be found: one each for the red, green, and blue color components of pixel value 5.

There are two different *map types* that are set with the *map_enable* field. The first is *optional*, and the second is *forced*. If the map_enable field is set to optional, this indicates that the imagedata is valid by itself, or is valid after indexing into the map.

If the map_enable field is set to force, the image data is NOT valid until it it indexed into the map. When the map_enable field is set to forced, this implies that the imagedata itself is not meaningful; it serves only as an index into the map, where the actual information is stored.

The *map scheme* defines how the maps are to be used with the image(s). If the *map scheme* is one-per-band, then each band in a multiband image will have its own map (images with only one band should always have a map scheme of one-per-band. If, on the other hand, the *map scheme* is shared, then all data bands will share a single map. If the *map scheme* is cycled, then a single band image uses multiple maps. This could be used in an animation sequence of a single scene; for example, it would be appropriate for the animation of a sunset sequence of different colors in a single band of an image.

I.3 VIFF HEADER POINTERS

When the data is in memory, pointers that are part of the xvimage structure are used to address the data. The *vutils* library provides basic utilities to read and initialize the pointers. A brief description of each of the pointers in the VIFF format is provided below:

char *imagedata

Points to a sequence of images; an image is made up of bands, and the bands are in a sequence. Alternatively, imagedata can point to vectors, where the vector dimension is the number of bands and the vector elements are across the bands.

char *maps

Points to a sequence of 2-dimensional maps; a map is organized as stacked columns. A imagedata data value indexes into a map row.

float *location

Points to bands of coordinate values. For example, if you are using two dimensional explicit locations, then there would be a band of x's followed by a band of y's. The number of locations or the number of pixels in a band is always the product of row size and column size. The 1D interpretation mentioned above does not require that either row size or col size be set to one. The convention is that the length of a 1D signal is the product of the two.

To convert other types of image files into VIFF format for use in Khoros, use one of the several file format conversion routines (see the section on File Formats in Chapter 8 of the Khoros Programmer's Manual, or the man pages on file_formats).

I.4 VIFF HEADER FIELDS

Khoros Image File Format Header Fields		
Field	**Description**	**Defines**
char identifier;	A one byte magic number that identifies a VIFF file	XV_FILE_MAGIC_NUM
char file_type;	A one byte code indicating Khoros file type (currently only VIFF).	XV_FILE_TYPE_XVIFF
char release;	A one byte code indicating the specific release of the viff.h file (currently 0); this does not have to agree with the Khoros system release number.	XV_IMAGE_REL_NUM
char version;	A one byte code indicating the specific version of the viff.h file (currently 3); this does not have to agree with the Khoros system version number. For example, version 1 release 0 is referred to as viff 1.0.	XV_IMAGE_VER_NUM
char comment[512];	A 512 byte space available for any use, but currently is used in Khoros as a comment field to document the VIFF data file.	none
char machine_dep;	A one byte code indicating how the particular architecture that the image was last processed on treats data. Currently supported are DEC order, IEEE order and NS order. Supported machines include the VAX, SUN, SONY News, Silicon Graphics, Motorola, Encore, Sequent, MIPS, DEC, IBM, Apollo, and NeXT.	VFF_DEP_IEEEORDER VFF_DEP_DECORDER VFF_DEP_NSORDER VFF_DEP_BIGENDIAN VFF_DEP_LITENDIAN

Khoros Image File Format Header Fields		
Field	**Description**	**Defines**
unsigned long row_size, col_size;	These two unsigned long fields indicate data size, specifically, the number of data items in a band. row_size indicates the length of a row (the number of columns, or the image width) in pixels. col_size indicates the length of a column (the number of rows, or the image height) in pixels. Images with row and column sizes of zero are sometimes valid, and indicate that only the map information is important. The product of the two values is the total number of data items present.	none
unsigned long subrow_size;	This unsigned long field specifies the length of any subrows in the image. This is useful when one wants pixel vectors to represent 2D objects (images). The size of each pixel "image" would be subrow_size (columns) by num_data_bands/subrow_size (rows). This field may be ignored except by routines that need the 2D interpretation.	none
unsigned long startx, starty;	These two long fields indicate the location of a subimage in a parent image. The image is a sub image if startx and starty have values greater than zero. startx and starty locate the upper left hand corner of where the image was extracted. This applies to 2D data that has implicit locations.	VFF_NOTSUB
float pixsizx, pixsizy;	These two floats specify the actual pixel size in meters at the time of digitization. This information is needed to do true measurements and calculate true frequencies. The ratio of these fields will give you the aspect ratio of the digitized pixel. Most CCD camera's are not one-to-one. These values may not have a meaning if the data is VFF_LOC_EXPLICIT.	none

Khoros Image File Format Header Fields

Field	Description	Defines
unsigned long location_type;	This unsigned long field indicates whether the image data has implicit or explicit locations. If the locations are implicit, the field location_dim must be set to zero (0), and the location data will be empty. If the locations are explicit, the field location_dim indicates the dimensionality of the space (1D, 2D and 3D will be most common). The explicit location data is pointed to by location, and is stored as bands of coordinates. For example, if (location_dim = 2), implying 2D locations, the location data would be stored; x1, x2, . . . , xn; y1, y2, . . . , yn.	VFF_LOC_IMPLICIT VFF_LOC_EXPLICIT
unsigned long location_dim;	This unsigned long indicates the dimensionality of the location space of data. Zero-, one-, and two-dimensionalities will be most common. If the location_type is implicit, this field must be set to zero (0). If the location_type is explicit, this field is set to the dimensionality correctly describing the data. Remember that 3-dimensional image data is represented with location_dim = 2 (x & y); the third dimension (pixel intensity) is stored in the imagedata.	none
unsigned long num_of_images;	This unsigned long indicates the the number of images (not bands) pointed to by *imagedata.	none
unsigned long num_data_bands;	This unsigned long indicates the number of bands per image or the dimensionality of vectors. In some cases, it may be convenient to think of an image pixel as a vector (when there is more than one band).	none

Khoros Image File Format Header Fields		
Field	**Description**	**Defines**
unsigned long data_storage_type;	This unsigned long field indicates the data storage type of the pixel or vector data. Currently, bit, byte, short, integer, float, float complex, double, and double complex data types are supported. The BIT storage type stores the bits in packed unsigned chars and pads to a byte; the order is Least Significant Bit (LSB) first.	VFF_TYP_BIT VFF_TYP_1_BYTE VFF_TYP_2_BYTE VFF_TYP_4_BYTE VFF_TYP_FLOAT VFF_TYP_COMPLEX VFF_TYP_DOUBLE VFF_TYP_DCOMPLEX
unsigned long data_encode_scheme;	This unsigned long field contains the information that specifies the encoding or compression method used for storage of the data. Only the raw and compress encoding schemes are supported in Khoros 1.0.	VFF_DES_RAW VFF_DES_COMPRESS VFF_DES_RLE VFF_DES_TRANSFORM VFF_DES_CCITT VFF_DES_ADPCM VFF_DES_GENERIC
unsigned long map_scheme;	This unsigned long field specifies the type of mapping that should occur. The map is an array, where the data pointed to by *imagedata is used as an index into rows of the array. ONEPERBAND indicates that each data band has a map. CYCLE is for when a single band is displayed by cycling through maps. SHARED is for when there is only one map for all bands to share. The GROUP mapping scheme is for future Khoros development, and should not be used as of Khoros 1.0.	VFF_MS_NONE VFF_MS_ONEPERBAND VFF_MS_CYCLE VFF_MS_SHARED VFF_MS_GROUP
unsigned long map_storage_type;	This unsigned long field indicates the type of data in the map. Data in the map may be of type char, short, integer, float, or complex. This is also the resulting data type after a mapping has been done.	VFF_MAPTYP_NONE VFF_MAPTYP_1_BYTE VFF_MAPTYP_2_BYTE VFF_MAPTYP_4_BYTE VFF_MAPTYP_FLOAT VFF_MAPTYP_COMPLEX

Khoros Image File Format Header Fields

Field	**Description**	**Defines**
unsigned long map_row_size, map_col_size;	These two unsigned long fields indicate the number of rows in the 2D map and the number of columns in the map. The maps are stored as a sequence of columns (stacked columns).	none
unsigned long map_subrow_size;	This unsigned long field specifies the number of subrows in a map. This is useful when using the output vector from the map is a 2D image, rather than just a vector. The size of the 2D image would be: map_subrow_size (columns) by map_row_size/map_subrow_size (rows). This field may be ignored except by routines that need the 2D interpretation.	none
unsigned long map_enable;	This unsigned long field specifies if the image data will be valid regardless of whether or not it has been sent through a map. In some cases the mapping is optional (VFF_MAP_OPTIONAL), while in others, the data must be mapped in order for it to have a valid meaning (VFF_MAP_FORCE). See the previous section on FILE FORMAT TERMINOLOGY for a more detailed explanation of the map_enable field.	VFF_MAP_OPTIONAL VFF_MAP_FORCE
unsigned long maps_per_cycle;	This unsigned long field specifies the number of maps that would constitute one cycle when the cycled map scheme type is used. Of course, this field is only valid when map_scheme is set to VFF_MS_CYCLE.	none

Khoros Image File Format Header Fields

Field	Description	Defines
unsigned long color_space_model;	This unsigned long field indicates the color space or the coordinate system being used to interpret the image bands. The color space model defines use the following convention: NTSC: National Television Systems Committee CIE: Commission Internationale de L'Eclairage UCS: Universal Chromaticity Scale RGB: Red Component, Green Component, Blue Component CMY: Cyan Component, Magenta Component, Yellow Component YIQ: Luminance, I and Q represent chrominance HSV: Hue, Saturation & Value IHS: Intensity, Hue & Saturation Most color map models only make sense when the map_scheme is set to VFF_MS_ONEPERBAND and the map_row_size = 3 and the num_data_bands = 1; alternatively, the map_scheme may be set to VFF_MS_NONE and the map_row_size = 0, and the num_data_bands = 3. The exceptions to this are: VFF_CM_NONE indicates that no color space model has been assigned, VFF_CM_GENERIC indicates that a color space is valid, but is being defined by the user. VFF_CM_genericRGB would imply that VFF_MS_ONEPERBAND is set with map_row_size = 3 and num_data_bands = 1. genericRGB is an RGB image but doesn't conform to a standard.	VFF_CM_NONE VFF_CM_ntscRGB VFF_CM_ntscCMY VFF_CM_ntscYIQ VFF_CM_HSV VFF_CM_HLS VFF_CM_IHS VFF_CM_cieRGB VFF_CM_cieXYZ VFF_CM_cieUVW VFF_CM_cieucsUVW VFF_CM_cieucsSOW VFF_CM_cieucsLab VFF_CM_cieucsLuv VFF_CM_GENERIC VFF_CM_genericRGB
unsigned long ispare1, ispare2; float fspare1, fspare2;	Spare fields available for user defined fields. These fields are not supported except for reading and writing correctly with respect to machine dependencies.	none

Khoros Image File Format Header Fields		
Field	**Description**	**Defines**
reserve	Reserve space is allocated so that the total length of the header is 1024 bytes and so that more fields can be added.	none
char *maps;	This is a pointer to the beginning of the map data. When it is used, it must be cast to the proper data type.	none
float *location;	This is a pointer to the beginning of the location data. The dimensionality of the location data is given in the field location_dim. (location_dim specifies the number of bands of coordinates.) The number of coordinate values per band is row_size*col_size.	none
char *imagedata;	This is a pointer to the beginning of the image data. When it is used in an image processing routine it must be cast to the proper data type.	none

I.5 EXAMPLES USING THE VIFF FORMAT

The table above succinctly describes each field and its meaning, but it is helpful to have an example to follow to reinforce the ideas. The following narrative describes the use of the most difficult fields (location_type, location_dim. num_data_bands, map_scheme, map_row_size, map_col_size, map_enable and color_space_model) by tracking an image through a typical Khoros image processing application .

You will probably begin your image processing task by acquiring an image using an eight bit digital to analog converter. When this image is first brought into the Khoros, it would usually be declared as:

Khoros Image File Format Header Fields

Field	**Defines**	**Description**
location_type	VFF_LOC_IMPLICIT	a normal digitized image
location_dim	don't care	
data_storage_type	VFF_TYP_1_BYTE	8 bit data
num_data_bands	1	one band per image
map_scheme	VFF_MS_NONE	no mapping
map_row_size	don't care	
map_col_size	don't care	
map_enable	VFF_MAP_OPTIONAL	don't need to map
color_space_model	VFF_CM_NONE	no color model

Next, you probably will process this image with several of the Khoros programs. As you do this, you are not likely to change the above fields but may change the data_storage_type field from VFF_TYPE_BYTE to some other type.

As you view the resulting image with editimage, you may want to *enhance* the image. This will change the fields so that the data is interpreted correctly. For instance, if you add "color" to your grey image and save the result, the header fields should be:

Khoros Image File Format Header Fields

Field	Defines	Description
location_type	VFF_LOC_IMPLICIT	a normal digitized image
location_dim	don't care	
data_storage_type	VFF_TYP_1_BYTE	8 bit data
num_data_bands	1	one band per image
map_scheme	VFF_MS_ONEPERBAND	one map
map_row_size	3	map pixels through three columns
map_col_size	256	256 rows in the map
map_storage_type	VFF_MAPTYP_1_BYTE	8 bit map result
map_enable	VFF_MAP_OPTIONAL	don't have to map
color_space_model	VFF_CM_genericRGB	pseudo colored

The displayed image may be enhanced even further by adding overlays to the image. The displayed overlay and image are saved by executing getimage (vman getimage). getimage actually does a screen dump into a VIFF file. If the screen is an eight plane display, then the resulting fields are:

Khoros Image File Format Header Fields

Field	Defines	Description
location_type	VFF_LOC_IMPLICIT	a normal digitized image
location_dim	don't care	
data_storage_type	VFF_TYP_1_BYTE	8 bit data
num_data_bands	1	one band per image
map_scheme	VFF_MS_ONEPERBAND	one map
map_row_size	3	map pixels through three columns
map_col_size	256	256 rows in the map
map_storage_type	VFF_MAPTYP_1_BYTE	8 bit map result
map_enable	VFF_MAP_FORCE	must be mapped!
color_space_model	VFF_CM_genericRGB	RGB interpretation

This means that the image data only has meaning if it is mapped through the three color map columns.

Let's return to the image as it was when it was first brought into the Khoros and look at vector data representations. You may use a Khoros routine that extracts features of an image at sparse, explicit locations. This is best stored as vector data.

Khoros Image File Format Header Fields

Field	**Defines**	**Description**
location_type	VFF_LOC_EXPLICIT	explicit locations
location_dim	2	x and y locations
data_storage_type	VFF_TYP_FLOAT	real numbers
num_data_bands	7	each vector has seven elements
map_scheme	VFF_MS_NONE	no map
map_row_size	don't care	
map_col_size	don't care	
map_storage_type	don't care	
map_enable	VFF_MAP_OPTIONAL	don't have to map
color_space_model	VFF_CM_NONE	

In the case that you want to have a vector for each pixel location in the image and also want a color map for each vector band (elements), then the fields should be specified as follows:

Khoros Image File Format Header Fields

Field	**Defines**	**Description**
location_type	VFF_LOC_IMPLICIT	a normal digitized image
location_dim	don't care	
data_storage_type	VFF_TYP_1_BYTE	8 bit data
num_data_bands	3	three bands per image
map_scheme	VFF_MS_ONEPERBAND	one map per band
map_row_size	1	map vector bands through one column
map_col_size	256	256 rows in the map
map_storage_type	VFF_MAPTYP_1_BYTE	8 bit map result
map_enable	VFF_MAP_OPTIONAL	don't have to map
color_space_model	VFF_CM_ntscRGB	red, green, blue

This is an example of a typical RGB image.

CONTENTS

Chapter 1 - WRITING PROGRAMS / VIFF FORMAT