

# Toolbox Programming

# *Toolbox Programming*

## **Chapter 1**

### **Introduction**

#### **Overview of the VisiQuest Software Development System**

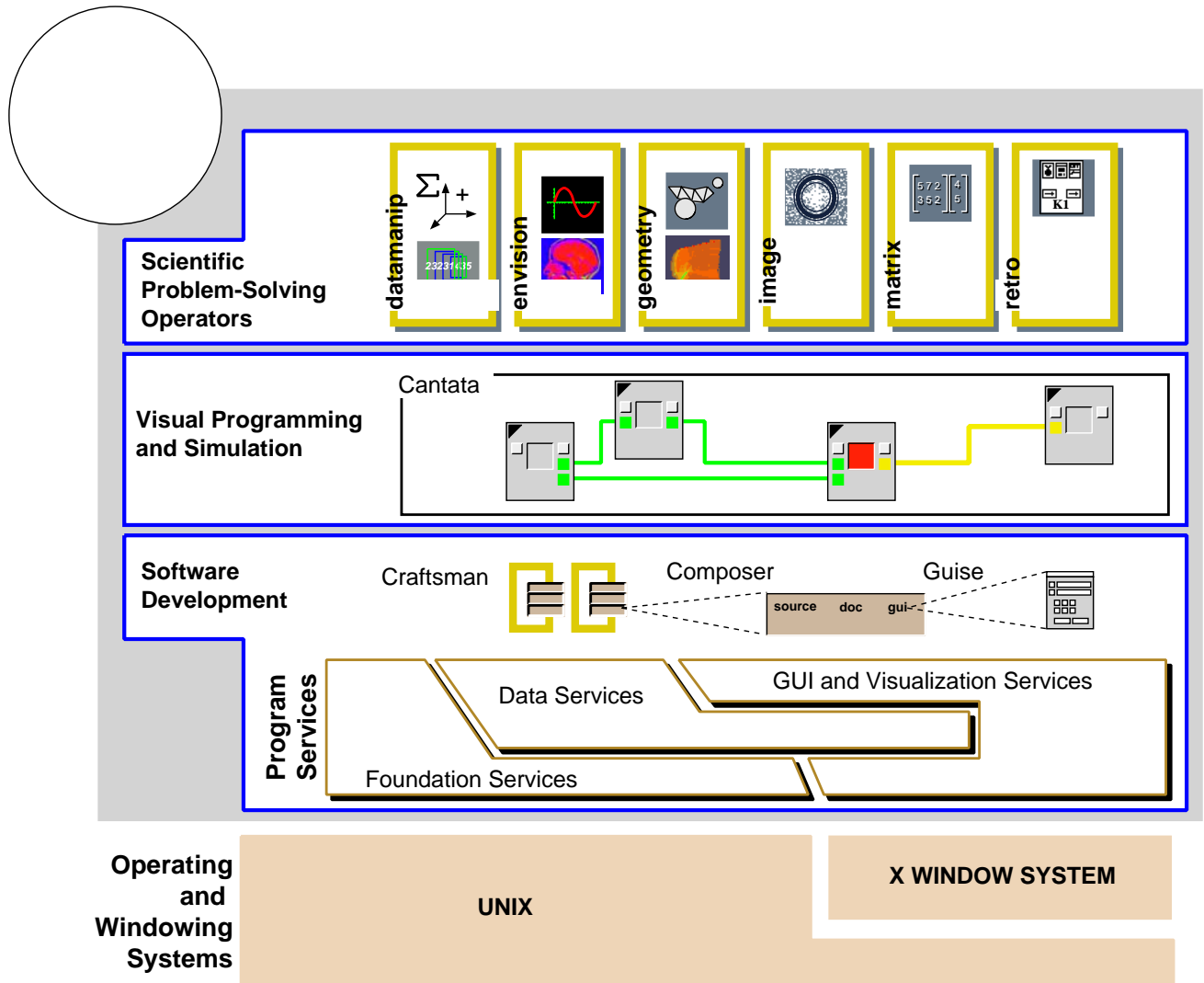
Copyright (c) AccuSoft Corporation, 2004. All rights reserved.



# Chapter 1 - Introduction

## A. Introduction

You may interact with VisiQuest 2001 on one or more levels. Each level represents a different interaction or level of programming. For example, an application user who wants to visualize a 2D data set will interact with the system at a different level than a developer who will actually implement an imaging application or a data processing routine. Figure 1 below depicts the various programming or "user interaction" levels available within the VisiQuest 2001 development environment. Depending on the task at hand, the level at which you work with VisiQuest 2001 will vary. VisiQuest users that interact with the system at the lowest level are referred to as "toolbox programmers", hence the name of this manual.



**Figure 1:** The VisiQuestPro 2001 system can be accessed at different levels, depending on the problem to be solved and the software development expertise of the user. At the highest level are the scientific and data manipulation operators, at mid-level is the visual programming environment, and at a lower level is the

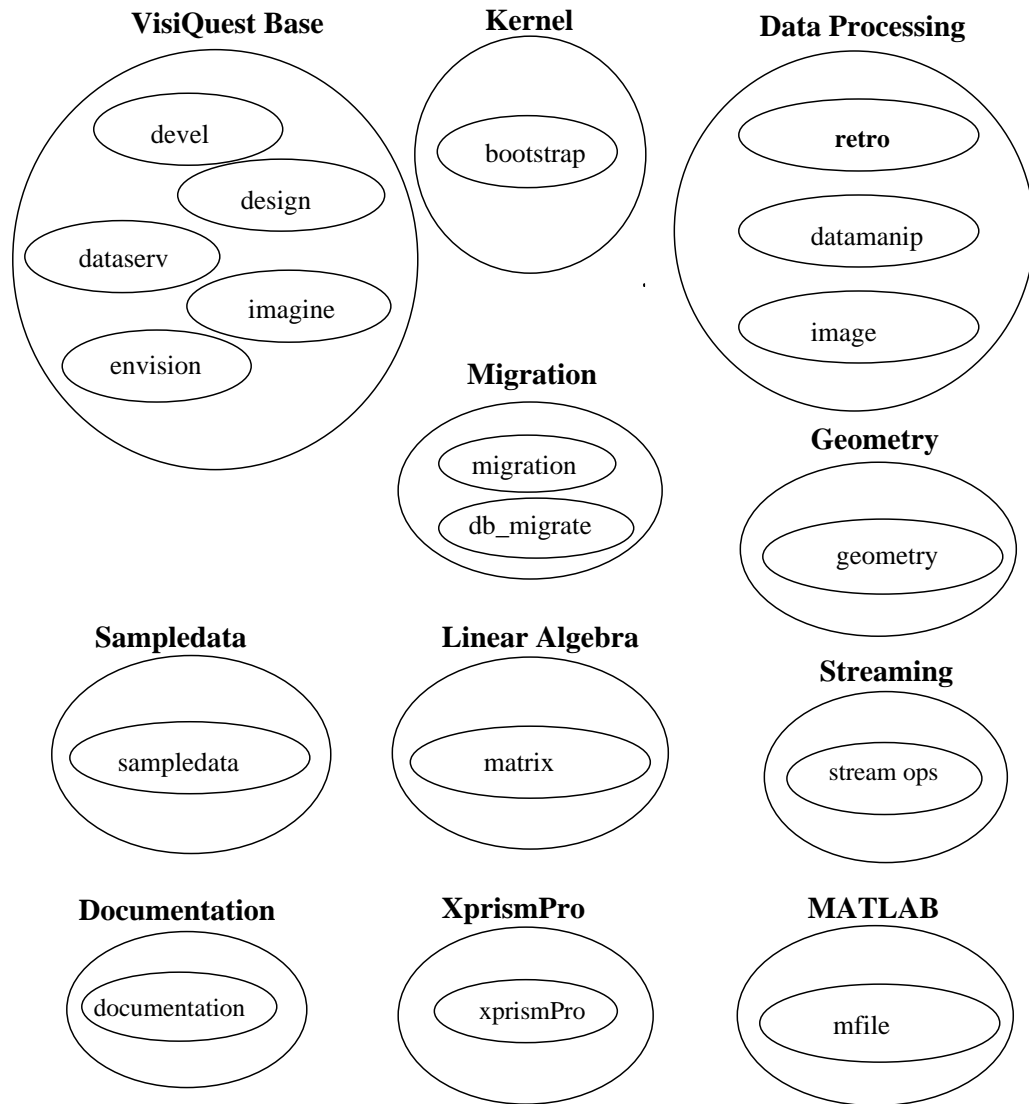
software development environment. The API's offered by the VisiQuest Program Services libraries are at the lowest level of the system.

---

This chapter will provide the toolbox programmer with an introduction to the software development tools available to facilitate the development of programs and applications within VisiQuest 2001. The VisiQuest 2001 software development tools provide a complete environment which supports the iterative process of developing, maintaining, delivering, and sharing software. These tools provide automation when possible, enforce consistency as necessary, and hide underlying complexity of software configuration, code generators, and documentation formatters.

Each program and library in VisiQuest is contained within a *toolbox*. A toolbox is a collection of programs and libraries that are managed as an entity. A toolbox imposes a predefined directory structure on its contents to provide consistency and predictability to software configuration. Typically, a toolbox contains programs and libraries that are characteristic of a given application domain. For example, programs that perform image processing operations might be contained within one toolbox, while programs that perform signal processing operations are contained within another toolbox.

## VisiQuest Software System Packages




---

**Figure 2:** The packages that make up the VisiQuest Pro system include Kernel, Base, Data Processing, Sample Data, Geometry, Linear Algebra, Documentation, XprismPro, Streaming, and Migration. Each package contains one or more toolboxes.

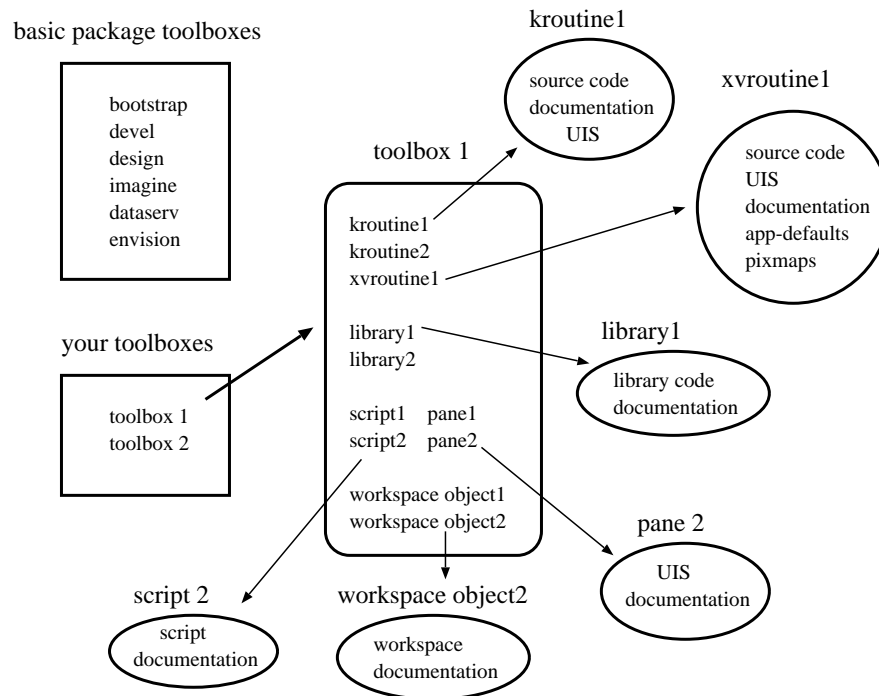
---

A *toolbox object* is an encapsulation of a number of pieces of software; similarly, the different pieces of software in the toolbox are also considered objects. A *software object* consists of the files associated with a particular piece of software, the directory structure in which the files are stored, and attributes such as the name of the software object, the path to its toplevel directory, the author's name, and so on.

Rather than dictating the structure of the various types of software objects as was done in earlier versions of VisiQuest, the flexible code generation system of VisiQuest allows software object class types to be defined by the developer of the code generator. Thus, the number of different types of software objects that can be sup-

ported is unlimited. <sup>1</sup> The VisiQuest 2001 system comes with several standard code generators, which support the following software object class types: *kroutine objects*, *xvroutine objects*, *library objects*, *pane objects*, *workspace objects*, and *script objects*.

The different class types for software objects reflect their purpose, the types of files associated with them, and the types of operations that can be performed on them. For example, a *kroutine object* is a program that the user will execute to perform a task; it has a user interface, source code, and documentation; operations that can be done on the program object include code generation, user interface design, source code modification, and so on. A *library object*, on the other hand, is simply a collection of functions which are used by other programs; it has source code and documentation but there is no user interface involved. The details of the various types of software objects will be explained later in this chapter.



**Figure 3:** The VisiQuest software system is made up of toolboxes. When using the VisiQuest software development environment, you create toolboxes of your own. In a toolbox, there may be a variety of software objects, including *kroutine objects*, *xvroutine objects*, *library objects*, *script objects*, *pane objects*, and *workspace objects*. All software objects have database files (not depicted) that store information about the software object's attributes (name, author, paths to files, etc).

The VisiQuest software development environment supports the organization depicted above in a way that is designed to reduce the detail and complexity inherent in a large-scale software system. This environment is

<sup>1</sup> The flexible code generation system of VisiQuest makes it possible to seamlessly create and integrate new software object class types as desired. No documentation on this procedure is available, although AccuSoft Corporation does offer services to deliver support for new software object class types.

comprised primarily of two high-level tools, craftsman and composer, for managing toolbox and software objects respectively. Craftsman is used to create, delete, and copy toolbox objects and software objects; Composer provides the toolbox programmer with convenient access to all of the software object components and can invoke all of the operations needed to edit and manage existing software objects. These two tools work together to provide a high level, visual environment in which toolboxes can be created and software can be written, documented, and installed. This manual is targeted at the toolbox programmer and details how to develop and integrate new programs and applications within VisiQuest 2001. This particular chapter serves to introduce terminology used throughout the manual as well as to provide an example that shows how a toolbox programmer uses the software development tools to develop a simple program. For specific details concerning toolbox creation/management, user interface development, program development, writing documentation etc., the individual chapters in this manual should be consulted.

## **B. Toolbox Objects**

A toolbox object provides a convenient way of presenting VisiQuest developers with an encapsulated collection of information processing programs, interactive applications, and/or libraries designed for a specific application area. The toolbox object enforces a pre-defined directory structure in which its software objects are located; it manages both itself and its software objects via an object-based interface to a software database. It should be noted that some of the directories below will not be created until they are needed.

A toolbox contains the following directories:

**bin** This directory is where all the executable programs from your toolbox are located.



**config**

This directory contains directory configuration files (Dir.\*) and machine configuration files (Site.\*). 1item "data" Data files that can be used with the programs in the toolbox are stored in this directory. Often, this directory may contain subdirectories indicating general categories of data, such as "images," "sequences," "signals," and so on. See the `sampledata` toolbox for an example of data directory layout.

**examples**

This directory contains unsupported example programs that can be distributed. Example programs are generally used to demonstrate the proper use of public library calls for a library contained in the toolbox. Note that these programs are *not* formalized software objects created with `composer`, but simply user-created directories containing a main program (no code generation involved). They generally have no documentation other than comments inside the code. For examples of the layout and implementation of example programs, see the `envision` toolbox.

**include**

This directory contains the public include files associated with libraries in the toolbox. The include directory contains one include file named `{toolbox}.h` for the entire toolbox. It will also have one subdirectory for each library object in the toolbox, where each subdirectory is named after its library and contains the *installed* versions of the include files for the library. In other words, the include files in these directories are never modified by hand; they are simply copies of the original include files which are located in each library and maintained by the library developer. They are copied here from the originals when the library developer executes "kmake install". This set up allows the original include files to be easily packaged with the library object, but provides copies of include files at the toplevel toolbox location where they are needed in order to simplify compile rules.

**lib** This directory contains the compiled archives for each library object that exists in the toolbox.

**mach**

This directory is only used when the toolbox is supporting compiles on multiple architectures. In this case, it will contain one subdirectory for each supported architecture (solaris 2.6, irix 6.5, etc) in which symbolic links to the toolbox source code are used to build a separate compile for each architecture. See Section ? for more details.

**manual**

This directory contains the manual for the toolbox. It will contain one subdirectory for each chapter in the manual, plus a README, and directories for the glossary, index, and hardcopy.

**objects**

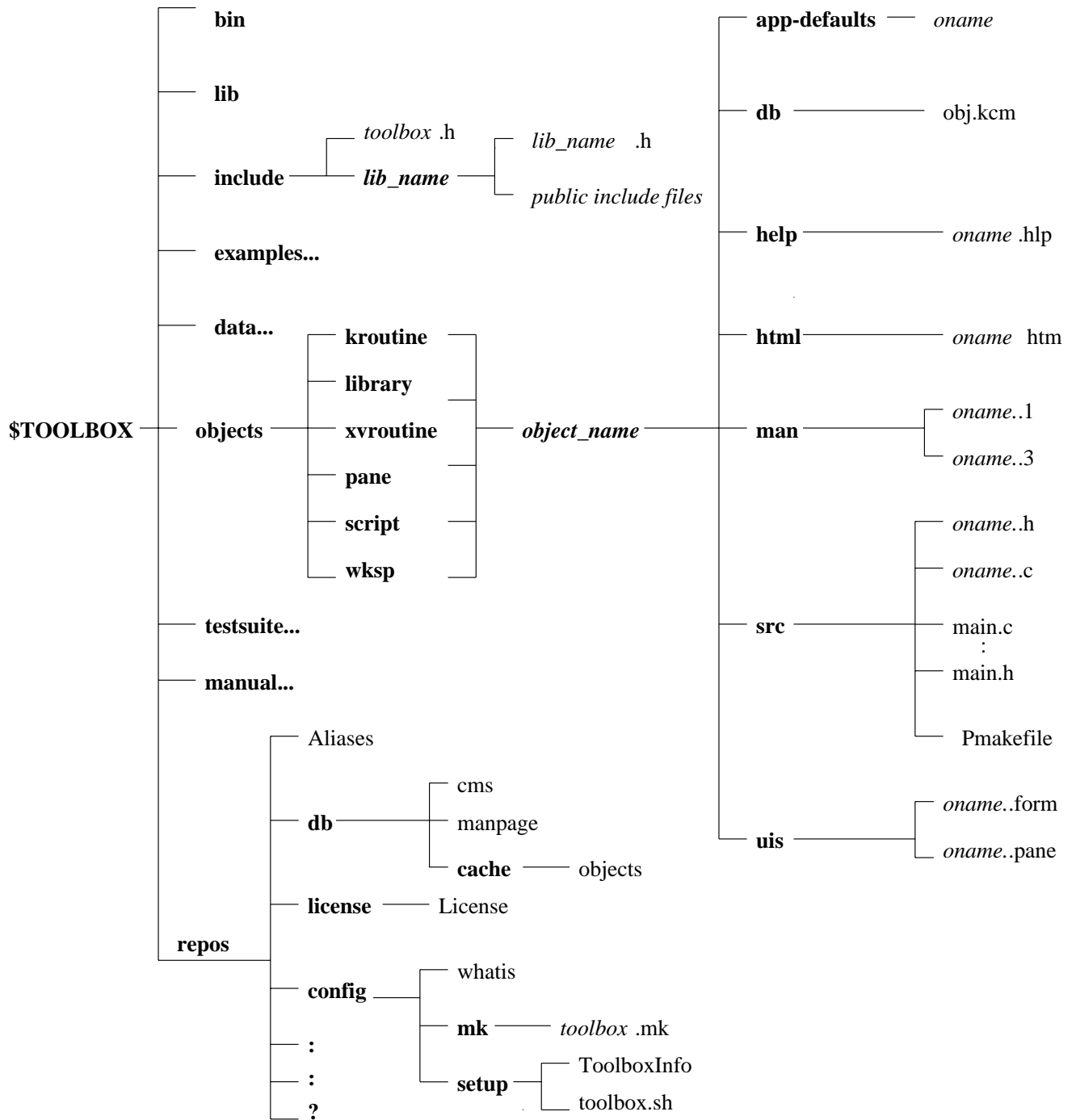
Software objects are located in this directory. There will be one directory for each class of software object that exists in the toolbox, named after the type; thus, there may be one or more of the standard VisiQuest *kroutine*, *xvroutine*, *script*, *library* and *pane* directories, as well as directories for user-defined software object types (if applicable). Under the classtype directories will be one subdirectory for each software object of that type, named after the software object itself.

**repos**

This directory is a repository for toolbox configuration files and the toolbox object database file.

**testsuite**

This directory is the location for any testsuites that are created in order to test the correctness of programs or libraries in the toolbox.



**Figure 4:** The directory structure associated with a toolbox, including the subdirectory structure that is associated with any software objects that may "live" in the toolbox.

## C. Software Object

A *software object* is contained within a *toolbox* and is composed of source code, documentation, and user interface. Every software object has configuration information files stored in a database that saves various information about the software object such as name, location, and so on. The different types of software objects include:

- *kroutine program objects*
- *xvroutine program objects*
- *library objects*
- *pane objects*
- *script objects*
- *workspace objects*

The next sections in this chapter cover each kind of software object listed above.

### C.1. Kroutine and Xvroutine Software Objects

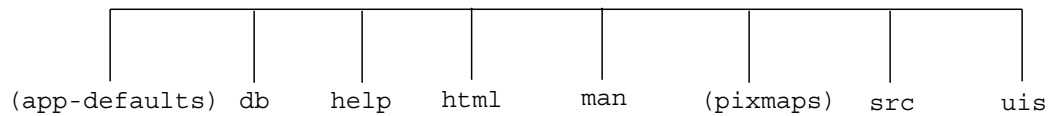
A VisiQuest *kroutine* is a data processing program made up of source code, a UIS file, documentation, and configuration information. It will be located in the *objects/kroutine* directory of a toolbox. The vast majority of programs in VisiQuest 2001 are *kroutines*; they include all data manipulation, image processing, and signal processing routines. A *kroutine* does not display any graphics or images; data is entered, processed, and written out. A *kroutine* does not have a GUI, but the GUI is not necessary to execute the program unless the program is accessed from within VisiQuest. A *kroutine* always has a Command Line User Interface (CLUI) with a Graphical User Interface (GUI) counterpart. An example of a *kroutine* is *karith2*, which is one of the many data processing routines available within VisiQuest 2001.

An *xvroutine* is a VisiQuest Toolbar that requires the user to provide input via a graphical user interface. In addition, it usually also displays graphics, images, or both. *Xvroutines* are located in the *objects/xkroutine* directory of a toolbox. There are two types of *xvroutines*: "interactive" and "non-interactive," although these terms are used loosely, as both types of *xvroutines* are interactive to some extent. Like *kroutines*, *xvroutines* always have a Command Line User Interface (CLUI) with a Graphical User Interface (GUI) counterpart.

An interactive or *classic* *xvroutine* has a "full" Graphical User Interface (GUI) in addition to (and generally a great deal more extensive than) its standard CLUI/GUI counterparts. The interactive GUI for the *xvroutine* may include multiple panes, multiple subforms, and so on. The number of interactive *xvroutines* in VisiQuest 2001 are relatively few, but they are the most visible VisiQuest 2001 programs; *editimage*, *VisiQuest*, *guise*, *craftsman*, and *spectrum* are some examples.

A non-interactive, or *hybrid* xvroutine is an application that displays graphics or images, but does not need the formality of the interactive GUI used by an "interactive" xvroutine; the standard CLUI/GUI counterparts are sufficient for these programs. There are a number of these "non-interactive" or "hybrid" xvroutines in VisiQuest 2001; a good example of this type of program is the `putimage` program in the `envision` toolbox.

## Kroutine and xvroutine objects



---

**Figure 5:** The top level directory structure of kroutines and xvroutines. Only xvroutines can use the optional *app-defaults* and *pixmap*s directories.

---

### **app-defaults**

This directory, used by xvroutines, may contain an application defaults file named after the program. Identical in syntax to a `.Xdefaults` file, this file can be used to set colors, fonts, and other related attributes in applications.

**db** This directory contains the database file, *obj.kcm*. The *obj.kcm* file maintains information about the software object and its attributes. It is used by the software development system and should not be manually edited.

### **help**

If the program is to be installed in VisiQuest, the code generator will generate a *\*.hlp* file in this directory. This help file is a re-formatted version of the documentation written in the *html* file that will provide the online help for the program when it is accessed via VisiQuest. By convention, xvroutines have a help button on each pane and subform of their GUIs; therefore, with xvroutines, a *\*.doc* file with the source of the online help for each help button is also located in this directory.

### **html**

This directory contains the html formatted source for the program documentation in a *\*.htm* file. Documentation is written in the *\*.htm* file, and then propagated to the help and man pages using the code generators.

### **man**

This directory contains the man page for the program. An nroff-formatted version of the *\*.html* file, the man file is used on UNIX systems when the user calls up the man page for the program using the "kman" command.

### **pixmap**s

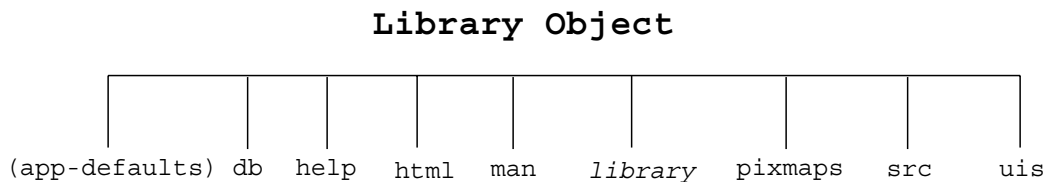
This optional directory, used by xvroutines, is the location for any pixmaps that may be used in the application.

**src** This directory contains the source code for the program as well as the Pmakefile.

- uis** This directory contains one or more User Interface Specification (UIS) files that define the user interface for the kroutine or xvroutine. Every VisiQuest kroutine and xvroutine must have a *\*.pane* file. Xvroutines also have a *\*.form* file. Xvroutines with complex interfaces may also have optional *\*.subform* files, in which the specification for each subform is isolated.

## C.2. Library Object

A *library object* represents a collection of functions which facilitate the re-use of code. It is a composition of source code and documentation organized in the *objects/library* directory of the toolbox.



---

**Figure 6:** The top level directory structure of a library object.

---

### **app-defaults**

This optional directory, used by only those libraries that are X11-based, may contain an application defaults file named after the library. Identical in syntax to a *.Xdefaults* file, this file can be used to set colors, fonts, and other related attributes on visual and GUI objects created by the library.

- db** This directory contains the database files "obj.kcm" and "function.db". The "obj.kcm" database file maintains information about the library object and its attributes. The "function.db" file stores a list of functions in the library and their short descriptions. These files are used by the VisiQuest software development tools for object management and tool synchronization; they should never be manually modified.

### **html**

The man page in html format, *\*.htm*.

### **man**

The man page for the library as a whole, as well as man pages for each public routine in the library, will be located in this directory. All man pages for the library use the ".3" postfix.

### *library*

The name of this include directory is the same as that of the library object. It contains all the public include files for the library. The versions of these public include files that are actually used by other software objects are copied to *\$TOOLBOX/include/{library}/* directory when a "kmake install" is done; thus, it is necessary to do a "kmake install" whenever a public library file in this

directory is modified.

**pixmaps**

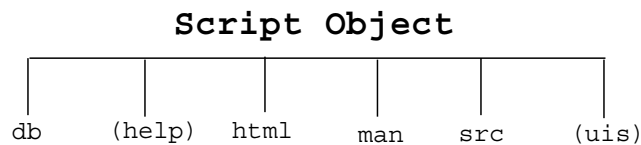
This optional directory is the correct location for any pixmaps which are used by functions in the library.

**src** This directory contains the source code for the library and the the Pmakefile.

**uis** This optional directory is used by only those libraries that are X11-based and implement visual or GUI objects having associated *menuforms*. With such libraries, the *uis* directory is used to store the UIS files defining the menuforms for the visual or GUI objects in the library.

### C.3. Script Objects

A *script object* is a utility program that is written in any scripting language. VisiQuest 2001 supports the creation and maintenance of *sh* scripts, *csh* scripts, and *perl* scripts. The CLUI must be written and maintained by the shell script programmer. If the script is to be installed in VisiQuest, it is the developer's responsibility to ensure that the CLUI implemented in the shell script matches the parameters specified in the \*.pane file.



---

**Figure 7:** The top level directory structure of a script object.

---

**db** This directory contains the database file, *obj.kcm*. The *obj.kcm* file maintains information about the software object and its attributes. It is used by the software development system and should not be manually edited.

**help**

If the script is to be installed in VisiQuest, the code generator will generate a *\*.hlp* file in this directory. This help file is a re-formatted version of the documentation written in the *html* file that will provide the online help for the program when it is accessed via VisiQuest.

**html**

This directory contains the html formatted source for the program documentation in a *\*.htm* file. Documentation is written in the *\*.htm* file, and then propagated to the man page using the code generators.

**man**

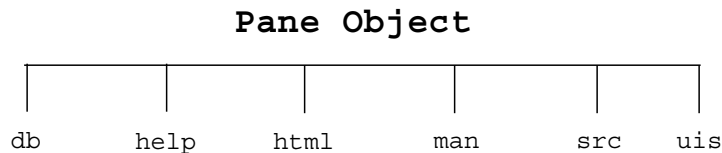
The man page for the script object. An nroff-formatted version of the \*.html file, the man file is used on UNIX systems when the user calls up the man page for the program using the "kman" command. The man page will use the ".1" postfix.

**src** This directory contains the script itself, as well as the Pmakefile.

**uis** This directory will contain the \*.pane file for the script if the developer wants the script to be available for use in VisiQuest. Accessibility via VisiQuest is the only reason for a script object to have a \*.pane file.

## C.4. Pane Objects

A *pane object* is a special type of program object which depends on another program (a kroutine or an xvroutine, referred to as the *base program* of the pane object) to provide its functionality. Pane objects allow developers to provide an alternate command line and a GUI to existing programs. A pane object's two key features are (a) a GUI to the program which provides the functionality, and (b) an automatically-generated shell script which simply executes the kroutine in question with the arguments that coincide with the desired GUI. The pane object was created (1) to conserve disk space and facilitate the re-use of code, and (2) as a mechanism that can be used to integrate non-VisiQuest programs into VisiQuest.




---

**Figure 8:** The top level directory structure of a pane object.

---

To conserve disk space and facilitate the re-use of code, consider algorithms which are very similar in structure but different in actual operation. For example, algorithms for unary pointwise operations on a data set (such as *square root*, *absolute value*, *offset*, etc.) always read in some data, process it using a single operator, and finally write out some data. It would be quite conceivable that different programs could be written to handle each case. However, a great deal of the code involved would be redundant, and the amount of disk space necessary to store the binaries would be multiplied with each additional unary operation that was implemented. Alternatively, a single data processing routine (kroutine) could be written to handle all the variations; this kroutine would accept a flag indicating the unary pointwise operation to be performed (sqrt, abs, etc.), and perform the specified operation accordingly. Thus, all unary pointwise operations can be combined into a single program.

From a GUI perspective, however, the critical issue is the unary operation to be performed. It would be nice to be able to access each pointwise operator individually from the command line as well as from the standard GUI and from within the visual language. The pane object allows us to have the best of both worlds. After the kroutine providing the multiple-operation functionality is implemented, a pane object representing each

capability of the kroutine is created, and the multiple-operational kroutine serves as a base program for a number of pane objects. Each pane object provides the user with a different "face" for the base program.

For example, let's continue to look at the case of unary pointwise operations. The `karith1` kroutine implements over 20 different single operator pointwise arithmetic operations (each of which has an associated pane object). It can be executed directly to do square root operations when the `[-sqrt]` flag is provided. However, a better GUI to the `karith1` functionality is provided with the `ksqrt` pane object, which uses `karith1` as a base program. `ksqrt` allows you to execute from the command line:

```
% ksqrt -i input_file -o output_file
```

Perhaps more important than the Command Line User Interface (CLUI) provided by `ksqrt` is the GUI it provides from within `VisiQuest`. With this alternative interface, it is easy for users to find what they are looking for: Arithmetic => Single Operand Arithmetic => Square Root. Thus, the desired CLUI and GUI are provided by the pane object, while the functionality for the operation is combined with many others in the `karith1` program object.

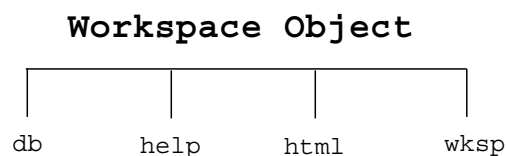
In addition to conserving disk space and encouraging re-use of code while presenting a simple GUI, a second use for pane objects is as support for the integration of non-`VisiQuest` programs into `VisiQuest`. The objective here is to get the programs into `VisiQuest` without having to rewrite them. The main hurdle is that every program in `VisiQuest` must have the standard `VisiQuest` GUI. The use of pane objects makes system integration an easy task. For each non-`VisiQuest` program, a pane object is created. The bulk of the effort in this procedure consists of creating a `VisiQuest` GUI/CLUI for each program that is to be integrated. With the standard `VisiQuest` GUI to the program, an automatically generated shell script is created which invokes the program as desired, and the integration task is accomplished.

## C.5. Workspace Objects

A *workspace object* is a software object that encapsulates a `VisiQuest` workspace file with its documentation. Workspace objects provide a nice way to "package" visual programs for other users or demos.

Workspace objects are accessible from the `VisiQuest` Object menus by category, subcategory and name. There is also a special Workspace Object browser, which is displayed by selecting "Open Object" from the `VisiQuest` "File" menu.

When a workspace object is restored in `VisiQuest`, a special "help" icon will appear on the `VisiQuest` command bar. The user may click on this icon to access online help for the workspace.



---

**Figure 9:** The top level directory structure of a workspace object.

---



**db** This directory contains the database file, *obj.kcm*. The *obj.kcm* file maintains information about the software object and its attributes. It is used by the software development system and should not be manually edited.

**help**

This help file is a re-formatted version of the documentation written in the html file that will provide the online help for the program when it is displayed in VisiQuest.

**html**

This directory contains the html formatted source for the workspace documentation in a *\*.htm* file. Documentation is written in the *\*.htm* file, and then propagated to the man page using the code generators.

**wksp**

This directory contains the saved VisiQuest workspace file that will be displayed when the workspace object is opened in VisiQuest.

## C.6. Software Object User Interfaces

Each program object and pane object has a standardized Command Line User Interface (CLUI) with a GUI counterpart. For example, a data processing program will have a CLUI which specifies the data to be read in, the resulting output data, and any other parameters needed to specify how the data is processed; when using the CLUI, the user will enter these values from the command line at run time. The GUI associated with the data processing program is simply a graphical representation of the CLUI. Instead of entering the values from the command line, the user fills in the desired values on the GUI before executing the program. Both the GUI and the CLUI of the data processing program are defined by the same User Interface Specification (UIS) file. As an example, consider the program *krm*, which is used to remove a toolbox, software object, or file. The CLUI usage statement for *krm* is shown below:

```
Usage for krm:  Removes a software object or file from a toolbox

% krm
[-tb]          (string)  toolbox to be removed (or toolbox from which to remove object
                (default = null)
[-oname]       (string)  software object to be removed
                (default = null)
[-i]           (infile)  file to be removed
                (default = null)
[-force]       (flag)    force removal?
[-V]           (flag)    gives the version number of the program
[-U]           (flag)    gives the usage of the program
[-P]           (flag)    interactive prompting for arguments
[-gui]         (flag)    run from GUI as defined in *.pane file
[-A]           (outfile) Creates an answer file
                (default = null)
[-a]           (infile)  Uses an answer file
                (default = null)
[-ap]         (infile)  prints answer file values
                (default = null)
```

---

**Figure 10:** The usage statement indicating the CLUI of *krm* as dictated by its UIS file. Note that the first

four CLUI arguments are specified explicitly in the krm UIS file; the latter six command line arguments, [-V], [-U], [-P], [-gui], [-A], [-a] and [-ap] are standard arguments for every VisiQuest 2001 program and are included automatically.

---

As indicated by the usage statement, to use krm to remove the program object `old_program` from the toolbox `my_toolbox`, the following command would be executed from the command line:

```
% krm -tb my_toolbox -oname old_program
```

The Graphical User Interface (GUI) of the krm program has the same arguments as the CLUI: a toolbox name, an object name, a filename, and a force option. The six standard command line arguments ([-V], [-U], etc) do not appear on the GUI.

As mentioned earlier, both the GUI and the CLUI of a VisiQuest 2001 program are defined with a User Interface Specification (UIS) file. The UIS file is an ASCII file with a strictly structured syntax. Some lines in the UIS file are present in every UIS file; other lines specify a particular argument in the CLUI and the corresponding selection on the GUI. The UIS file that defines the GUI and CLUI of krm is shown below.

```
-F 4.3 1 0 52x1+0+0 +0+0 'VisiQuest' VisiQuest
-M 1 1 52x1+0+0 +1+0 'Removes a software object or file from a toolbox' subform
-P 1 0 52x1+0+1 +0+0 ' ' krm
-D 1 0 9x1+0+0 'Options' _gui_options
-H 1 6x1+0+0 'License' 'license' $BOOTSTRAP/repos/license/License license
-E
-R 1 0 1 5x1+35+0 'Run' 'execute krm' $DEVELBIN/krm
-H 1 5x1+41+0 'Help' 'help page for krm' $DEVEL/objects/kroutine/krm/help/krm.hlp help
-Q 1 0 5x1+47+0 'Close'
-s 1 0 1 0 0 'String' 52x1+0+2 ' ' 'Toolbox Name' 'toolbox to be removed (or toolbox fro
-s 1 0 1 0 0 'String' 52x1+0+3.5 ' ' 'Object Name' 'software object to be removed' onan
-I 1 0 1 0 0 1 52x1+0+5 ' ' 'Filename' 'file to be removed' i
-t 1 0 1 0 0 17.25x1+0+6.5 'Force removal?' 'force removal?' force
-E
-E
-E
```

---

**Figure 11:** The krm.pane UIS file defines the GUI and CLUI of the krm program. The [-F], [-M], and [-P] lines are required for every UIS file; the [-D] to [-E] submenu definition is included for use when the GUI for the program is used within VisiQuest, and is standard for \*.pane files. The [-R] line specifies the *Run* button, the [-H] line specifies the *Help* button, and the [-Q] line specifies the *Quit* button, all three of which appear at the top of the GUI (these lines are ignored by the CLUI). The following four UIS lines specify the CLUI arguments of krm; these are the two [-s] lines that specify the "-tb" and "-oname" arguments, the [-I] line that specifies the "-i" input file argument, and the [-t] line that specifies the "-force" flag. On the GUI, these last four UIS lines are interpreted as two string selections, an input file selection, and a flag selection which the user can set as desired before executing krm by clicking on the "Run" button.

---

UIS files are created with the `guise` GUI design tool (see Chapter 4 of this manual). If necessary, they may also be edited manually with a visual editor. UIS files have three purposes:

1. To define the CLUI of VisiQuest programs.
2. To define the GUI of VisiQuest programs.
3. To define the interactive GUI for interactive X-Windows based applications (xvroutines).

There are two different types of UIS files, each with a distinct purpose. Depending on the type of application, a given program may have one or both types of UIS files. The two types of UIS files include:

```
*.pane files  
*.form files
```

where "\*" represents the unique name of the program in question.

### C.6.1. The \*.pane UIS File

All program objects and pane objects have a \*.pane file. This file specifies the CLUI of the program, and its GUI counterpart, as in the previous example with krm.

### C.6.2. The \*.form UIS File

The \*.form file specifies the interactive GUI for an interactive X-based application (*xvroutine*). Only *xvroutines* use a \*.form file; the \*.form file only specifies the interactive GUI for the application, and not the CLUI or the standard GUI that appears when the xvroutine is executed from VisiQuest or when using its [-gui] option from the command line.

For a good example, run the editimage image display and manipulation application (% editimage). The large, interactive GUI that pops up when you run editimage is defined by its \*.form file. This interface, in contrast with the simple editimage pane that appears when you execute editimage using (% editimage -gui), or when you open an "editimage" glyph in VisiQuest. *This latter GUI is the counterpart of the CLUI for editimage, and is defined by its \*.pane file.*

## D. Simple Programming Example

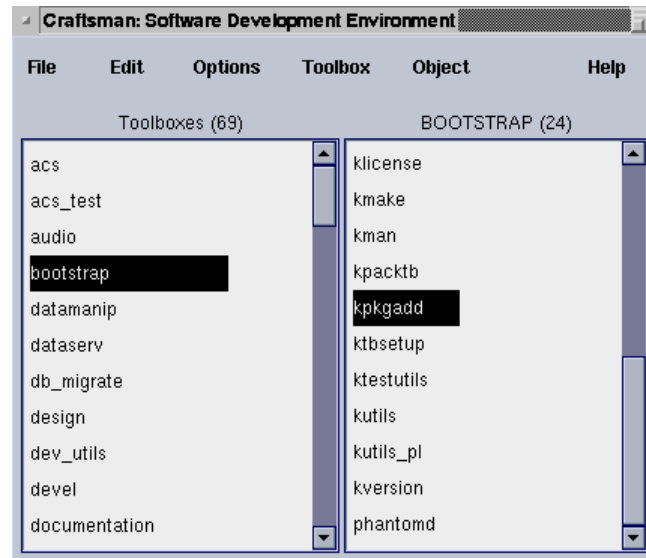
Now that you are familiar with the terminology, a simple programming example follows that will illustrate how the software development environment can be used to create a new program. The example is a simple kroutine called "mytest1" that takes an input file, opens it as a VisiQuest data object, and copies it to an output file. It will take an optional string argument on the command line and print it. If a string is not provided on the command line, it will print out "hello world."

### D.1. Creating a Toolbox

The first step in writing a new program is to use an existing toolbox in which to create the program object, or to create a new toolbox which will contain the new program. This is done using *craftsman*. For this example, we will create a new toolbox called *test\_toolbox*. The first step is to start up *craftsman* from the command line as follows:

```
% craftsman
```

This will bring up the craftsman toolbox and software object management tool. When craftsman is displayed, you will notice two lists, the left one corresponding to existing toolbox objects, and the right one corresponding to software objects. The toolbox list contains all toolboxes that currently exist. The list of software objects will remain empty until you click on a specific toolbox, at which time it will be updated with the software objects contained in that toolbox.

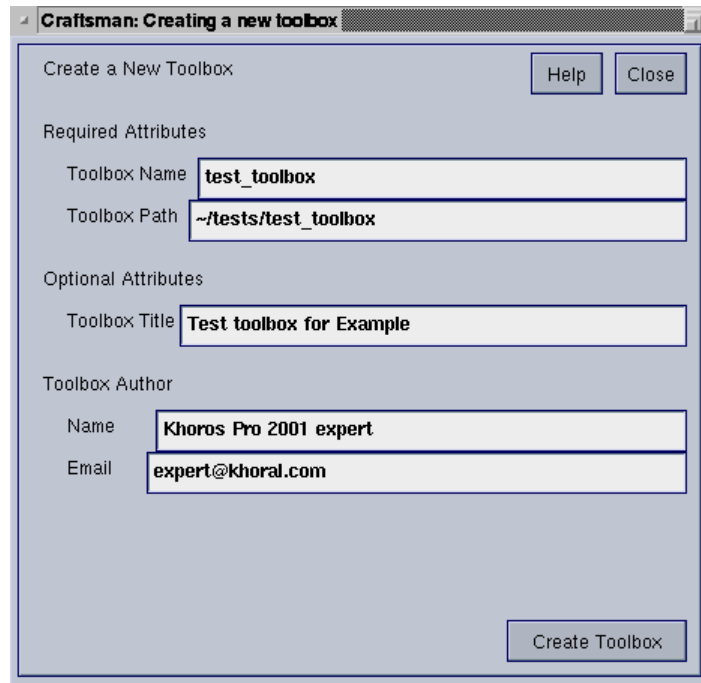


---

**Figure 12:** The craftsman software development environment allows you to create and manage both toolbox objects and software objects.

---

Above the list of toolboxes and the list of software objects are pulldown menus entitled *File*, *Edit*, *View*, *Toolbox*, and *Object*, respectively. Select *Create Toolbox* from the *Toolbox* pulldown menu. This action brings up a subform which allows you to enter toolbox information such as toolbox name, path, title, author, and so on. Fill out these fields, using "test\_toolbox" as the toolbox name. Then click on "Create Toolbox" to create the test toolbox.



---

**Figure 13:** Select "Create Toolbox" from the "Toolbox Operations" pulldown menu to display the subform which is used to specify and create a new toolbox.

---

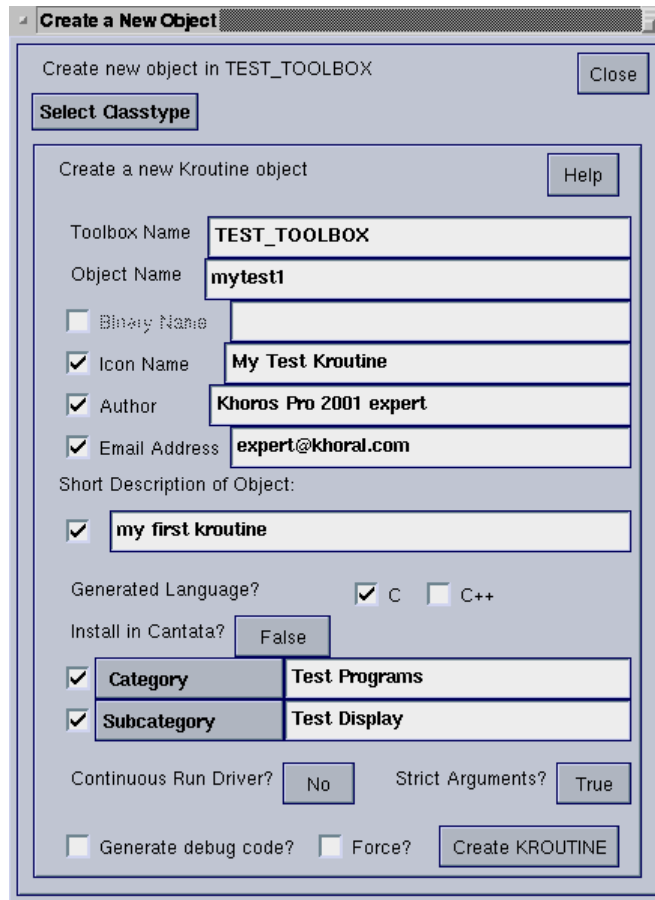
While craftsman is creating your new toolbox it will display a notify window, and the zen cursor will be displayed. When the new toolbox has been created, the notify window will disappear, the cursor will go back to normal, and the new toolbox will appear in the toolbox list.

## D.2. Creating a Software Object in the New Toolbox

Once your working toolbox has been created, you can create your new program object inside your new toolbox. To do this, select *Create Object* from the *Object* menu. This action brings up a subform which allows you to create new software objects.

First, you must select the desired software object class type from the walking menus that are available from the button at the top labeled *Select Classtype*. In fact, for this example, we need not change the class type since kroutine is the default. However, selecting *VisiQuest Routines->Xvroutine*, for example, changes the pane to have the values that must be specified for creating an xvroutine.

Now, enter the appropriate information for the class type, such as object name, binary name, icon name, category, subcategory, object type, and so on. Since complete documentation is given on craftsman in Chapter 2 of this manual, we will not repeat that information here. Set the attributes for the new program object similarly to the following:



---

**Figure 14:** Composer is used to create the new kroutine.

---

After entering the values for the attributes of your program object, click on the *Create Kroutine* button to create your new kroutine program object.

While craftsman is creating your new kroutine, it will display the notify window. When the new program object has been created, the notify window will disappear, and the new kroutine object will appear highlighted in the object list.

### D.3. Editing The Software Object

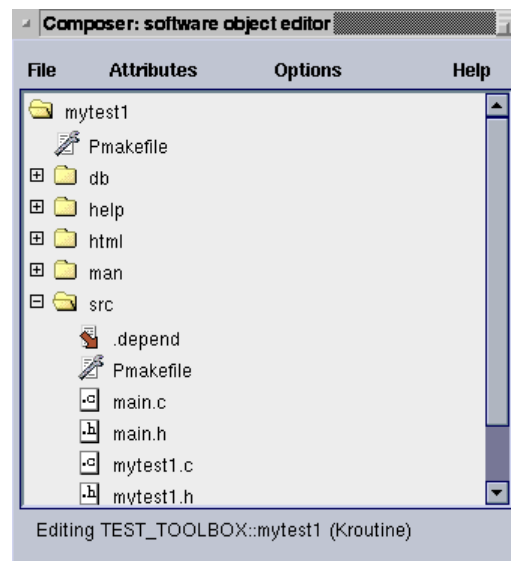
When craftsman creates a new software object, it copies templates and generates several files based on the software object class type. These files are discussed in detail in Chapter 5; for now, we will ignore the details and summarize the procedure as we go.

First, it is necessary to edit the GUI in the \*.pane file to specify the desired arguments to the new program. Next, we must edit the source code files to provide the functionality (which, as you recall, will take an input file and copy it to an output file, as well as printing a string if it was provided, or printing "hello world" if no string was provided). The documentation for the program must be written (at least in a limited fashion). Then, the program must be compiled and tested. Finally, we will want to see it integrated into VisiQuest, and verify

that it works within a visual program. Note that if this was a "real" program, we would iterate on the software development procedure for some time before the program was mature enough for general use.

The new kroutine is already selected from *craftsman's* object list; select *Edit Object (Composer)* from the *Object* menu. This action invokes the *composer* object editor on the new kroutine software object. *Composer* allows you to edit and manage existing software objects. It provides access to the *guise* GUI design tool, the *preview* GUI display tool, text editors, and formatted text display mechanisms. All files associated with the software object are accessible via the file list. In addition, *composer* software object editor allows you to set software object attributes, generate code, and compile the program.

The main *composer* window contains a list of all the software object directories. When a directory is preceded by a "+" symbol, the directory is closed; files within it are not listed. Clicking twice on a closed directory will open it, and display the files in the directory, indented beneath the directory name. An open directory is preceded by a "-" symbol; to close an open directory, click twice on it.



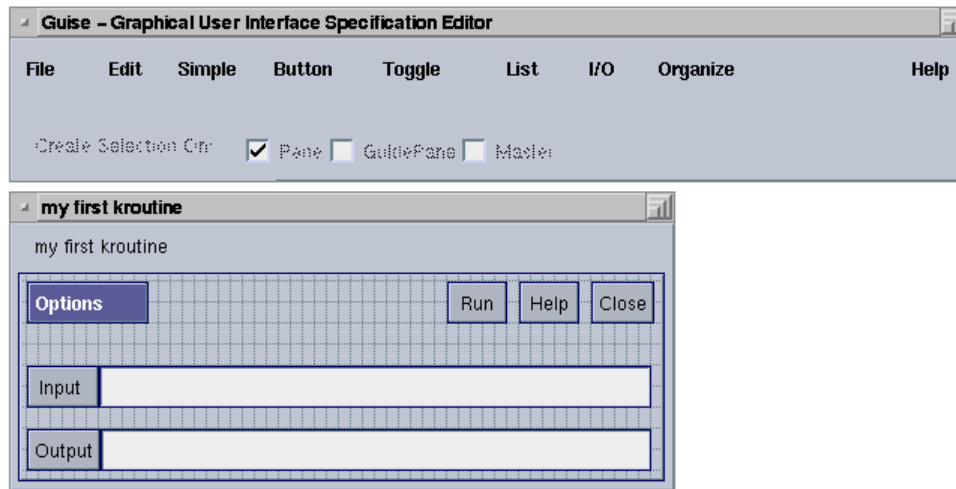
---

**Figure 15:** Here, *composer* is invoked on "mytest1". The directories (and Pmakefile) at the top level of the "mytest1" object are displayed. Clicking twice on the "uis" directory opens it, and displays the "mytest1.pane" UIS file.

---

### D.3.1. Editing the User Interface

The first step in writing the kroutine is to specify its user interface. *Guise* allows you to specify the GUI/CLUI of the new kroutine interactively. Open the "uis" directory by clicking on it twice, and then double-click on the "mytest1.pane" UIS file to bring up *guise*.




---

**Figure 16:** By default, the `guise` graphical user interface design tool will create a `*.pane` file defining a user interface with an input file and an output file, in addition to the standard "Options", "Run", "Help" and "Close" buttons.

---

`Guise`, the GUI design tool (UIS file editor), is used to create and modify user interfaces of software objects. The GUI defined by the template `*.pane` file has the standard "Options" pull-down menu (for use in `VisiQuest`), a "Run" button to execute the program, a "Help" button to access online help pages, and a "Close" button to close the GUI. Aside from these standardized GUI items, the template `*.pane` file defines an input file selection entitled, "Input", with the variable name "i", and an output file selection entitled, "Output" with the variable name "o". Thus, the CLUI defined by the template `*.pane` file would dictate that the program would be executed on the command line as:

```
% program -i {input file} -o {output file}
```

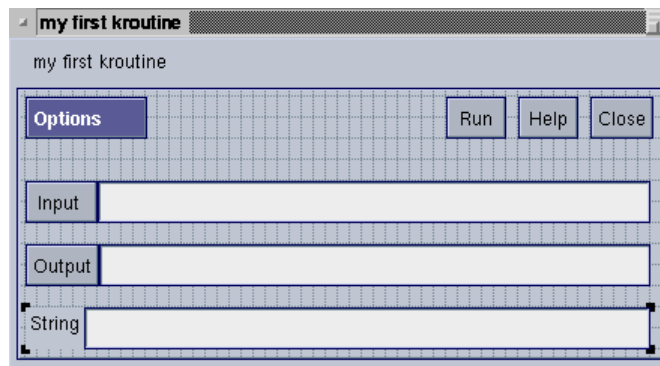
Recall that the `mytest1` program is supposed to print the text "hello\_world" or another string that the user might provide as well as taking an input file and producing an output file. So, in addition to the input file and output file selections provided by default, we want to add a string selection.

To edit the GUI of a software object, the GUI *must* be in *edit mode*. By default, `guise` brings up GUI's in edit mode as evidenced by the grid that appears on the backplane of the GUI. You put GUI's into and out of edit mode by *meta-clicking* on its backplane. "Meta-click" means that you hold the "Shift" key down while clicking the left mouse button on the backplane of the GUI. To familiarize yourself with this procedure, meta-click on the backplane of the test kroutine's GUI. It should go out of edit mode, indicated by the disappearance of the grid. Meta-click again on the GUI; it will go back into edit mode, and the grid will re-appear. GUI selections can *only* be edited interactively when the backplane is in edit mode.

To add a string selection, use the *Simple* pull-down menu to select the *String* selection. The string selection will appear on the GUI. You may move the string selection by "grabbing" it in the middle using the left mouse button and "dragging" it around on the user interface. "Grabbing" it near to the right side of the text box will cause the text box to resize horizontally. Make the string selection longer so that its right end aligns with the right side of the *Close* button. Move it down so it is not too close to the output file selection.



At this point, the GUI for your test program should look like this:



---

**Figure 17:** The GUI for "mytest1" after adding a string selection.

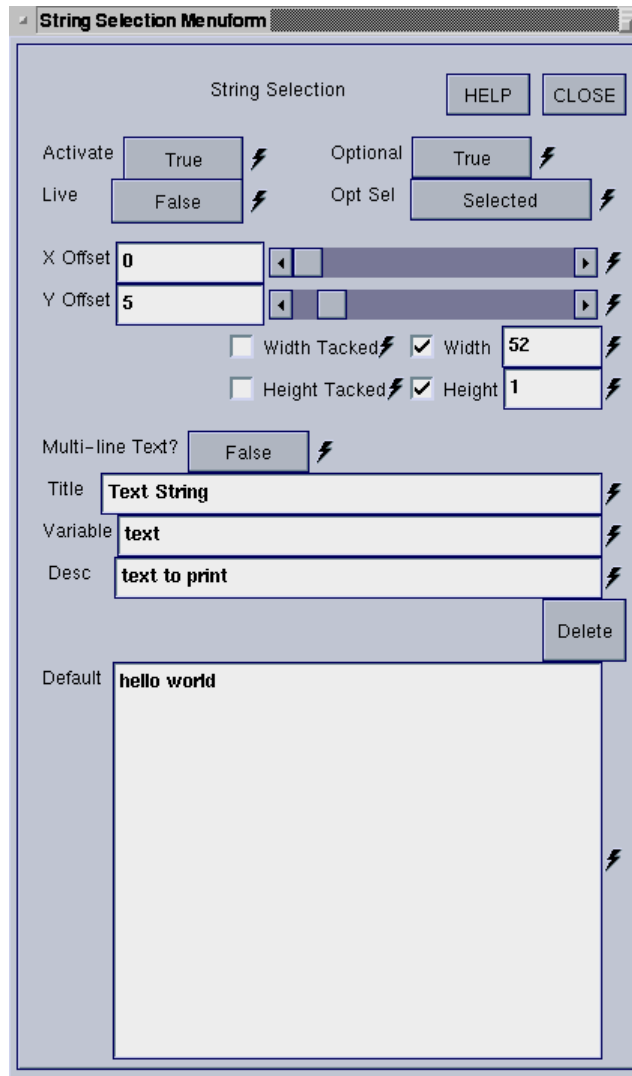
---

Next, you must set the attributes of the string selection. To do this, bring up the menuform for the string selection by selecting it with the middle mouse button.

Make changes to the attributes of the string selection as follows:

- 1) *Default:* hello world
- 2) *Title:* Text String
- 3) *Variable:* text
- 4) *Desc:* text to print.
- 5) Change the *Optional* logical selection (in the upper right hand corner) to True The menuform for the string selection is displayed below.

The menuform for the string selection as it appears after specifying these values is displayed below.



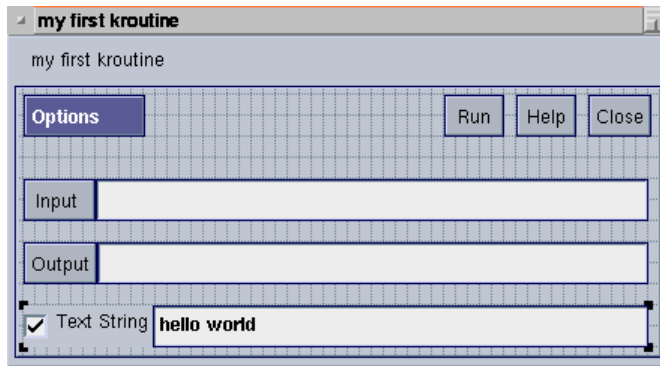

---

**Figure 18:** Each item on the GUI is a self-contained *user interface object*. Like toolbox objects and software objects, user interface objects have attributes that define their characteristics. When displayed in its GUI form, each user interface object has the capability to display its *internal menuform* with which the user may set its various attributes. The menuform associated with the string user interface item allows you to set all its attributes; some of these apply only to the GUI, other apply only to the CLUI, and still others apply to both. Some attributes of the string selection include geometry (GUI only), title (GUI only), variable (GUI and CLUI), default (GUI and CLUI), description (CLUI only), and so on.

---

When you are done, hit the *Close* button of the menuform to make it go away. You may also want to display and review the menuforms for the input file selection and the output file selection. For this example, we will leave them with default values.

Now, your GUI should look like the one below:



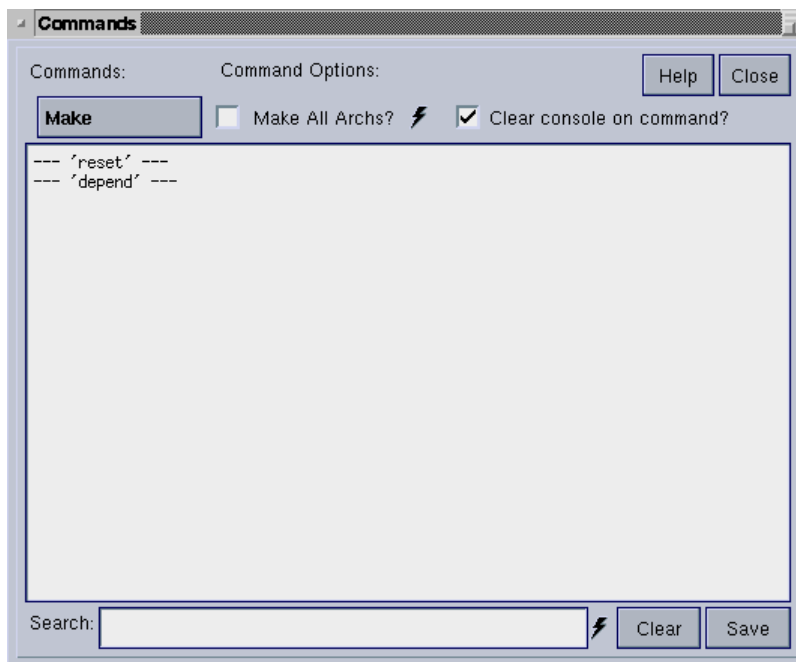
---

**Figure 19:** The GUI for mytest1 as specified by the mytest1.pane file after making changes to the string selection.

---

Now that all necessary changes to the GUI have been made, save them by selecting *SAVE (Needed)* from the *File* menu of *Guise*. After saving the changes, exit *guise* by selecting *Quit* from the *File* menu.

When *craftsman* initially created the kroutine object, it generated the CLUI code and documentation using the default "mytest1.pane" file that it copied from the template \*.pane file. Whenever changes to a UIS file are made, it is necessary to regenerate the source code and documentation, which will be outdated. To regenerate the source code and documentation, select *Commands->on Console*. This will display the Commands subform, shown below.



---

**Figure 20:** The Commands subform allows you to re-generate code for the CLUI, code for the GUI, and documentation. It also allows you to compile and install your software. Compile messages and warnings are printed in the large text window.

---

Select "kmake regen" from the *Make* menu to update all source code and documentation based on the new GUI as specified by the "mytest1.pane" file, which now has an input file, and output file, and an optional string selection. Composer will display the zen cursor while it is re-generating the code and the documentation for the program and display the generation status in the text window.

### D.3.2. Editing Source Files

Now that the GUI has been designed, you can edit the source code files to add the desired functionality. Note that we could have edited the source code first and the GUI second, but it's often easier to define the (initial) GUI first. In order to access all source code files in the software object, double-click on the "src" directory that appears in the main composer list. This displays all source code files associated with the kroutine. The "main.c" and "main.h" files contain only auto-generated code, and should not be edited, although you should view their contents. Code is added to the source code files which are named after the program, in this example, "mytest1.c" and "mytest1.h".

The automatically generated CLUI Information structure will hold the value of the string that was entered by the user. This CLUI Information structure is always called *clui\_info*, and is defined in the "main.h" file. The fields in the *clui\_info* structure are generated according to the CLUI arguments specified in the \*.pane file. They always consist of the variable name, followed by an underscore, followed by the data type. Thus, since the string argument that we defined in the "mytest1.pane" file had the variable name "text," we can deduce that the corresponding field in the CLUI Information structure will be *clui\_info->text\_string*. In addition, *clui\_info->text\_flag* will have a value of TRUE if the user in fact provided the [-text] argument on the command line. While editing the "mytest1.c" file, notice that the fields in the CLUI Information structure are always automatically generated in the header at the top of the main program for easy reference.

Edit "mytest1.c" by selecting it from the list and selecting *Edit->as Text*. This will cause an xterm to appear containing a session with your text editor. Add code to the "run\_mytest1()" function to open an input file as a VisiQuest data object, open an output VisiQuest data object, copy the input data object to the output data object, and print the value of the optional string argument. The Program Services 2 manual covers all functions to open, close, and manipulate VisiQuest data objects; these functions will not be covered here. The code for the "run\_mytest1()" function is as follows:

```
int run_mytest1(void)
{
    kobject data_object_in;
    kobject data_object_out;

    /*-- open input file as a VisiQuest data object --*/
    data_object_in = kpds_open_input_object(clui_info->i_file);

    /*-- open VisiQuest data object for output --*/
    data_object_out = kpds_open_output_object(clui_info->o_file);

    /*-- copy input data object to output data object --*/
    kpds_copy_object(data_object_in, data_object_out);

    /*-- close both objects (closing the output object writes it to disk) --*/
    kpds_close_object(data_object_in);
    kpds_close_object(data_object_out);

    /*-- print value of string argument --*/
    if (clui_info->text_flag)
        kfprintf(kstdout, "User specified string '%s'\n", clui_info->text_string);
}
```

```

        else kfprintf(kstdout, "Default string is '%s'\n", clui_info->text_string);
    return TRUE;
}

```

When you are done, save and quit the file.

## D.4. Editing Documentation Files

Documentation is always an important component of the software development process; you will want to fully document the "hello\_world" routine so that on-line documentation is available both from within VisiQuest and from the command line. When the program object was generated by *craftsman*, it generated three documentation files:

1. An HTML page ("mytest1.htm")
2. A VisiQuest online help page ("mytest1.hlp")
3. An on-line man page ("mytest1.1")

The "mytest1.htm" file contains HTML format specifications and provides a basic framework for you to add text to.

*Important Note:* The code generators take the information from the html page and propagate it to the man page and the help page. Thus, you should *only* edit the html page, and allow the information to be copied and reformatted for the other types of documentation. Never edit the help page or man pages directly, although you may view them.

Double click on the "mytest1.html" page or select *Edit->as Text*. *Composer* will initiate a session with your editor, containing the template "mytest1.htm" html page. This file should look like:

```

<HTML>
<HEAD>
    <TITLE> mytest1 Online Help Page </TITLE>
</HEAD>
<BODY>
<H2>TEST_TOOLBOX commands</H2><HR>
<H3>PROGRAM NAME </H3>
mytest1 - My example kroutine
<H3> DESCRIPTION </H3>
<!-- begin_arguments -->
<H3> REQUIRED ARGUMENTS </H3>
<DL>
<DT> <B>-i</B>
<DD>
type: infile
<BR>
desc: First Input data object
<BR>
<DT> <B>-o</B>
<DD>
type: outfile

```

```

<BR>
desc: Resulting output data object
<BR>
</DL>
<H3> OPTIONAL ARGUMENTS </H3>
<DL>
<DT> <B>-text</B>
<DD>
type: string
<BR>
desc: text to print
<BR>
default: hello world
<BR>
</DL>
<!-- end_arguments -->
<H3> EXAMPLES </H3>
<H3> SEE ALSO </H3>
<H3> RESTRICTIONS </H3>
<H3> REFERENCES </H3>
<!-- begin_short_copyright -->
<H3> COPYRIGHT </H3>
Copyright (C) 1993 - 1999, AccuSoft Corporation, ("AccuSoft Corporation").
All rights reserved. See $BOOTSTRAP/repos/license/License or run
klicense.
<!-- end_short_copyright -->
</BODY>
</HTML>

```

Documentation files have *tags* denoting text segments that are automatically generated. You may edit the html file as desired, as long as you do not change or delete these tags. You may not change of the text between these tags. The tags surround automatically generated documentation for arguments and copyright as follows:

```

<!-- begin_arguments -->
<!-- end_arguments -->
<!-- begin_short_copyright -->
<!-- end_short_copyright -->

```

First, add text after the *DESCRIPTION* header. Note that valid HTML formatting commands should be used when editing documentation files. You might wish to change the description to something similar to the following:

```

<H3> DESCRIPTION </H3>
This program copies the input file to the output file, and prints
the text specified. If no text is specified, "hello world" is printed.
<!-- begin_arguments -->

```

Also, you may wish to include some examples describing how to run "hello\_world." This can be done by adding examples between the *EXAMPLES* and *SEE ALSO* headers, as follows:

```

<H3> EXAMPLES </H3>
The following example displays the text "hello world."
<BR>
% hello_world -i infile.kdf -o outfile.kdf
<BR>
The following example displays the text "hi there beautiful world."
<BR>
% hello_world -i infile.kdf -o outfile.kdf -text "hi there beautiful world"

```

```
<BR>
<H3> EXAMPLES </H3>
```

In addition, you may add text to the SEE ALSO field, the RESTRICTIONS field, and the REFERENCES field. When you are done, save and quit the html page.

Note that if you wish to change the copyright notice, you do not edit the copyright in the documentation file. Instead, you must select *Toolbox->Toolbox Attributes* in *craftsman*, and use the Copyright subform there to change the long copyright (used in the source code) and the short copyright (used in the documentation). After changing the copyright, regenerate the code and the documentation with *composer*.

Once you have completed the man page, it is necessary to propagate these changes to the help and man pages. To propagate the changes, select the *kmake regen* from the *Make* menu of the Commands subform. This will regenerate the documentation.

## D.5. Compiling and Running The Program

Once the appropriate changes and additions have been made to the source code files, you are ready to compile the program. Compile by selecting *kmake* from the *Make* menu of the Commands subform. The result of the compile will appear in the Output Log window.

Alternatively, you may compile the program directly from the command line. Of course, you will first need to change to the source directory of the "mytest1" object. This will be in:

```
$TEST_TOOLBOX/objects/kroutine/mytest1/src
```

where \$TEST\_TOOLBOX represents the path to the "test\_toolbox" toolbox that you recently created. In this directory, type:

```
% kmake
```

Once the program is compiled, run the program from the command line. Try two tests:

```
% hello_world -i image:ball -o out.kdf
% hello_world -i image:ball -o out.kdf -text "The sky is blue today"
```

## D.6. Testing the Kroutine From Within VisiQuest

Assuming that the tests succeed, install the program with *kmake install* (available from the *Make* menu). Now that you have successfully created, compiled and tested your "mytest1" program, you can also include and execute it within *VisiQuest*. Recall that when first creating this program object using *craftsman*, you set *Install in VisiQuest?* to "True." This caused it to be included in *VisiQuest* as soon as it was available (even if it was not debugged yet). Also recall providing values for *Icon Name*, *Category*, and *Subcategory*; these attributes provide labels that are used when the program is automatically integrated into the *VisiQuest* GUI.

When creating new programs, it is frequently more convenient to set *Install in VisiQuest?* to NO, and then re-set the program attributes relating to *VisiQuest* when the program is ready for public consumption. These attributes can be easily changed by using *composer*'s *Attributes* subform to fill in a category, subcategory, and icon name, setting the *Install in VisiQuest* flag to "Yes," and hitting the *Apply Changes* button.

To verify that the "mytest1" program is now available in the visual language, execute

from the command line and wait for it to initialize. Note that one item in the Glyphs menu is now labeled with the *Category* string you specified for the program object: "Test Programs." Use the left mouse button to click on *Test Programs* to produce the walking menu of subcategories under *Test Programs*. Note that there is only one item in the menu, labeled with the *Subcategory* string you specified for the program object: "Text Display." If you were actually developing a complete toolbox, you might have several subcategories appear in the pull-down list under the *Test Programs* category.

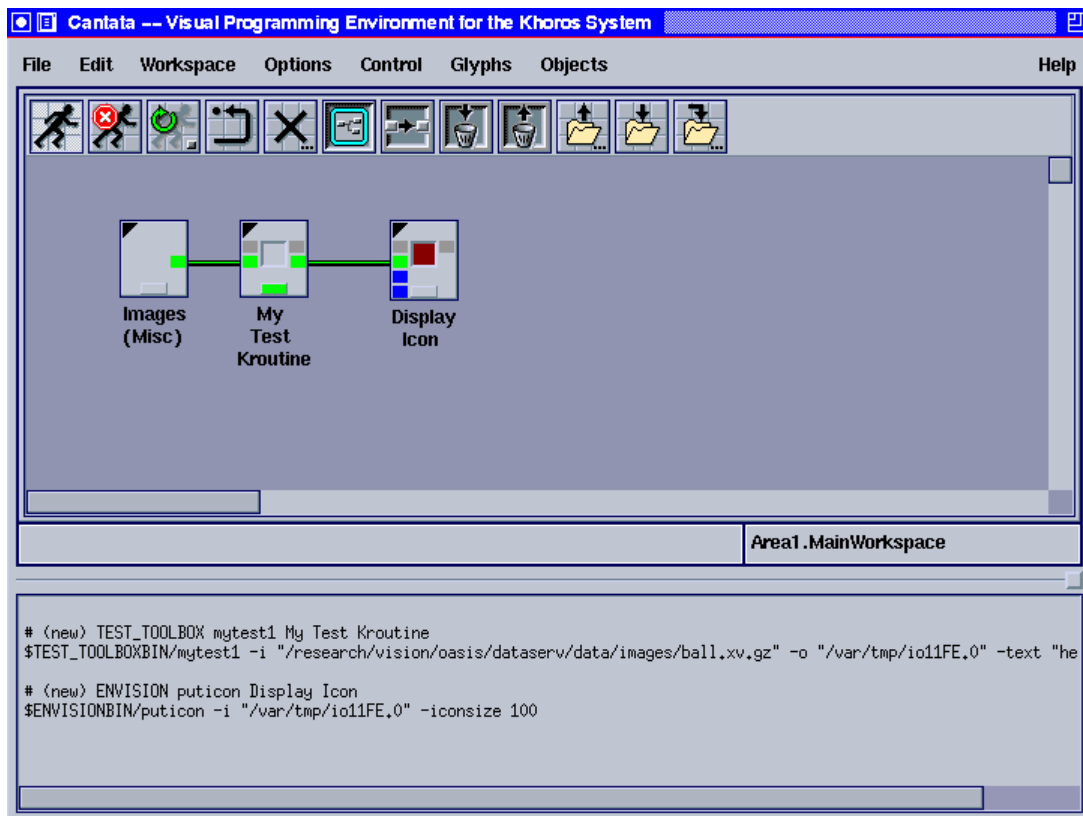
Select *Text Display* from the pulldown menu; this will display another walking menu containing the *Icon Name* string you gave for the program object: "My Test Kroutine." Selecting the "My Test Kroutine" entry from the walking menu causes the "mytest1" glyph to be created in VisiQuest's workspace. You can reposition the glyph by selecting it with the left mouse button and dragging it to the desired position.

Display the pane (GUI) for "mytest1" that you created earlier by clicking with the left mouse button on the *form* icon on the glyph (the black triangle in the upper-left corner). Using the GUI, specify the optional text string. Close the pane when you are done.

Now, the "mytest1" glyph requires an input and an output. Use the Glyph menus to create a glyph to provide an input file by selecting *Input/Output->Data Files->Images (Misc)*. If desired, you may open the pane for this glyph and select another input file besides the default ball image. Next, use the Glyph menus to create a glyph to display the image by selecting *Visualization->Non-interactive Image Display->Display Icon*. Hook the three glyphs together so that the "Images (Misc)" glyph provides the input for your "My Test Kroutine" glyph, and the output of your kroutine glyph provides the first input for the "Display Icon" glyph.

Finally, click on the VisiQuest "Run" button. This will run the visual program with your test kroutine glyph. Your test kroutine will read the input image and copy it to its output, where the "Display Icon" glyph will accept it as input and display it as an icon. When your test program runs, you will see the console button at the bottom of the glyph turn green, indicating that the program wrote text to stdout. Clicking on the console button displays its output.





**Figure 21:** A functional visual program using our new kroutine glyph.

When you are finished experimenting with VisiQuest, select *Exit* from the *File* menu.

## E. Conclusion

By this point, you have fully developed and tested a simple program using the VisiQuest 2001 software development tools. You should now be somewhat familiar with the software development environment and how it can be used to create both toolboxes and software objects. This chapter served to illustrate how the software development tools can be used together to develop programs and applications. The rest of this volume contains extensive documentation for how to use each of the tools and provides much more detail and reference than was mentioned in this chapter. The remaining chapters serve to document programming conventions, creating toolboxes, creating software objects, creating user interfaces, and displaying Graphical User Interfaces (GUI's). Chapter 5 gives extended information on developing software using VisiQuest 2001, and is considered required reading before serious software development begins.

# Table of Contents

A. Introduction . . . . .	1-1
B. Toolbox Objects . . . . .	1-5
C. Software Object . . . . .	1-8
C.1. Kroutine and Xvroutine Software Objects . . . . .	1-8
C.2. Library Object . . . . .	1-10
C.3. Script Objects . . . . .	1-11
C.4. Pane Objects . . . . .	1-12
C.5. Workspace Objects . . . . .	1-13
C.6. Software Object User Interfaces . . . . .	1-14
C.6.1. The *.pane UIS File . . . . .	1-16
C.6.2. The *.form UIS File . . . . .	1-16
D. Simple Programming Example . . . . .	1-16
D.1. Creating a Toolbox . . . . .	1-16
D.2. Creating a Software Object in the New Toolbox . . . . .	1-18
D.3. Editing The Software Object . . . . .	1-19
D.3.1. Editing the User Interface . . . . .	1-20
D.3.2. Editing Source Files . . . . .	1-25
D.4. Editing Documentation Files . . . . .	1-26
D.5. Compiling and Running The Program . . . . .	1-28
D.6. Testing the Kroutine From Within VisiQuest . . . . .	1-28
E. Conclusion . . . . .	1-30

**This page left intentionally blank**

# *Toolbox Programming*

## **Chapter 2**

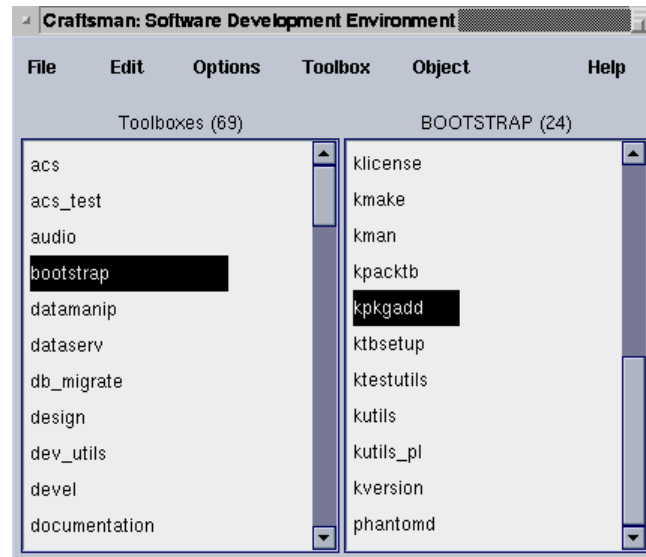
# **Craftsman**

## **Toolbox Object Management**

Copyright (c) AccuSoft Corporation, 2004. All rights reserved.



# Chapter 2 - Craftsman



**Figure 1:** Craftsman is the VisiQuest toolbox management tool. The left-hand list displays available toolboxes. Having selected a toolbox, the right-hand list displays objects within that toolbox.

## A. Introduction to Craftsman

The craftsman application is used to create and manage toolboxes and software objects. The operations supported are:

- *Toolbox Operations:* create a new toolbox; add a toolbox reference; change attributes of an existing toolbox; remove a toolbox reference; klint a toolbox; migrate a toolbox; delete an existing toolbox; create a package containing one or more toolboxes.
- *Software Object Operations:* create a new software object; spawn composer to edit a software object; change attributes; copy or move objects between toolboxes; rename an object; klint an object; delete an object; tearaway an object.

### A.1. Documentation Prerequisites

It is assumed that you have read the following documents and are familiar with the terminology introduced therein:

- Chapter 1, *Introduction*, in the *VisiQuest Overview*. This chapter provides a general overview of the levels of programming or interaction between the user and VisiQuest, the software development tools, and the programming services.
- The first chapter, *Introduction*, of the *Toolbox Programming* manual.

## A.2. Command-line Arguments

Craftsman is invoked using the following command:

```
% craftsman
```

Craftsman understands the standard VisiQuest command-line arguments, and additional `-tb` and `-oname` switches, used to specify an initial toolbox and object respectively. So, to start craftsman on the toolbox **test\_toolbox**, you would execute:

```
% craftsman -tb test_toolbox
```

Or, to start craftsman with the toolbox **datamanip** and the software object **kabsdiff** selected, you would run the following command:

```
% craftsman -tb datamanip -oname kabsdiff
```

## A.3. Craftsman's User Interface

When you start craftsman a single window will appear, which contains two scrolled lists. The left-hand list contains currently available toolboxes. The *Toolbox* menu button on the menubar provides access to the various toolbox operations. Clicking the left mouse button on a toolbox name highlights that toolbox, making it the currently selected toolbox.

The right-hand list displays all objects in the currently selected toolbox. Clicking the left mouse button on an object name selects that object. A text string above the object list gives the name of the currently selected toolbox, and the number of objects. The *Object* menu button on the menubar provides access to basic object operations: create, delete, edit, copy, move, rename, klint, and change attributes.

At any time, *craftsman* will only let you perform operations which are applicable. Inappropriate commands are ghosted out so that they cannot be selected. For example, if you haven't selected a toolbox, then the *Delete Toolbox* option on the *Toolbox Operations* menu will not be active.

Whenever you select a potentially destructive operation, you will be prompted to make sure you want to continue.

When running operations which are not instantaneous, or nearly so, *craftsman* will display the zen cursor to encourage you to wait patiently.

## A.4. Toolbox Operations

There are a number of toolbox operations available from the Toolbox menu. All toolbox operations except "Create Toolbox ...", "Add Toolbox Reference ...", and "Create Package ..." act on the currently selected toolbox. The user interface is enabled and disabled as toolboxes are selected or deselected to help prevent invoking inappropriate operations.

Toolbox Operations are as follows:

### **Create Toolbox ...**

This brings up a subform which allows you to create a new toolbox.

### **Add Toolbox Reference ...**

This brings up a subform that lets you add a toolbox reference which will make the software objects in that toolbox available to you.

### **Toolbox Attributes ...**

This brings up a subform that allows you to change the modifiable attributes of an existing toolbox.

### **Remove Toolbox Reference**

Removes the reference to the currently selected toolbox; software objects in the toolbox will no longer be available to you.

### **Klint Toolbox**

Runs the klint program on the currently selected toolbox. Klint checks for toolbox integrity; it reports discrepancies between existing files and database contents.

### **Migrate Toolbox**

Runs the kmigrate program on the currently selected toolbox. Kmigrate will update a toolbox from an earlier version to the current version of VisiQuest.

### **Delete Toolbox**

Prompts for confirmation before deleting the currently selected toolbox. Note that deleting a toolbox is irreversibly destructive; all directories, files and information related to the toolbox are deleted.

### **Create Package ...**

This option allows you to create a package containing one or more toolboxes. Toolboxes are packaged prior to distribution.

## A.5. Object Operations

Once a toolbox is selected from the Toolboxes list on the left, the software objects in that toolbox will appear in the Objects list at the right. All Object operations except "Create Object ..." act on the currently selected toolbox. The user interface is enabled and disabled as objects are selected or deselected to help prevent



invoking inappropriate operations.

Object Operations are as follows:

**Create Object ...**

This brings up a subform which allows you to create a new software object.

**Edit Object (Composer) ...**

This executes the Composer software object editor on the currently selected software object.

**Object Attributes ...**

This brings up a subform that allows you to change the modifiable attributes of an existing software object.

**Copy Object ...**

A subform is displayed which allows you to copy the currently selected software object.

**Move Object ...**

A subform is displayed which allows you to move the currently selected software object from one toolbox to another.

**Rename Object ...**

A subform is displayed which allows you to give the currently selected software object a new name.

**Klint Object**

Runs the klint program on the currently selected object. Klint checks for software object integrity; it reports discrepancies between existing files and database contents.

**Delete Object**

Prompts for confirmation before deleting the currently selected object. Note that deleting a software object is irreversibly destructive; all directories, files and information related to the object are deleted.

**Tearaway Object(s)**

Runs the xtearaway program to tear away software objects so that they may be used outside of the VisiQuest environment. It creates a temporary toolbox in the specified output directory, copies the object(s) to that toolbox, runs kpacktb to create a standalone package, and removes the temporary toolbox when done. The xtearaway program first creates a temporary configuration file for kpacktb to use and gives the user an edit session to inspect or modify this file if needed. In most cases the user will not need to change anything and will choose to use the default settings.

## B. Creating a new Toolbox

A toolbox provides the framework and context for software objects. At its lowest level a toolbox is a standardized directory structure with associated database and configuration files. The database provides information about the toolbox, and should only be accessed through the software development tools.

When you create a new toolbox, a standard directory structure is constructed, with its root directory being that which you gave as the Toolbox Path. The default structure, and the purpose of each directory, is described in Chapter 1 of the *Toolbox Programming Manual*.

### B.1. Toolbox Attributes

Every toolbox has a number of attributes associated with it, which provide information about the toolbox, and its author(s). All of the attributes can be specified when creating the toolbox; all except Toolbox Name and Toolbox Path may be changed later if necessary.

#### **Toolbox Name**

A toolbox's name is specified at creation time. The software development tools do not support renaming of toolboxes, so this attribute is created as read-only.

#### **Toolbox Path**

A toolbox's path is the path to the toplevel directory of the toolbox's directory structure. The path is specified when the toolbox is created, and cannot be changed thereafter.

#### **Title**

The *title attribute* for a toolbox is intended to be a short string, such as "*Geometry Visualization Toolbox*."

#### **Toolbox Author Name**

This is the toolbox author's name(s).

#### **Toolbox Author Email**

This is the email address of the point-of-contact for the toolbox.

### B.2. Creating the toolbox

The first item on the toolbox operations menu is *Create Toolbox*. Selecting this item will bring up a subform used to provide attributes of the new toolbox. You must provide the name and path for the new toolbox; the remaining attributes are optional, and can be set at a later date by selecting *Toolbox Attributes* from the same menu.

When specifying the path for the toolbox, you should include the name of the toolbox as the final directory in the path. For example, if user `fred` is creating a toolbox called `GIS`, he could specify the path in any of the following ways

```
/home/fred/gis  
~/gis  
./gis
```

When Craftsman starts up it checks to see whether the `VisiQuest_MAIL` and `VisiQuest_NAME` environment variables are set. If they are, then the *Author email* and *Author name* selections are set from their values.

## C. Adding Toolbox References

This subform allows you to add a reference to another toolbox to your VisiQuest environment. Note that this operation does *not* have anything to do with the currently selected toolbox; it is an independent operation. Adding a new toolbox reference will enable you to execute programs in that toolbox, modify its software objects using Craftsman and Composer, and access its operators in VisiQuest.

Specify the path of the toolbox to which you wish to add a reference. Click on the "Add Toolbox Reference" button to actually add the reference. This will cause your `$HOME/.kri/KP2001/Toolboxes` file to be updated with a line referencing the specified toolbox, and that toolbox will now be available in your VisiQuest environment.

## D. Changing Toolbox Attributes

The *Toolbox Attributes* subform is used to view and update the modifiable attributes associated with a toolbox.

This subform contains a menu entitled, "Select Attribute Type". From this menu, you may select four panes that are used to view and modify a toolbox's attributes:

### General Attributes

This pane contains general toolbox attributes that you are allowed to modify. You may change the toolbox title, toolbox author name, and toolbox author email address. After entering values as desired, click on "Apply Changes" to actually change the toolbox attribute values.

### Dependencies

A toolbox is dependent on another toolbox if it uses library functions from the other library or if it uses definitions made in the other toolbox's include files. All VisiQuest toolboxes are dependent on bootstrap; they may or may not be dependent on any other toolboxes. The toolboxes on which this toolbox is dependent will appear in the Toolbox Dependencies list on the left hand side of the subform. Toolbox dependencies are listed in a heirarchical manner.

For example, suppose the toolbox **test\_toolbox** is dependent on the **dataserv** toolbox because it contains kroutines which use polymorphic data services. The Toolbox Dependencies list will display the **dataserv** toolbox preceded by a "+" sign. The "+" sign indicates that the **dataserv** toolbox has toolbox dependencies of its own, which will be inherited by **test\_toolbox**. Clicking on the "+" sign "opens" the **dataserv** toolbox dependencies list, displaying the **bootstrap** toolbox.

To add a new dependency on another toolbox, select the desired toolbox from the right-hand list entitled "Add New Dependency" and click on "Add Selected Dependency". Functions and definitions in that toolbox will then be available for use in coding in your toolbox. To delete a toolbox dependency, select the toolbox from the "Toolbox Dependencies" list and select "Delete Selected Dependency". Chapter 5 of the Toolbox Programming Manual, "Writing Software With VisiQuest 2001", goes into greater detail on toolbox dependencies.

### Copyright

This pane is used to modify the copyright statements associated with a toolbox. The short copyright will appear at the bottom of documentation files. The long copyright will appear at the top of source code files.

### Files

The files pane gives access to various toolbox files:

1. The *pmake* configuration file for the toolbox
2. The toolbox include file
3. The ToolboxInfo file, which should contain a couple of paragraphs of ASCII text describing the toolbox
4. The toolbox Aliases file

If the file in question already exists, the button in front of it will read "Edit", and you can edit the file by clicking on the button. If the file in question does not yet exist, you can create one by clicking on "Create" and then edit the file by clicking on "Edit". Detail on the use and syntax of these files is given in Chapter 5 of the Toolbox Programming Manual, "Writing Software With VisiQuest 2001".

## E. Removing Toolbox References

This subform allows you to remove a reference to the currently selected toolbox in your VisiQuest environment. Removing a toolbox reference will prevent you from viewing or executing programs in that toolbox; you will not be able to see or modify its software objects using Craftsman and Composer, and its operators will not appear in VisiQuest. Removing a toolbox reference does not delete the toolbox itself, so you can always re-add the reference to the toolbox later if you wish.

## F. Klinting a Toolbox

This option allows you to run the **klint** program on the currently selected toolbox. Klint is a perl script which will check to see whether a toolbox is in a "good" state. It will check to see if the toolbox database matches the files in the toolbox's directories.

It reports any files found in the toolbox directory structure that are not part of the database, as well as any files referenced in the database that do not appear in the toolbox directory structure. In response to its output, you

should either delete files if they are extraneous, or add them to the toolbox's database using **craftsman** if they are valid.

Klint is particularly useful when getting a toolbox ready for release, and should be always be run before the toolbox is packaged for delivery.

## G. Migrating a Toolbox

If the currently selected toolbox is not up to date, the *Migrate Toolbox* menu selection will be activated. This selection will run the **kmigrate** program to bring the toolbox up to VisiQuest. See the VisiQuest Migration Manual for details on toolbox migration.

## H. Deleting a Toolbox

The last item on the *Toolbox* menu is *Delete Toolbox*. This item is only active if you have selected a toolbox from the list of toolboxes.

When deleting a toolbox you will first be prompted to confirm that you really do want to remove the toolbox. If you click on the 'Yes' button, the toolbox will be deleted, and the list of available toolboxes updated.

IMPORTANT NOTE: once a toolbox is deleted it cannot be recovered.

## I. Creating a Software Object

If you have selected a toolbox, the *Create Object* item on the object operations menu becomes active. Selecting this will bring up the object creation subform.

The first step when creating a new object is to select the Class type of the object to be created. It is assumed that you are familiar with the class types as described in the *Introduction to Toolbox Programming* chapter of this manual.

### I.1. Class Types

The VisiQuest software development system supports number of different software object class types. Selection of the class type is done using the "Select Classtype" walking menu. Standard VisiQuest software object class types include: kroutine, library, pane, script, workspace object, and xvroutine. A non-VisiQuest software object class type is also supported, the ANSI C/C++ standalone program.

It is assumed that you are familiar with these types, and the implications of a given object type. If not, please see Chapter 1, *Introduction to Toolbox Programming*, of this manual. More complete details on each software object class type is given in Chapter 5, *Writing Software With VisiQuest*.

When you select the Class type, the pane for creating software objects of that class type is displayed. Some class types have special attributes that are used only with that class type. For specific explanations of the attributes of each class type, see Chapter 5, *Writing Software With VisiQuest*, or click on the "Help" button that appears on the pane (this help button will display the man page for the appropriate code generator).

After filling out the values for the software object attributes, click on the Create button to create the software object and automatically generate code and documentation for it. The remainder of this section covers those software object attributes that are common to most class types.

## I.2. Object Name

An object's name is a single word comprised of alphanumeric characters and the underscore character, '\_'. Craftsman will not let you create an object with any other characters in the object name, since non-alphanumerics can cause problems in generated code and/or documentation.

## I.3. Binary or Archive Name

An object's binary name has slightly different interpretations depending on the object's type:

- For program objects, this attribute is the name of the executable.
- For a library object, this attribute specifies the name of the library archive. For example, the abstract widget library is a library with object name *xvwidgets*, but binary name of *xvw*. This means that the archive file is called *libxvw.a*, and on the link-line the library is included with `-lxvw`.

If the binary name is not specified it defaults to the object name.

## I.4. Icon Name

The *icon name* of an object is an optional attribute used to specify an alternate name for the object. If the icon name attribute is defined for an object, then VisiQuest will use that in preference to the object name. For example, the *kroutine kadd* might have an icon name of *Add*; or *kprdata* might be *Print Data*.

## I.5. Point of Contact Attributes

The author's *name* attribute is used to specify the name of the creator, owner, or *point of contact* for that object. This should be a full name, such as "Frank Sinatra". The author's *e-mail* attribute should be a fully-specified e-mail address for the author or point of contact. This attribute should be set on any object which is going to be distributed, since this attribute is by the bug reporting facility. If this attribute is not set, then the toolbox author attributes are used.

## I.6. Category & Subcategory

An object's *category* is used to classify the object, usually according to a broad domain of application. For example, a sobel edge detection routine might have a category of *Image Processing*. The *subcategory* attribute is used to further decompose all objects within a given category. So within the Image Processing category, a sobel edge detection routine might have a subcategory of *Edge Detectors*.

All objects must have a category and subcategory.

## I.7. Short Description

This is a short description of the software object. It is used in the documentation which is generated for the object, and the header of the main program. It is also printed when a user uses **kman -k**, and displayed when the software object appears in the FinderList of **VisiQuest**.

## I.8. Install in VisiQuest?

This attribute indicates whether or not the software object will appear as an operator in VisiQuest.

## J. Editing a Software Object

Having selected an object from the object list, the *Edit Object (Composer)* item on the object operations menu will become active. Selecting this item will invoke the software object editor, composer, on the selected object. Composer can also be run on an object by double clicking the mouse on that object in the object list. Composer is described in chapter 3 of the *Toolbox Programming* manual.

Craftsman will only run one composer for an object. If you select *Edit Object (Composer)* after composer is already running for that object, the composer window will be raised.

## K. Modifying Software Object Attributes

Selecting "Object Attributes ..." from the "Object" pulldown menu opens a pane displaying all of the modifiable attributes for the currently selected object. You may make changes to these attributes as required and then click on the "Apply Changes" button to update the object and regenerate its code.

## L. Copying a Software Object

Having selected an object from the object list, the *Copy Object* item on the object operations menu will become active. This operation is used to copy the currently selected object to another toolbox. Selection of this operation results in a subform, where you must select the destination toolbox. You must then hit the *Copy Object* button to invoke the copy function. As the copy is being made, a notifier window is displayed; this is removed when the operation completes.

If the selected object already exists in the destination toolbox, you will be prompted to choose whether you want to overwrite the object or abort the operation.

## M. Moving a Software Object

This menu item is used to move the currently selected software object to a different toolbox. The destination toolbox must be selected from the list on the Move Object subform. You must then hit the **Move Object** button to invoke the move function.

## N. Renaming a Software Object

This pane is used to rename the currently selected software object.

The new name must start with a letter of the alphabet, and can only contain letters or digits. The notifier will be displayed while the object is being renamed, since it takes a while.

## O. Linting The Software Object

This option allows you to run the **kint** program on the currently selected object. Kint is a perl script which will check to see whether a software object is in a "good" state. It will check to see if the object database matches the files in the object's directories.

It reports any files found in the object directory structure that are not part of the database, as well as any files referenced in the database that do not appear in the object directory structure. In response to its output, you should delete extraneous files, or add them to the object's database using composer.

Kint is particularly useful when getting a toolbox ready for release, and should be part of the release process.

## P. Deleting a Software Object

Having selected an object from the object list, the *Delete Object* item on the object operations menu will become active. If you select this item on the menu, you will be prompted to confirm that you really do want to delete the object. If you confirm deletion, the object will be deleted from the toolbox, and the object list will updated to reflect this change.

**IMPORTANT NOTE:** Deleting a software object is irreversibly destructive: all directories, files and information related to the object are deleted.



## Import MATLAB Code

Choose Object>Import MATLAB Code to access the MATLAB Integration wizard. Choose one of the following:

<XREF-section>MLibrary	A MLibrary object is a library available to the VisiQuest user to create new objects. It is not available in the Visual Programming Environment as a glyph.
<XREF-section>MRoutine	MRoutines are best suited for MATLAB code that has variables, inputs, and outputs.
<XREF-section>MXVRoutine	MXVRoutines are best suited for MATLAB code that is intended to be used as a standalone application.

### See Also

<XREF-section>MATLAB Integration Tutorial

### MLibrary

Complete the following to create a MLibrary Object.

1. Type the following in the General Object Information page, and click Next.

Toolbox Name	Select the toolbox in which to create the new object. This toolbox will determine the location in which the object files will be stored as well as the location of the object in Craftsman. Please note this location for future editing purposes.
Object Name	Specify the name of the object to be created in VisiQuest. This name will appear in the Craftsman tool, and will also be used in the directory structure for storing the object files.
Library Name	The name of the library.
Short Description	This description should be a short, concise statement of the function of the object. When a glyph is created from the object, this description will appear as the short description in the Glyph Explorer, as well as in the title bar of the pane. It is also used for keyword search within the VisiQuest system.
Author Name	The name of the person authoring the object.
Author Email	The email address of the object author.

2. In the Object Properties page, select the underlying language dependency for your generated VisiQuest MLibrary object, either "C" or "C++". Also choose whether or not to make the toolbox dependent on this object. Click Next.
3. In the Import MATLAB Files page, select the MATLAB files to be copied into the VisiQuest object. Include all source M files that are necessary for your functions to run. Click Next.
4. In the Generate Object page, click "Generate" to create the MATLAB object. Output from the MATLAB object generator appears in the dialog box. Once you have generated the MATLAB object, you cannot go back to change its properties with the wizard. Use Composer upon completion to change the object properties.

### MRoutine

Complete the following to create a MRoutine Object.

1. Type the following in the General Object Information page, and click Next.

Toolbox Name	Select the toolbox in which to create the new object. This toolbox will determine the location in which the object files will be stored as well as the location of the object in Craftsman. Please note this location for future editing purposes.
Object Name	Specify the name of the object to be created in VisiQuest. This name will appear in the Craftsman tool, and will also be used in the directory structure for storing the object files.
Binary Name	The name of the binary.
Glyph Name	The name of the glyph.
Short Description	This description should be a short, concise statement of the function of the object. When a glyph is created from the object, this description will appear as the short description in the Glyph Explorer, as well as in the title bar of the pane. It is also used for keyword search within the VisiQuest system.
Author Name	The name of the person authoring the object.
Author Email	The email address of the object author.

2. In the Object Properties page, select the underlying language dependency for your generated VisiQuest MRoutine object, either “C” or “C++”. Also choose whether or not to Install this MRoutine as a glyph in VisiQuest, and choose the Category and Subcategory. Click Next.
3. In the Import MATLAB Files page, select the MATLAB files to be copied into the VisiQuest object. Include all source M files that are necessary for your functions to run. Click Next.
4. In the Generate Object page, click “Generate” to create the MATLAB object. Output from the MATLAB object generator appears in the dialog box. Once you have generated the MATLAB object, you cannot go back to change its properties with the wizard. Use Composer upon completion to change the object properties.

### **MXVRoutine**

Complete the following to create a MXVRoutine Object.

1. Type the following in the General Object Information page, and click Next.

Toolbox Name	Select the toolbox in which to create the new object. This toolbox will determine the location in which the object files will be stored as well as the location of the object in Craftsman. Please note this location for future editing purposes.
Object Name	Specify the name of the object to be created in VisiQuest. This name will appear in the Craftsman tool, and will also be used in the directory structure for storing the object files.
Binary Name	The name of the binary.
Glyph Name	The name of the glyph.
Short Description	This description should be a short, concise statement of the function of the object. When a glyph is created from the object, this description will appear as the short description in the Glyph Explorer, as well as in the title bar of the pane. It is also used for keyword search within the VisiQuest system.
Author Name	The name of the person authoring the object.
Author Email	The email address of the object author.

2. In the Object Properties page, choose the following options and click Next:
  - a. Select the underlying language dependency for your generated VisiQuest MRoutine object, either "C" or "C++".
  - a. Choose whether or not to Install this MXVRoutine as a glyph in VisiQuest, and choose the Category and Subcategory.
  - a. Choose whether or not to create with batch option, and if checked, specify the Batch parameter.
  - a. Choose whether or not to call user-editable routine.
3. In the Import MATLAB Files page, select the MATLAB files to be copied into the VisiQuest object. Include all source M files that are necessary for your functions to run. Click Next.
4. In the Generate Object page, click "Generate" to create the MATLAB object. Output from the MATLAB object generator appears in the dialog box. Once you have generated the MATLAB object, you cannot go back to change its properties with the wizard. Use Composer upon completion to change the object properties.

## MATLAB Integration Tutorial

After you have finished running the "Import MATLAB Code" wizard in craftsman, there are still several things that you need to do in order to make your program functional. This includes setting the program CLUI (Command Line User Interface), writing the code that processes the CLUI, and writing the code that translates data between the VisiQuest Polymorphic Data Set and MATLAB mxArray.

### *Setting the Program CLUI*

All VisiQuest programs have a defined CLUI, which specifies the inputs and outputs of the program. Input can be parameters as well as data objects, while outputs, if present, are data objects. The code generators for mroutines and mxvroutines will create a default CLUI which includes a single input data object and a single output data object. In order for your new MATLAB integration program to be useful, you need to modify the CLUI created by the wizard to reflect the specific inputs and outputs of the integrated MATLAB functionality. This is done by running the

VisiQuest development tool guise (Graphical User Interface Specification Editor). The easiest way to launch guise for your new program is via the VisiQuest development tool composer. Simply expand the UIS node in composer's object tree view, and then double click on the .pane file for your program.

## Regenerating the Program

After you have finished editing your program's CLUI with guise, you must regenerate the program. This is done by opening the Commands window in composer via the Commands selection in the Options menu. In the Commands window, select the "Generate code" command, and click the "Run" button. This re-executes the code generator, which updates the program source code to reflect the changes in the CLUI parameters.

## Connecting the Inputs and Outputs

At this point, your program will have all the specified inputs and outputs, but does not know how to do anything with them. You need to edit your program in order to connect the command line parameters, convert inputs into appropriate MATLAB variables, execute the MATLAB code, and finally retrieve any outputs from the MATLAB code.

The VisiQuest MFile toolbox includes several sample MATLAB integration programs. The mroutine samples are exCHennon and exCppHennon. These are C and C++ implementations of the same functionality. The mxvroutine is exDisplayHennon. In addition, there are mllibrary samples, exCChaos and exCppChaos (C and C++ samples of the same functionality). We will use the exCHennon mroutine to walk you through the steps of interfacing the CLUI with the MATLAB function arguments.

Looking at the file <VisiQuest Install Directory>/Mfile/mfile/objects/mroutine/exCHennon/src/exCHennon.c, examine the function run\_exCHennon(). This is the function that does the work of the program. When it is called, the CLUI data structure has already been loaded. NOTE: The CLUI data structure for the program is defined in the main.h header file for the program. The MATLAB function that we are going to call is defined in the m-file mexCHennon.m, and has the following signature:

```
function img = mexCHenon(a,xLo,xHi,yLo,yHi,nLevels,numPts,varargin)
```

Looking at the top of the function run\_exCHennon(), we see that we have allocated local variables to hold the MATLAB variables for the mexCHennon function call, as well as the VisiQuest kobject required for the mroutine's output object:

```
kobject outobj;
kobject refout;
mxArray *img = NULL;
mxArray* a = NULL;
mxArray* xLo = NULL;
mxArray* xHi = NULL;
mxArray* yLo = NULL;
mxArray* yHi = NULL;
mxArray* nLevels = NULL;
mxArray* numPts = NULL;
mxArray* s = NULL;
```

The next thing that happens in the mroutine is the initialization of the MATLAB runtime environment. Note that this initialization depends on the version of MATLAB that is in use. The VisiQuest code generator creates this code for you, although you may need to modify it if you use more than one mfile or mllibrary in your mroutine. Please see the comments in the generated code.

```
/* Before doing anything else, initialize the MATLAB runtime
 * environment. The sequence for doing this changed slightly from
 * v5 to v6, and then changed COMPLETELY for v7.
 */
```

```

#if ! defined(KHOROS_MATLAB_VERSION)
    mlfEnterNewContext(0, 0);

#elif defined(KHOROS_MATLAB_VERSION) && KHOROS_MATLAB_VERSION == 6
    mlfEnterNewContext(0, 0);
    InitializeModule_mexCHenon();

#elif defined(KHOROS_MATLAB_VERSION) && KHOROS_MATLAB_VERSION == 7
    /*-- This array is used to pass the runtime options to
     * mclInitializeApplication(). These are the same options as
     * the -R parameter on the mcc compiler
     */
    char* mcropts[] = { "-nojvm" };

    /*-- Note that you may need to add the MATLAB directories to
     * your LD_LIBRARY_PATH environment variable in order to run
     * this program. Also, you will need to define the
     * XAPPLRESDIR environment variable in order to avoid warning
     * messages when you run this program (see the MATLAB Compiler
     * Guide for more information).
     */
    mclInitializeApplication((const char**) mcropts, (sizeof(mcropts)/
sizeof(char*)));

    /*-- Initialize the mfile shared library runtime logic.
     * If you use mfiles that have been packaged into separate
     * mlibrary software objects, then you'll need to call the
     * Initialize() function for each of those libraries at this
     * point, too.
     */
    libexCHenonInitialize();
#endif

```

Finally, we can convert our CLUI inputs to the MATLAB input variables, and call the C-version of the mexCHenon function. Again, there are differences, as outlined in the code, in how this done for the different versions of MATLAB:

```

/*-- Now set up the arguments for the henon() mfile function.
 * It's been wrapped in C code, so we call the mlfMexCHenon()
 * wrapper. Of the 8 arguments to henon(), 7 are required.
 * However, since the exCHenon CLUI makes the corresponding CLUI
 * arguments optional, and provides a default value for
 * each one, I can just blindly use the CLUI values for those 7.
 * The 8th argument to henon() is different, however: it's optional
 * in the m-code, and I want to pass a NULL value to mlfMexCHenon()
 * when the CLUI parameter is omitted (that causes henon() to use
 * the time-of-day clock to seed the random number generator).
 */
#if defined(KHOROS_MATLAB_VERSION) && KHOROS_MATLAB_VERSION == 7
    a = mxCreateDoubleScalar( clui_info->aa_double );
    xLo = mxCreateDoubleScalar( clui_info->xLo_double );
    xHi = mxCreateDoubleScalar( clui_info->xHi_double );
    yLo = mxCreateDoubleScalar( clui_info->yLo_double );
    yHi = mxCreateDoubleScalar( clui_info->yHi_double );
    nLevels = mxCreateDoubleScalar( (double) clui_info->nLevels_int );
    numPts = mxCreateDoubleScalar( (double) clui_info->numPts_int );
    if (clui_info->s_flag)
        s = mxCreateDoubleScalar( (double) clui_info->s_int );

```

```

    mlfMexCHenon(1, &img, a, xLo, xHi, yLo, yHi, nLevels, numPts, s);
#else
    a = mlfScalar( clui_info->aa_double );
    xLo = mlfScalar( clui_info->xLo_double );
    xHi = mlfScalar( clui_info->xHi_double );
    yLo = mlfScalar( clui_info->yLo_double );
    yHi = mlfScalar( clui_info->yHi_double );
    nLevels = mlfScalar( (double) clui_info->nLevels_int );
    numPts = mlfScalar( (double) clui_info->numPts_int );
    if (clui_info->s_flag)
        s = mlfScalar( (double) clui_info->s_int );

    img = mlfMexCHenon(a, xLo, xHi, yLo, yHi, nLevels, numPts, s);
#endif

```

Now we need to retrieve the results of the functions call, and save the data in our output object. Not ethe usage of the mxGetN(), mxGetM(), and mxGetPr() functions for accessing the data returned by the MATLAB function call.

```

    /*-- It's always a good idea to wrap VisiQuest library function
    * calls in a ktry/kcatch context.
    */
    ktry_begin(NULL, "run_exPutHenon routine");

    if (kcatch(KANY_ERROR, khandle_error, NULL))
    {
        kprintf("VisiQuest exception thrown during exPutHenon processing -
processing failed.\n");
        return FALSE;
    }

    /* create reference to output object */
    outobj = kpds_open_output_object(clui_info->o_file);
    kpds_create_value(outobj);
    refout = kpds_reference_object(outobj);

    kpds_set_attribute(refout, KPDS_VALUE_DATA_TYPE, KDOUBLE);

    /* Size() returns the number of rows and columns
    * in the mxArray object passed back from mexCppHenon()
    */
    kpds_set_attribute(refout, KPDS_VALUE_SIZE, mxGetN(img), mxGetM(img), 1,
1, 1);

    /* Need to reset index order since mxArray objects use
    * column-major indexing */
    kdms_set_attribute(refout, KDMS_SEGMENT_VALUE, KDMS_INDEX_ORDER, order);

    /* mxGetPr(m.GetData) returns a pointer to the data in the mxArray
    * object, m, passed back from mlfMexCHenon() */
    kpds_put_data(refout, KPDS_VALUE_ALL, (kaddr)mxGetPr(img));

    kpds_close_object(refout);
    kpds_close_object(outobj);

    ktry_end();

```

Finally, we need to clean up an allocated data, and terminate the MATLAB runtime environment. Again, there are differences in how this must be done, depending on which version of MATAB is in use. Also, if you used more than one mfile function, or mlibrary, you will need to call the appropriate termination function for each (see `TerminateModule_mexCHennon()` and `libexCHennonTerminate()` below).

```

/* clean up all the storage that was returned to me by the
 * mfile code */
mxDestroyArray(img);

/* clean up all the storage that I allocated.
 * Note that MATLAB 6 keeps track of these things in the context,
 * and will destroy them automatically, so I must NOT destroy these
 * in MATLAB 6.
 * After cleaning up the storage that I allocated, go through
 * the shutdown sequence.
 */
#if ! defined(KHOROS_MATLAB_VERSION)
mxDestroyArray(a);
mxDestroyArray(xLo);
mxDestroyArray(xHi);
mxDestroyArray(yLo);
mxDestroyArray(yHi);
mxDestroyArray(nLevels);
mxDestroyArray(numPts);
if (s != NULL)
    mxDestroyArray(s);

mlfRestorePreviousContext(0, 0);

#elif defined(KHOROS_MATLAB_VERSION) && KHOROS_MATLAB_VERSION == 6
TerminateModule_mexCHenon();
mlfRestorePreviousContext(0, 0);

#elif defined(KHOROS_MATLAB_VERSION) && KHOROS_MATLAB_VERSION == 7
mxDestroyArray(a);
mxDestroyArray(xLo);
mxDestroyArray(xHi);
mxDestroyArray(yLo);
mxDestroyArray(yHi);
mxDestroyArray(nLevels);
mxDestroyArray(numPts);
if (s != NULL)
    mxDestroyArray(s);

libexCHenonTerminate();
mclTerminateApplication();
#endif

```

## ***Build and Test Your Program***

Now that you have written the integration code, you can compile and execute your program. To do this, select the Options Menu, Commands item in composer to open the Commands window. Select the command "kmake install", then click the run button. This will build your program and install it in the VisiQuest system. Now oyu can run and test your new VisiQuest-MATLAB program. If you created the program for use within the VisiQuest Visual Programming Environment, it will now be available as a glyph.

## ***Additional Information***

For additional information on the VisiQuest MATLAB Integration Toolbox, see the VisiQuest MFile Manual.

For additional information on using the VisiQuest development tools, see the Toolbox Programming Manual.

For additional information on using the VisiQuest Polymorphic Data Services, see the Programming Services Manual, Volume II.



## Q. Commands Subform

The *Commands* subform is accessed from the Options pulldown menu on craftsman's menubar and may be used to invoke various commands on the toolbox object being edited. This includes code generation as well as compilation.

A console widget is used to capture the output (stdout and stderr) of every run. Examine the contents of the console widget after each command to see the result of the operation.

### Q.1. Command Options

#### Q.1.1. Clear console on command?

Clears the console on each new command.

#### Q.1.2. Make All Archs

When VisiQuest is set up so that the toolbox is compiled on a number of architectures, this flag can be set so that the "Make" menu is updated with the "kmakeall" commands that are used to compile on multiple architectures.

### Q.2. Make

The "Generate Code" button on the *Make* menu allows you to run the code generators. The rest of the buttons on the *Make* menu allow you to use *kmake* to compile and install your toolbox object; it provides access to a variety of kmake targets.

#### Q.2.1. Generate Code

The *Generate Code* button is used to invoke the code generation routines on the toolbox object being edited.

Code generation includes:

- the code framework for the toolbox object
- html, man and help pages
- driver code for xv routines with an automatically generated GUI. You can also regenerate code from the command-line using the following commands:

```
% cd $TOOLBOX/objects/{classtype}/{oname}/src
% kmake regen
```

## Q.2.2. Kmake

This kmake target will compile your source files, and either link to generate an executable, or generate a library archive.

## Q.2.3. Kmake Install

This will install the executable for your object in the **bin** directory of the toolbox containing the object, or install the library archive in the **lib** directory.

## Q.2.4. Kmake Install-Hdrs

Used with library objects, this will install the include files of your library, copying them from **\$TOOLBOX/objects/library/{oname}/{oname}/** to the **\$TOOLBOX/include** directory.

## Q.2.5. Kmake Makefiles

This target will regenerate the object's Pmakefiles.

## Q.2.6. Kmake Struct

This target will execute **kgenstruct** to parse any \*.x specification file defining data services structure and generate C code for translating an instance of that structure between memory and a file. For more information, do

```
% kman kgenstruct
```

and read about Data Services Structure Extensions in Chapter 6, "Extensions" of the *Data Services Manual*.

## Q.2.7. Kmake Clean

This target will remove all object code and other intermediate or temporary files.

## Q.2.8. Kmake Srcmachs

Used with multiple architecture setups only, this target creates symbolic links of all the files in the toolbox object in the srcmach directories for each supported architecture.

## Q.2.9. Kmake Uninstall

This target will remove all binaries and/or library archives associated with the software object.

## Q.3. Search

Searches all files in the toolbox object for the string provided.

#### **Q.4. Clear**

Clears the console window.

#### **Q.5. Save**

Allows you to save the contents of the console window to a file.

### **R. Shell**

Selecting "Shell" from the Options menu will create an xterm containing a system shell from which you may issue commands from the command line. Note that this feature only is relevant for the UNIX version of VisiQuest Pro 2001.

### **S. Setting Preferences**

The **Preferences** subform is accessed from craftsman's **Edit** menu. The craftsman preferences allow you to specify which object types you want to show and which you don't want to show.

#### **S.1. Preferences: Show Object Types**

By default, when you select a toolbox, the right-hand list in Craftsman's main window will display all of the software objects in the selected toolbox. The *Show Object Types* selection is used to request that only software objects of certain classtypes be listed. This option can be useful when you are working with a toolbox which contains a large number of software objects.

# Table of Contents

A. Introduction to Craftsman . . . . .	2-1
A.1. Documentation Prerequisites . . . . .	2-1
A.2. Command-line Arguments . . . . .	2-2
A.3. Craftsman's User Interface . . . . .	2-2
A.4. Toolbox Operations . . . . .	2-3
A.5. Object Operations . . . . .	2-3
B. Creating a new Toolbox . . . . .	2-5
B.1. Toolbox Attributes . . . . .	2-5
B.2. Creating the toolbox . . . . .	2-5
C. Adding Toolbox References . . . . .	2-6
D. Changing Toolbox Attributes . . . . .	2-6
E. Removing Toolbox References . . . . .	2-7
F. Klinting a Toolbox . . . . .	2-7
G. Migrating a Toolbox . . . . .	2-8
H. Deleting a Toolbox . . . . .	2-8
I. Creating a Software Object . . . . .	2-8
I.1. Class Types . . . . .	2-8
I.2. Object Name . . . . .	2-9
I.3. Binary or Archive Name . . . . .	2-9
I.4. Icon Name . . . . .	2-9
I.5. Point of Contact Attributes . . . . .	2-9
I.6. Category & Subcategory . . . . .	2-10
I.7. Short Description . . . . .	2-10
I.8. Install in VisiQuest? . . . . .	2-10
J. Editing a Software Object . . . . .	2-10
K. Modifying Software Object Attributes . . . . .	2-10
L. Copying a Software Object . . . . .	2-10
M. Moving a Software Object . . . . .	2-11
N. Renaming a Software Object . . . . .	2-11
O. Linting The Software Object . . . . .	2-11
P. Deleting a Software Object . . . . .	2-11
Q. Commands Subform . . . . .	2-12
Q.1. Command Options . . . . .	2-12
Q.1.1. Clear console on command? . . . . .	2-12
Q.1.2. Make All Archs . . . . .	2-12
Q.2. Make . . . . .	2-12
Q.2.1. Generate Code . . . . .	2-12
Q.2.2. Kmake . . . . .	2-13
Q.2.3. Kmake Install . . . . .	2-13
Q.2.4. Kmake Install-Hdrs . . . . .	2-13
Q.2.5. Kmake Makefiles . . . . .	2-13
Q.2.6. Kmake Struct . . . . .	2-13
Q.2.7. Kmake Clean . . . . .	2-13
Q.2.8. Kmake Srcmachs . . . . .	2-13
Q.2.9. Kmake Uninstall . . . . .	2-13
Q.3. Search . . . . .	2-13
Q.4. Clear . . . . .	2-14

Q.5. Save . . . . . 2-14  
R. Shell . . . . . 2-14  
S. Setting Preferences . . . . . 2-14  
    S.1. Preferences: Show Object Types . . . . . 2-14

# *Toolbox Programming*

## **Chapter 3**

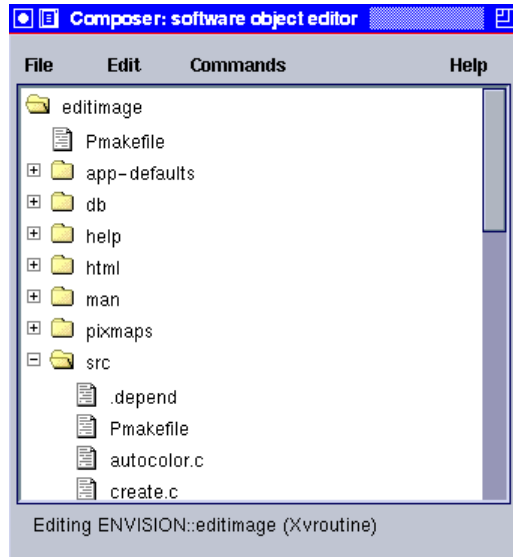
# **Composer**

## **Software Object Management**

Copyright (c) AccuSoft Corporation, 2004. All rights reserved.



# Chapter 3 - Composer



**Figure 1:** The master form for composer, when invoked on editimage, which is in the **envision** toolbox.

## A. Introduction to Composer

The composer software object editor is used to edit, manipulate and compile existing software objects. Software objects are created using craftsman. This chapter assumes that you have read the previous chapter on craftsman.

### A.1. The Composer Model

A software object can be viewed as a collection of files. The files in an object are organized in a directory structure related to how the files are used in the software object. For example, source code files for a kroutine object would be in the `src` directory, `pixmaps` in the `pixmaps` directory.

Although most files are ASCII text, there are also other ways of viewing and editing the information represented. For example, a UIS file can be edited with a text editor, or with `guise`. Composer also provides access to the code generators, generic file operations, and `make`.

All software objects have a number of attributes associated with them that are used to describe the object, and provide additional configuration information, such as that required by other parts of the system. Composer can



be used to modify a subset of these attributes.

## A.2. Command-line Arguments

When invoking composer you must provide the name of a toolbox, and an object within that toolbox. For example, to edit editimage, which is in the **envision** toolbox, you would issue the following command:

```
% composer -tb envision -oname editimage
```

Composer understands the standard VisiQuest command-line switches, so you can always use the `-U` switch to see the usage statement. Composer can be invoked from within Craftsman via the *Edit Object* option on the Object menu.

## A.3. Composer's User Interface

There are three major regions in the user interface:

- The toolbox, object name, and class type are displayed at the bottom of the form. In this case, the toolbox is ENVISION, the object name is editimage, and the class type is Xvroutine.
- The center of the form is a scrolled treelist which presents the files and directory structure of the object being edited. Directories may be opened and closed as desired. Clicking the left mouse button on a file name or directory selects that file and its directory. Any subsequent operations will act on the selected file and/or directory.
- There is a menubar at the top of the user interface. Clicking *File* on the menubar displays a pull-down menu of actions that can be invoked on the selected file. Actions are activated or deactivated depending on whether a file is selected or not. The Quit button is also on the File menu.

## B. Changing Object Attributes

The *Object Attributes* item on the Edit menu brings up a subform which allows you to change the modifiable attributes of a software object.

When the subform comes up it will be filled out according to the current values of the attributes. If you change any of the attributes, you must click on the *Apply Changes* button for the changes to be saved in the object's database. Changes will be lost if you do not use the *Apply Changes* button.

## C. The Options Menu

The *Options* menu allows the user to execute commands, either on the Commands subform or in a Shell (xterm) window.

## D. The Commands Subform

The *Commands* subform is used to invoke various commands on the software object being edited. This includes code generation as well as compilation.

A console widget is used to capture the output (stdout and stderr) of every run. Examine the contents of the console widget after each command to see the result of the operation.

### D.1. Command Options

#### D.1.1. Clear console on command?

Clears the console on each new command.

#### D.1.2. Make All Archs

When VisiQuest is set up so that the software is compiled on a number of architectures, this flag can be set so that the "Make" menu is updated with the "kmakeall" commands that are used to compile on multiple architectures.

### D.2. The Make Menu

The "Generate Code" button on the *Make* menu allows you to run the code generators. The rest of the buttons on the *Make* menu allow you to use *kmake* to compile and install your software object; it provides access to a variety of *kmake* targets.

#### D.2.1. Generate Code

The *Generate Code* button is used to invoke the code generation routines on the software object being edited.

Code generation includes:

- the code framework for the software object
- html, man and help pages
- driver code for xv routines with an automatically generated GUI. You can also regenerate code from the command-line using the following commands:

```
% cd $TOOLBOX/objects/{classtype}/{oname}/src
% kmake regen
```

## D.2.2. Kmake

This kmake target will compile your source files, and either link to generate an executable, or generate a library archive.

## D.2.3. Kmake Install

This will install the executable for your object in the **bin** directory of the toolbox containing the object, or install the library archive in the **lib** directory.

## D.2.4. Kmake Install-Hdrs

Used with library objects, this will install the include files of your library, copying them from **\$TOOLBOX/objects/library/{oname}/{oname}/** to the **\$TOOLBOX/include** directory.

## D.2.5. Kmake Makefile

This target will regenerate the object's Pmakefiles.

## D.2.6. Kmake Klint

This option allows you to run the **klint** program on the currently selected object. Klint is a perl script which will check to see whether a software object is in a "good" state. It will check to see if the object database matches the files in the object's directories.

It reports any files found in the object directory structure that are not part of the database, as well as any files referenced in the database that do not appear in the object directory structure. In response to its output, you should delete extraneous files, or add them to the object's "cms.db" database file.

## D.2.7. Kmake Struct

This target will execute **kgenstruct** to parse any \*.x specification file defining data services structure and generate C code for translating an instance of that structure between memory and a file. For more information, do

```
% kman kgenstruct
```

and read about Data Services Structure Extensions in Chapter 6, "Extensions" of the *Data Services Manual*.

## D.2.8. Kmake Clean

This target will remove all object code and other intermediate or temporary files.

## D.2.9. Kmake Srcmachs

Used with multiple architecture setups only, this target creates symbolic links of all the files in the software

object in the srcmach directories for each supported architecture.

## D.3. Other Features

Other features available on the Commands subform include the following:

### D.3.1. Search

Searches all files in the software object for the string provided.

### D.3.2. Clear

Clears the console window.

### D.3.3. Save

Allows you to save the contents of the console window to a file.

## E. File Operations

The *File* menu provides access to a number of routines to perform various actions on files. These are described below.

### E.1. Editing Files

Under the File pulldown menu, there are up to three types of edit/view supported per file type, and a selection of generic file operations. All editors are accessed via the *File* menu. Each is discussed below.

#### E.1.1. Special Edit

Selecting *Special Edit* invokes a special editing program (where appropriate) for the selected file. For example, for a User Interface Specification file, the appropriate special editor is *guise* (discussed in Chapter 4 of this manual). *Guise* provides a visual representation of a program's GUI that can be edited by adding, deleting, resizing, or moving interface objects directly. For html files, if the environment variable `VisiQuest_BROWSER` is set, this browser will be used as the special editor. For example, if you want to use *netscape* when editing html files, you might set `VisiQuest_BROWSER` as follows:

```
% setenv VisiQuest_BROWSER "netscape %f"
```

If `VisiQuest_BROWSER` is not set, `VisiQuest_EDITOR` (described below) is used.

#### E.1.2. Edit

The *Edit* menu item is used to invoke a text editor on ASCII files. Composer follows three steps when determining which editor to use:

- If you have set the `VisiQuest_EDITOR` environment variable, then composer interprets that string as the command to run, and assumes that if your editor needs a window to run in, such as an `xterm`, then your command string includes this. The string must contain at least one occurrence of the string `'%f'`, which will be interpolated to the name of the currently selected file. If the string contains the sequence `'%g'`, then this will be interpolated to an X style geometry specification. When editing UIS files, the geometry will be set to a short, wide window; when editing source files, a long, regular width window is used. For example, if you want to use `vi` when editing files, you might set `VisiQuest_EDITOR` as follows:

```
% setenv VisiQuest_EDITOR "xterm -geometry %g -title '%f' -e vi %f"
```

or for `emacs`:

```
% setenv VisiQuest_EDITOR "emacs -geometry %g -xrm '*title: %f' %f"
```

- If `VisiQuest_EDITOR` is not set, composer will look for the `EDITOR` environment variable, and if defined, will run that command inside an `xterm`.
- As a last resort, composer will try to invoke `vi` inside an `xterm`. This assumes that both `vi` and `xterm` are on your path.

### E.1.3. View

This menu item is used to view, or preview, the currently selected file. In most cases this will just bring up a text viewer on the raw ASCII file. For UIS files a non-editable version of the GUI is created. For html files, the browser specified by the `VisiQuest_BROWSER` environment variable is used.

### E.2. Add ...

This subform is used to add a new file to the software object.

To create and add a previously non-existent file to the software object, enter the name of the file in the "File To Add" selection and click on "Add File" or enter `<return>`. The new file will be created from a template and added to the software object.

To add an existing file to the software object's database, enter the name of the file in the "Existing File" selection and click on "Add File". For example, suppose there is source code in a `*.c` file that you wish to include in the current software object. Entering the path to the `*.c` file and clicking on "Add File" will cause composer to copy the file into your software object and update its database to register the new file.

### E.3. Copy ...

This pane is used to copy the currently selected file. Enter the new name of the file and click on "Copy File"

to copy the file.

## E.4. Delete

On selecting this operation, you will first be prompted to confirm that you really do want to delete the currently selected file. If you do, the file will be deleted and references to it removed from the object. Composer's list of files is updated to reflect the change.

**Note:** be careful when using this feature. Composer will not give you any warnings as to the potential consequences of removing the selected file.

## E.5. Rename ...

This pane is used to rename the currently selected file. Enter the "New Name" followed by <return> or click on "Rename File". You will be prompted for confirmation before the file is renamed.

## E.6. Print ...

This pane is used to print the currently selected file. You can provide the name of the printer you wish to use. For example, if you entered *zippy* in the **Printer** selection, then composer will run the command:

```
lpr -Pzippy selected-file
```

You can also use the **Print Command** selection to specify a different print command. For example, you might enter the following command:

```
a2ps -p -1 -Pzippy
```

Composer will append the name of the selected file to the command specified, and run that instead of *lpr*. To specify a **Print Command**, you must first press the toggle to the left of the Print Command selection.

**This page left intentionally blank**

# Table of Contents

A. Introduction to Composer . . . . .	3-1
A.1. The Composer Model . . . . .	3-1
A.2. Command-line Arguments . . . . .	3-2
A.3. Composer's User Interface . . . . .	3-2
B. Changing Object Attributes . . . . .	3-2
C. The Options Menu . . . . .	3-2
D. The Commands Subform . . . . .	3-3
D.1. Command Options . . . . .	3-3
D.1.1. Clear console on command? . . . . .	3-3
D.1.2. Make All Archs . . . . .	3-3
D.2. The Make Menu . . . . .	3-3
D.2.1. Generate Code . . . . .	3-3
D.2.2. Kmake . . . . .	3-4
D.2.3. Kmake Install . . . . .	3-4
D.2.4. Kmake Install-Hdrs . . . . .	3-4
D.2.5. Kmake Makefile . . . . .	3-4
D.2.6. Kmake Klint . . . . .	3-4
D.2.7. Kmake Struct . . . . .	3-4
D.2.8. Kmake Clean . . . . .	3-4
D.2.9. Kmake Srcmachs . . . . .	3-4
D.3. Other Features . . . . .	3-5
D.3.1. Search . . . . .	3-5
D.3.2. Clear . . . . .	3-5
D.3.3. Save . . . . .	3-5
E. File Operations . . . . .	3-5
E.1. Editing Files . . . . .	3-5
E.1.1. Special Edit . . . . .	3-5
E.1.2. Edit . . . . .	3-5
E.1.3. View . . . . .	3-6
E.2. Add ... . . . .	3-6
E.3. Copy ... . . . .	3-6
E.4. Delete . . . . .	3-7
E.5. Rename ... . . . .	3-7
E.6. Print ... . . . .	3-7



**This page left intentionally blank**

# *Toolbox Programming*

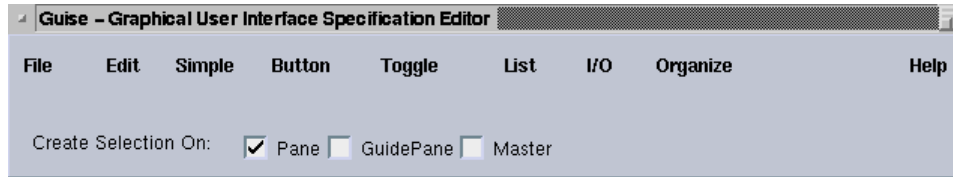
## **Chapter 4**

### **Guise**

#### **User Interface Creation & Maintenance**



# Chapter 4 - Guise



**Figure 1:** The guise GUI specification editor allows the user to input and display an existing UIS file, or to create a new one. Selections may be added, old selections deleted, or existing selections changed.

## A. Overview

guise is a direct-manipulation Graphical User Interface (GUI) design tool for use with all VisiQuest software objects. Guise allows you to create and modify a User Interface Specification (UIS) file, which defines the GUI and/or Command Line User Interface (CLUI) of your program. Guise can be run independently, or can be accessed via composer.

A menubar interface has been introduced for use in guise. From left to right, the menus are: *File*, *Edit*, *Simple*, *Button*, *Toggle*, *List*, *I/O*, *Organize*, and *Help*.

## B. Short Description of Menus & Contents

### B.1. The File Menu

The file menu contains the *New*, *Open...*, *Save*, *Save As...*, *Reload*, and *Quit* menu selections.

- New* allows you to start your GUI editing over with a fresh blank GUI template.
- The *Open...* subform is where an input file can be specified.
- Save* will appear as "Save (Needed)" whenever you have made changes to the currently displayed GUI. This action will save to the current file name.
- Save As...* allows you to save to a different file name.
- Reload* is for loading the most recently saved version of the GUI you are currently editing.
- The *Quit* menu selection is used to exit guise.

## B.2. The Edit Menu

- *Delete* deletes a currently selected GUI item.
- *Copy* duplicates a currently selected GUI item.
- *Edit* selection opens the editing subform for the currently selected GUI item.
- *Edit File Manually* spawns an edit session (like "vi" or "emacs") in which the GUI file currently loaded can be edited directly.
- *Options* opens the *Options* subform which contains the "Force Over-Write" and "Move/Resize Multiple Panes Together" toggles.

## B.3. The Simple Menu

The *Simple* menu contains the *Flag*, *Logical*, *Integer*, *Float*, *Double*, *String*, *Integer Array*, *Float Array*, *Double Array*, *Workspace*, and *Label* menu selections. Each of these selections, when highlighted, will place the respective type of variable, a workspace, or a label on the current pane, guide pane, or master form being edited.

## B.4. The Button Menu

The *Button* menu contains the *Action*, *Routine*, *Help*, and *Quit Button* menu items. Each of these items, when selected, will place the respective type of button on the current pane, guide pane, or master form being edited.

## B.5. The Toggle Menu

The *Toggle* menu contains the *Flag*, *Logical*, *Integer*, *Float*, *Double*, *String*, *Input File* and *Output File* menu items. Each of these items, when chosen, will place the selected type of toggle on the current pane, guide pane, or master form being edited.

## B.6. The List Menu

The *List* menu contains the *Cycle*, *List*, *DisplayList*, and *StringList* menu selections. Each of these selections, when highlighted, will place the respective type of list on the current pane, guide pane, or master form being edited.

## B.7. The I/O Menu

The *I/O Menu* contains the *Input File*, *Output File*, *Standard Input*, and *Standard Output* menu selections. Each of these selections, when highlighted, will place the respective type of list on the current pane, guide

pane, or master form being edited.

## B.8. The Organize Menu

The *Organize* menu contains the *Create ME Group (required)*, *Create ME Group (optional)*, *Create MI Group*, *Create Loose Group*, *Create Submenu*, *New Pane*, and *Create Master* menu selections. These selections will create the respective groups using allowable items which are selected on the current pane, guide pane, or master form being edited.

## B.9. The Help Menu

selections. The *Help* selection executes khelp, which displays the help page relevant to the page on which that particular *Help* button is located. When selected from the menubar, it khelp will display this document. The License selection displays the VisiQuest License agreement, also via khelp.

Kroutine, xvroutine, pane, and some script software objects have at least one UIS file. The UIS file is comprised of lines made up of fields in a strictly defined syntax. Each UIS line describes either a program argument on the CLUI, an item on the GUI, or both. The UIS was designed at a high level of abstraction; each type of line contains the information necessary to completely describe that type of item in the GUI, as well as in the CLUI. Thus, from a single UIS file for a particular application, the code for both the CLUI and the GUI may be generated directly. The UIS file is used for three distinct purposes in VisiQuest:

1. It defines the CLUI of all VisiQuest programs.
2. It defines the GUI of the program that is used when the program is run using the [-gui] option; this same GUI is used to integrate the program into the VisiQuest visual language.
3. For interactive X-Windows based applications (xvroutines), it defines the interactive GUI of the application.

For *kroutines*, a single UIS file will suffice for all purposes. *Xvroutines* often use two UIS files: one to define the CLUI of the program and to integrate it into VisiQuest, and another to define the more sophisticated independent GUI of the program. For *pane objects*, the UIS file provides the alternate interface to the base program of the pane object. For *script objects*, the UIS file is used to integrate the script into VisiQuest. For more detailed information about the UIS file, see Appendix A of the Toolbox Programmer's Manual.

## C. Start Up

```
% guise

[-force]  (flag)    force output
[-o]      (outfile) output UIS file
           (default = new.pane)

[Optional Mutually Exclusive Group - Specify one or none of the 2 options:]
[-i] (infile) initial UIS file
[Optional Mutually Inclusive Group - Specify all or none of the 3 options:]
```

```

[-tb]      (string) toolbox
[-oname]   (string) object name
[-uis]     (string) uis file
           pane (for *.pane file),
           or form (for *.form file)
           [default: pane]

[-V]      (flag)      gives the version number of the program
[-U]      (flag)      gives the usage of the program
[-P]      (flag)      interactive prompting for arguments
[-gui]    (flag)      run from GUI as defined in *.pane file
[-x]      (integer)   x value for GUI autoplacement
[-y]      (integer)   y value for GUI autoplacement
[-A]      (outfile)   Creates an answer file
[-a]      (infile)    Uses an answer file
[-ap]     (infile)    prints answer file values

```

**[-i {UIS file}] OR [-tb {toolbox} -oname {object} -uis {'pane'/'form'}]**

When started without the [-i] and without the [-tb], [-oname] and [-uis] arguments, **guise** begins by creating an empty pane. When started with the [-i] option specifying an input UIS file, **guise** displays the GUI defined by that UIS file and allows you to modify it; the output filename is taken from the input file unless otherwise specified. When started with the [-tb], [-oname], and [-uis] arguments, **guise** will open the specified program object in the specified toolbox, and will look for the \*.pane or \*.form file as dictated by the [-uis] option. Assuming the implied UIS file is found, **guise** displays the GUI defined by that UIS file, and allows you to modify it. The output filename is taken from the input file unless otherwise specified. The only difference between using the [-i] argument and the [-tb], [-oname], and [-uis] arguments is that, in the former case, you must know the path to the desired UIS file, while in the latter case, you must know the toolbox and program with which the UIS file is associated.

**[-o {UIS file}]**

The [-o] argument specifies the name of the output UIS file. When the [-i] or [-tb], [-oname], and [-uis] options are used to specify an input UIS file, the output filename is taken to be the same as the input unless otherwise specified. Therefore, either the [-o] argument should be used or care should be taken to change the output filename if the input file is not to be affected by changes made using **guise**.

**[-force]**

The [-force] option may be used to suppress prompting before the output file is overwritten. Setting the "Force Over-Write?" selection on the **guise** GUI to True will achieve the same effect.

**[-x {value}] [-y {value}]**

The *x* and *y* arguments are usually specified as a pair. Together, they indicate the automatic placement of the GUI at the (x,y) location, where *x* and *y* are given as device (screen) coordinates. By default, *x* and *y* are both -1, indicating manual placement of the GUI.

**[-V]**

This argument will print out the current version of the application.

**[-U]**

This argument will print out the command line argument structure for the application in less detail than is given here.

**[-P]**

This argument will enable interactive prompting for each of the command line arguments to the program.

**[-gui]**

This argument provides a graphical alternative to the interactive prompting mode described above; it brings up the GUI of the application as defined in its \*.pane file. You may enter values for the start-up arguments as desired. When they are complete, click on the *Run* button to start the program.

**[-A {filename}]**

This argument will create an answer file containing the specified values of command line arguments for later use with the [-a] option. This CLUI answer file will be named as specified if a filename is given. If no filename is specified, the default filename "\$HOME/VisiQuest.ans" will be used. Note: the "-A" argument is often used in conjunction with the "-P" argument; you are interactively prompted for arguments to the application, and those arguments are saved for later executions.

**[-a {filename}]**

This argument will use the answer file values for the command line arguments as they were saved earlier in a CLUI answer file. The specified CLUI answer file will be used if a filename is given. If no filename is specified, the default file "\$HOME/VisiQuest.ans" will be used. No other command line arguments should be necessary when the "-a" argument is used.

**[-ap {filename}]**

This argument will query the saved CLUI answer file for the command line argument values and echo them. The specified CLUI answer file will be queried if a filename is given. If no filename is specified, the default filename "\$HOME/VisiQuest.ans" will be used.

## D. GUI Edit Mode

Before you can use the direct-manipulation capability of any VisiQuest GUI, it must first be in *edit mode*. When a GUI is in edit mode, the backplane will display a grid where each square is 1/2 character width by 1/2 character height. To change from "edit mode" to "normal mode" or vice versa, use the "meta-left-mouse-button click" or "meta-click" for short. This means that you must hold down the "meta" key (the key which acts as "meta" on your keyboard will depend on your keyboard mapping -- most common are "Alt," "Compose," "Ctrl," and "Shift"), and simultaneously click the left mouse button on the *backplane* of the GUI. To put the GUI back into "normal mode," meta-click on the *backplane* again."

When a GUI is in edit mode, you may "select" an item on the GUI by clicking the left mouse button on it. When an item is "selected," its corners are bracketed with control markers; the meaning of button clicks within it are changed so that you can change its size, location, and attributes. Changing the GUI back to normal mode will cause all the selections on the GUI to work normally again; button clicks on the selections act as they

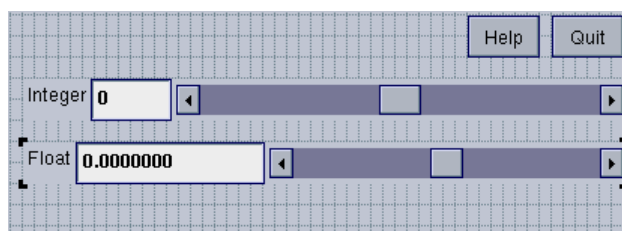


would in a finalized graphical user interface, e.g., the *Quit* button dismisses the GUI.

If a selection is currently in edit mode and you click (without using the meta key) on *another selection*, the second selection will be "selected" while the first selection is "unselected." If you *meta-click* on the second selection, the second selection and the first selection will both be "selected" and you will be able to move them as a pair. Using the meta-click while in edit mode will allow you to create groups to be moved as a unit.

This capability supports the creation of mutually inclusive groups, mutually exclusive groups, loose groups, and submenus from within *guise*. Note that the creation of nested groups is not supported.

Note that the GUI of ANY VisiQuest xvroutine may be put in edit mode by the user. When *guise* or *preview* is used, "full edit mode" is put on, and *any* attribute of any item on the GUI may be changed. In contrast, end users of an xvroutine may put the application in edit mode, but they will only be able to change those attributes of its selections that will not affect the performance of the application in any way. "Non-critical" attributes, such as title and geometry, are set by the end user of the application, while "critical" attributes, such as the variable name, are only editable by the developer via *guise* or *preview*. The capability for allowing the end user access to non-critical attributes of the GUI provides the end user of an application a mechanism with which to "personalize" the GUI of an application as desired; the [-form] option which is automatically built into every VisiQuest xvroutine and is provided specifically for this purpose.



---

**Figure 2:** Put a GUI into "edit mode" by doing a "meta-click" (clicking with the left mouse button while holding the "Shift" key down) on its *backplane*. The backplane of this GUI encompasses the entire area of the GUI that is not part of the *Integer* selection, the *Float* selection, the *Help* button, or the *Quit* button. Here, the GUI has been put into "edit mode;" this is shown by the grid displayed on its backplane. The *Integer* selection has been "selected;" its corners are bracketed with control markers. Clicking on the *Integer* selection with the LEFT mouse button will grab it for movement or resize. Clicking on the *Integer* selection with the MIDDLE mouse button will bring up its menuform for attribute modification. Clicking on the RIGHT mouse button will bring up widget-specific information.

---

## E. Adding Selections

Go to the *Simple*, *List*, *I/O*, *Toggle*, or *Button* menus to select items of these types to add to the GUI being edited. Selecting an item from one of these menus will cause a selection of that type to be created on the working pane, at which time you may modify its size, location, or attributes. The *Workspace* and *Label* buttons can be used to create workspaces and labels. When a selection is first created with *guise*, the GUI is automatically put into edit mode, and the selection is automatically selected (control brackets will appear around it).

You may further edit any selection by going to the *Edit* menu and selecting *Edit selection*.

## F. Modifying Size & Location Of GUI Items

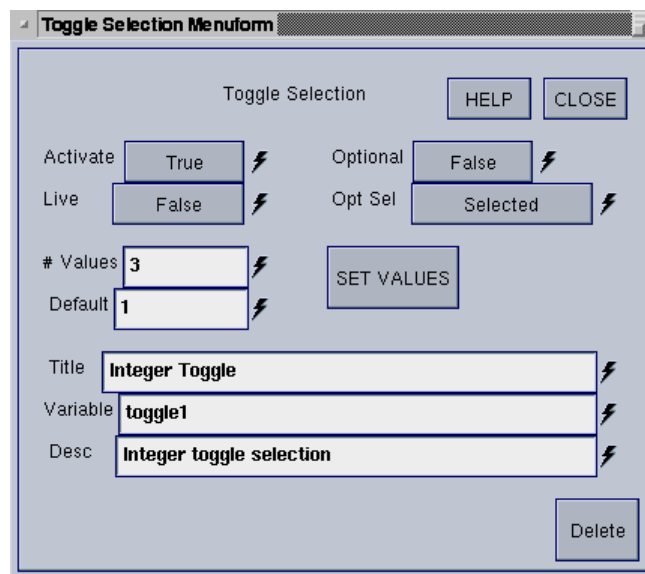
To change the size or location of an item, first make sure that the GUI is in edit mode. Then, click normally on the item of interest to select it. Now, use the LEFT mouse button to grab the selection; grabbing the item on an edge will allow you to resize the item in that direction. Grabbing the item in the middle will allow you to move the item.

## G. Modifying Other Attributes Of GUI Items (Menuforms)

To change attributes other than geometry on a GUI item, first make sure that the GUI is in edit mode. Then, click normally on the item of interest to select it. Click the MIDDLE mouse button on the item to open the *menuform* for that item. Be patient -- sometimes they take a little while to come up.

A *menuform* allows you to change all attributes that are relevant to a particular GUI item. Geometry may also be controlled by direct manipulation. Each type of menuform is specific to the GUI item which brought it up. You may not use the menuform of a Logical selection, for example, to set the attributes of an InputFile selection. The same menuform may be used to set attributes of two GUI items of the same type. For example, a menuform that is brought up for one OutputFile selection can also be used to control the attributes of a second OutputFile selection; simply click the MIDDLE mouse button on the second OutputFile selection in order to initialize the attributes reflected in the menuform from those of the first OutputFile selection to those of the second OutputFile selection.

For explanations of each of the GUI attributes, there is a *HELP* button on each menuform. For more complete details, please see Appendix A of the Toolbox Programmer's Manual. Specifically, the section which details "Most Common GUI Fields" is helpful; in this context, a *GUI Field* is the same as a GUI attribute.



**Figure 3:** The grid shows that the GUI containing this flag selection is in "edit mode," and the control brackets around the flag selection indicate that it is currently "selected." The MIDDLE mouse button was clicked on the flag selection to bring up its *menuform*, displayed here. Using the flag selection's menuform, any attribute of the flag selection can be changed: whether or not it is active, whether or not it is "live," its default, its location, its title, its variable name, and its description. An on-line description of the

flag selection can be obtained by pressing the menuform's *Help* button. The flag selection may be deleted using the *Delete* button.

---

## H. Widget-specific Information

To get specific information about a particular widget in the GUI, meta-click on the backplane. As stated above, this will put the GUI in "edit mode." Then, click on the widget to "select" it. Now, click the RIGHT button on the widget. An information object will pop up, giving layout and other relevant information about the widget. This information is geared more towards the VisiQuest infrastructure developer than the VisiQuest toolbox developer; however, the capability is left accessible for those who may find it useful.

## I. The File Menu

### I.1. Starting over ('New')

If you wish to start from scratch, you can either run `guise` with no arguments, or select *New* from the *File* menu. This will load a blank GUI template. You will be prompted to save previous changes to the GUI you are currently editing before the blank template is loaded.

### I.2. Displaying A New UIS File ('Open')

at any time you may display a new input UIS file by selecting *Open* from the *File* menu. This will display the *Open* subform. Fill in the blank in this selection in the `guise` main form and press <Return>, or you may use the file browser button to the left of the blank field to open the file browser and select a file from the directory lists.

### I.3. Reloading The GUI ('Reload')

Occasionally, one needs to reload the UIS file to redisplay the GUI prototype. To do this, select *Reload* from the *Edit* menu. The *Reload* selection displays the UIS file specified by the "Input UIS File." Furthermore, it will only reload the GUI prototype as of the last SAVE. Thus, *Reload* can be used for discarding unwanted changes. In action, *Reload* prompts the user to save changes, discard them, or cancel. When changes are discarded, the current window disappears and the last window with saved changes displays in its place.

## I.4. Saving Changes ('Save' and 'Save As')

To save an existing uis file which has already been specified, go to the *File* menu and select *Save*.

Note: If the SAVE button reads, "SAVE (Needed)", all the most recent changes to the previous prototype will be lost unless you click the OK button when prompted if you want to save changes before the new UIS file is displayed. The output UIS file parameter will be updated to match the new input file, so if changes are not to be made to the input UIS file, you must change the output filename.

You may also Save by selecting *Save As* from the *File* menu and pressing <Return> in the text box of the output file selection.

## I.5. Saving changes to a new file ('Save As')

Also from the *Save As* subform, you may specify a different filename by deleting the file name already in the field and typing in a new one. Press <Return> to save the file to the new name.

## J. The Edit Menu

The Edit menu allows access to three convenience options, *copy*, *edit*, and *delete*. To use any of these options, first select one or more GUI item(s) by clicking the LEFT mouse button on the item(s) when the backplane is in edit mode. The item(s) will be bracketed with control markers. Any convenience option invoked from the Edit pulldown menu will take effect on the selected items; if no selections are selected, an error message is then displayed.

Select 'Copy' from the Edit menu to duplicate the selected item. This selection will not duplicate an item to a different pane or form.

Select 'Edit' from the Edit pulldown menu to display the menuform for the selected item. This has the same effect as clicking the MIDDLE mouse button on the item.

Select 'Delete' from the Edit pulldown menu to delete the selected item.

### J.1. The Options Subform

The Options subform is displayed by selecting Options... from the Edit menu.

In the Options subform are two toggle selections; one for "Force Over-Write" and one for "Move/Resize Multiple Panes Together". These selections may be changed as needed by clicking the mouse button on the desired value for each toggle.

## **K. The Organize Menu**

### **K.1. Creation of Submenus and Groups ('Organize')**

One function of the *Organize* menu is to allow you to create submenus and groups from **currently selected** GUI items. To create a submenu or a group, first select two or more GUI items by clicking the LEFT mouse button on the selection(s), holding the meta key down, when the backplane is in edit mode. Note that if you do not hold the meta key down, only the latest selection will be selected. Continue the selection process until all the GUI items to be combined into a group or a submenu are bracketed with control markers. Then, select *Create ME Group (required)*, *Create ME Group (optional)*, *Create MI Group*, *Create Loose Group*, or *Sub-menu* to specify the type of grouping desired.

#### **K.1.1. Create ME Group (required)**

A required mutually exclusive group will require the user to specify *exactly one* of the selections in the group. Note that buttons may not be part of a group. All members of the mutually exclusive group must be optional, so each selected GUI item will automatically be made optional if it is not optional when the grouping process takes place. One of the selections will automatically become the default of the group (its optional button will be highlighted). To change which of the selections is the default of the group, take the backplane out of edit mode, and click on the optional box of the desired default selection.

#### **K.1.2. Create ME Group (optional)**

An optional mutually exclusive group works the same as a required mutually exclusive group (see above), except that it does not require the user to select any of the selections in the group. In other words, an optional mutually exclusive group requires the user to specify *one or none* of the selections in the group.

#### **K.1.3. Create MI Group**

A mutually inclusive group requires the user to specify *all or none* of the selections in the group. The selections in the mutually exclusive group are made optional, and all are selected by default. If you want *none* of the selections to be selected by default, take the backplane out of edit mode, and click on the optional box of one of the selections in the group; all the optional boxes will then be un-highlighted to indicate that none are selected by default.

#### **K.1.4. Create Loose Group**

A loose group requires the user to specify *at least one* of the selections in the group. The selections in the loose group are made optional, and one is selected by default. To set those items in the group as defaults, take the backplane out of edit mode, and click on the optional box(es) of the selections in the group until the desired items are selected by default. Note that an element cannot be a member of more than one group.

## K.2. Submenus

Submenus may only be created with labels, action buttons, quit buttons, help buttons, and routine buttons. If any other type of item is selected when *Submenu* is chosen from the Organize menu, an error will occur. When the submenu is created, all the selected buttons and/or labels will be collapsed into a submenu; the buttons and/or labels will disappear, and a single menubutton labeled *SUBMENU* will appear. Take the backplane out of edit mode and click on the menubutton; the previously selected GUI items will now appear in the pulldown menu. Bring up the menuform for the menubutton in order to change its label, variable, and so on.

To Delete a submenu once it has been created, display the menuform for the submenu button and click on *Delete*. You will be prompted to delete either the submenu button *and* its contents or only the menubutton itself. If you indicate that only the submenu button is to be deleted, the buttons and/or labels that were previously incorporated into the submenu will be re-mapped onto the backplane in their original positions (note that some may be obscured by other GUI items on the form if you have made changes in geometry since the submenu creation). If you indicate that the contents of the submenu are also to be deleted, all the buttons and/or labels that were incorporated into the submenu will be deleted along with the submenu button.

## K.3. Deleting Groups

Guise is not directly capable of dissolving a grouping once it is created. If you need to dissolve a group, use the *EDIT MANUALLY* button to edit the UIS file by hand. Delete the UIS line corresponding to the grouping you wish to dissolve as well as the [-E] line that follows the UIS lines corresponding to the members in the group. For example, a \*.pane file defining a pane with a required mutually exclusive group of two flag selections might look like this:

```
-F 4.3 1 0 30x30+10+20 +0+0 ' ' form
-M 1 1 30x5+0+0 +0+0 ' ' subform
-P 1 0 50x15+0+0 +0+0 ' ' pane
-H 1 6x1+38+0 'Help' 'online help page' $DEVEL/help/Empty.doc help
-Q 1 0 6x1+45+0 'Quit'
-C 1
    -t 1 0 1 0 0 5.75x1+1+1 'Flag' 'flag' flag1
    -t 1 0 1 1 0 5.75x1+1+2 'Flag' 'flag' flag2
-E
-E
-E
-E
```

To dissolve the group, delete the [-C] line corresponding to the ME group, and the [-E] line that matches it, so that the UIS file reads as follows:

```
-F 4.3 1 0 30x30+10+20 +0+0 ' ' form
-M 1 1 30x5+0+0 +0+0 ' ' subform
```

```

-P 1 0 50x15+0+0 +0+0 ' ' pane
-H 1 6x1+38+0 'Help' 'online help page' $DEVEL/help/Empty.doc help
-Q 1 0 6x1+45+0 'Quit'
-t 1 0 1 0 0 5.75x1+1+1 'Flag' 'flag' flag1
-t 1 0 1 1 0 5.75x1+1+2 'Flag' 'flag' flag2
-E
-E
-E

```

Similarly, [-B] lines defining mutually inclusive groups, and [-K] lines defining loose groups can also be deleted, along with their matching [-E] lines, to dissolve mutually inclusive and loose groups.

## K.4. Multiple Panes on Subforms ('New Pane')

When creating or modifying \*.form files (for xv routines only), you may create a new pane on a subform by selecting "New Pane" from the Organize menu.

If your current subform has only a single pane, a guidepane will be created on the left side of the subform. The guidepane will contain two guide buttons, one to access the original pane, and another to access the new pane. The second guide button will be automatically selected, displaying the new (blank) pane to the right. Select the first guide button to re-display the original pane. Note that you have to turn off edit mode to select the other pane. Also, if you decide you don't want the new pane, you can select it with the left mouse button and click Delete from the Options menu.

If your current subform already has more than one pane, a new guide button will be created on the bottom left side of the guide pane. The new guide button is automatically selected, displaying the new (blank) pane to the right. Select the other guide button(s) to re-display the other pane(s).

## K.5. GUIs with Multiple Subforms ('New Subform')

When creating or modifying \*.form files (for xv routines only), you may create a GUI that has multiple subforms accessible from a master form. by selecting *New Subform* from the Organize menu. This will create a new subform.

If your current GUI does not have a master form, a master form will be created (it will have a separate toplevel from the subform). The master form will contain a subform button with which to access the subform. The original subform will remain mapped, and the subform newly-created button will be depressed, indicating that the subform is displayed.

If your current GUI already has a master form, selecting the *New Subform* will create a new subform button on the master form, and a new subform that corresponds to the new subform button. The second subform button will be automatically selected, displaying the new (blank) subform which you will be prompted to place. Guise does not allow you to have multiple subforms displayed at the same time in order to simplify the process of deciding where a newly-created GUI item should go. Select the other subform button(s) to re-display the other subform(s). To delete the master form/subform, select the subform button while in edit mode and click Delete from the Options menu.

## L. Editing The UIS File Manually ('Edit File Manually')

The current version of `guise` is still limited with respect to editing power. However, if you have a basic knowledge of the UIS, you can use the *Edit File Manually* menu selection, located in the *Edit* menu, to spawn an editing session with a visual editor (such as "vi") in a dedicated xterm window to make changes to the GUI file directly that are not supported by `guise`.

The visual editor that is invoked, as well as the geometry of the xterm in which it appears, depends on the `VisiQuest_EDITOR` environment variable. You may dictate which visual editor is to be used by setting this variable. Example settings are shown below.

```
% setenv VisiQuest_EDITOR "xterm -geometry %g -title '%f' -e vi %f"
% setenv VisiQuest_EDITOR "emacs -geometry %g -xrm '*title: %f' %f"
```

Make the desired changes to the UIS in the xterm; as soon as you are done, save and quit the editing session. When you quit the editing session, the xterm will automatically go away. **Important Note:** the UIS file that you edit in the xterm is *not* the final UIS file whose name appears in the *Output UIS File* selection of `guise`. When you use the "EDIT MANUALLY" feature of `guise`, you are editing a temporary file in the location specified by the `TMPDIR` environment variable. As such, it is important that you use the *Save* button of `guise` to save changes made with the visual editor, or they will be lost.

When a visual editor session is in progress, writing the UIS file in the xterm will cause the GUI prototype to be re-mapped to reflect your change. Similarly, when you use `guise` to add new selections to the GUI while the xterm is up, the xterm will be re-mapped so that your session with the visual editor is kept up to date. To prevent the mapping and un-mapping of the xterm from becoming an annoyance, it is recommended that you only leave the xterm up as long as necessary to edit the UIS file.

## M. Specifying GUI Item Destination ('Create Selection On:')

When your GUI has a guidepane or a master form, there can be a question as to the destination of a newly-created GUI item. For example, suppose your GUI has a master form that displays a subform with a guidepane. Now suppose you elect to create a *Help* button. Where does the help button go? On the master form? On the guidepane of the displayed subform? On the displayed pane of the subform? To answer this question, set the value of this toggle which dictates the target location of a newly created GUI selection, located below the main `guise` menubar. When set to *Pane*, the new item will be created on the displayed pane. When set to *Guide-Pane*, the new item will be created on the guidepane of the displayed subform, and when set to *Master*, the new item will be created on the master form.

### M.1. Simple Variables

Using the "Simple" menu, you can create Integer, Float, Double, String, Flag, Logical, Workspace, and Label selections.

#### M.1.1. Integer



### **Item Description**

Integers may be part of a toggle, or members of a mutually inclusive or mutually exclusive group. They may be optional or required. Bounded or unbounded values may be specified. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

### **Use by the GUI**

When used by the GUI, the *integer selection* consists of an optional box (if the integer is optional) a title, and a text box in which the integer may be entered. Bounded integers may use an optional scroll bar. Live integers will have the stylized carriage return symbol appended to the right side.

### **Use by the CLUI**

When used by the CLUI, the integer specifies an *integer argument*, as in "-int\_variable 21". Values entered for bounded integer arguments will be checked to be sure they are within bounds; the user will be re-prompted for invalid entries. When an optional integer argument is absent from the command line, it takes on the default value specified.

## **M.1.2. Float**

### **Item Description**

A float is used when a floating point number is needed. A float may be part of a toggle, or a member of a mutually inclusive or mutually exclusive group. It may be optional or required. Bounded or unbounded values may be specified. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

### **Use by the GUI**

When used by the GUI, the float specifies a *float selection*. The float selection consists of an optional box (if the float is optional), a title, and a text box in which the float may be entered. A bounded float may use an optional scroll bar. A live float will have the stylized carriage return symbol appended to the right side.

### **Use by the CLUI**

When used by the CLUI, the float specifies a *float argument*, as in "-float\_variable 0.123". Values entered for bounded float arguments will be checked to be sure they are within bounds; the user will be re-prompted for invalid and bogus entries. When an optional float argument is absent from the command line, it takes on the default value specified.

## **M.1.3. Double**

### **Item Description**

A double is used when double precision accuracy is needed. It may be part of a toggle, or a

member of a mutually inclusive or mutually exclusive group. It may be optional or required. Bounded or unbounded values may be specified. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### **Use by the GUI**

When used by the GUI, the double specifies a *double selection*. The double selection consists of an optional box (if the double is optional), a title, and a text box in which the double may be entered. A bounded double may use an optional scroll bar. A live double will have the stylized carriage return symbol appended to the right side.

#### **Use by the CLUI**

When used by the CLUI, the double specifies a *double argument*, as in "-dbl\_variable 0.987654321". Values entered for a bounded double argument will be checked to be sure they are within bounds; the user will be re-prompted for out-of-bounds and bogus entries. When an optional double selection is absent from the command line, it takes on the default value specified.

### **M.1.4. String**

#### **Item Description**

Strings may be part of a toggle, or members of a mutually inclusive or mutually exclusive group. They may be optional or required. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### **Use by the GUI**

When used by the GUI, the *string selection* consists of an optional box (if the string is optional), a title, and a text box in which the string may be entered. Live strings will have the stylized carriage return symbol appended to the right side.

#### **Use by the CLUI**

When used by the CLUI, the String specifies a *string argument*, as in "-string\_variable 'my special string'". When an optional string selection is absent from the command line, it takes on the default value specified (the default value may be NULL). Any printable string is considered to be valid input.

### **M.1.5. Flag**

#### **Item Description**

A flag allows the input of an implied boolean value. A flag may be part of a toggle, or a member of a mutually inclusive or mutually exclusive group. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

**Use by the GUI**

When used by the GUI, the flag specifies a *flag selection*. The flag selection consists of an optional box and a title. When the optional box is highlighted, the value of the flag is considered to be TRUE (1). When the optional box is un-highlighted, the value of the flag is considered to be FALSE (0). A live flag selection has the stylized carriage return symbol appended to the right side.

**Use by the CLUI**

When used by the CLUI, the flag specifies a *flag argument*. If the flag is provided on the command line, as in "-flag\_variable," the argument is considered to be TRUE (1). If the flag is absent from the command line, the argument is considered to be FALSE (0).

## M.1.6. Logical

**Item Description**

A logical allows the input of an explicit boolean value. Logicals may be part of a toggle, or members of a mutually inclusive or mutually exclusive group. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

**Use by the GUI**

When used by the GUI, the *logical selection* consists of an optional box (if the selection is optional), a title, and a button which may be switched back and forth between the two possible values. Live logical selections have the stylized carriage return symbol appended to the right side.

**Use by the CLUI**

When used by the CLUI, the Logical specifies a *logical argument*. The logical value is provided on the command line explicitly, as in "-logical 0" for a value of FALSE, or "-logical 1" for a value of TRUE. When an optional logical argument is absent from the command line, it takes on the default value specified. The user will be re-prompted for invalid entries.

## M.1.7. Arrays

**Item Description**

Arrays may be of type integer, float, or double. Arrays may be part of a toggle, or members of a mutually inclusive or mutually exclusive group. They may be optional or required. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

**Use by the GUI**

When used by the GUI, the *integer array selection* consists of an optional box (if the array is optional) a title, and a text box in which the integer array values may be entered.

## Use by the CLUI

When used by the CLUI, the integer array specifies an *integer array argument*, as in "-int\_array '21 32 56 67; 99 1 90 4'". Integer array values on the command line must be delimited by tic marks or quotation marks.

Arrays are input as space-separated lists of numbers, with a semi-colon delineating the beginning of a new array row. For example, a 6-element 1D array might be specified as:

```
-n "23 5 7 9 0 1"
```

And a 3x2 array might be specified as:

```
-n "4 2 9; 5 4 8"
```

Array input can be performed using a short notation that generates an array of numbers. Delimitation between numbers is made using a colon, with the first number being the starting number, the second number being the ending number or a numerical increment, and the third number being the ending number if an increment is provided.

For example,

```
0:9:1
```

would be translated as:

```
0 1 2 3 4 5 6 7 8 9
```

When even increments and complete rows are specified, interpretation is definitive. However, it is possible to specify non-even increments and incomplete rows, specifications fall into three categories:

- O. Specifications give even increment, complete rows.
- A. If an increment does not land on the end value, stop at the last even increment before the end. For example, 1:11:2 would parse the same as 1:12:2
- B. If a row is incomplete, we will use 0's to pad it to the end

The following are examples of each case:

```
12 20 13;          (Case O, 2D 3x2 array)
 2   5   8

15 36 4 9          (Case O, 1D 4x1 array)

4 6 9 12; 15 23 0 0 (Case O, 2D 4x2 array)

1:10:2            (Case A, 1D 5x1 array)

1:11:2            (Case O, 1D 6x1 array)

8 9 10 12         (Case O, 1D 8x1 array)
1 2  3  4

2 3 5 9; 1 3      (Case B in 2nd row, 2D 4x2 array)

100:1000:100;    (Case O, 2D 11x2)
500:5000:500
```

5:50:1; 10:100:20	(Case A + Case B in 2nd row, 2D 46x2 array)
20:100:10; 1:100:1	(Case B in 1st row, 2D 100x2 array)
1 5; 23 56 80 1	(Case B in 1st row, 2D 4x2 array)
1:8:2; 2 4 6 8 10	(Case A+Case B in 1st row, 2D 5x2 mixed spec array)
1:11:2 5 8	(Case O, 1D 8x1 mixed specification array)
2 3 5 10:15:1	(Case O, 1D 9x1 mixed specification array)

Of course, multiple lines of array data cannot be used on the command line; they can be entered into a VisiQuest pane, however.

## M.2. Workspace

### Item Description

The Workspace provides a general purpose manager widget on the GUI which may be used as a backplane for display of images, graphics, or special-purpose GUI elements that are not provided directly. The workspace is used exclusively in the \*.form files of *xvroutines*.

### Use by the GUI

The application may use the workspace for whatever purpose may be applicable. It has absolutely no functionality on its own; the application is provided with the address of the requested widget, and is then responsible for the use and maintenance of this widget. Once the workspace widget is displayed, the GUI does not manage or interfere with it further.

### Use by the CLUI

The line in the UIS file representing the workspace is ignored by the CLUI.

## M.3. Label

### Item Description

The Blank (label) provides a non-operational label widget on the GUI which may be used for general information and text display purposes. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

### Use by the GUI

The blank selection is used to display a label on the GUI

### Use by the CLUI

The line in the UIS file representing the blank selection is ignored by the CLUI.

## M.4. List Variables

Using the "List" menu, you can create Cycle, List, and StringList selections.

### M.4.1. Cycle

#### Item Description

A cycle allows the input of one of a number of predefined choices. Each choice is represented by both a number and a label, where the numbers are incremental integers beginning with an integer that you can specify. For example, you might have a cycle that moved through the sequence, "2 (dot), 3 (dash), 4 (dot-dash)". A Cycle may NOT be part of a toggle, but it may be a member of a mutually inclusive or mutually exclusive group. It may be optional or required. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### Use by the GUI

When used by the GUI, the Cycle specifies a *cycle selection*. The cycle selection consists of an optional box (if the selection is optional), a title, and a button which may be cycled through the set of possible values. Live cycle selections have the stylized carriage return symbol appended to the right side. Because of their presentation, cycles are only recommended in situations where there are a small number (3 - 6) choices.

#### Use by the CLUI

When used by the CLUI, the Cycle specifies a *cycle argument*. The cycle value is provided on the command line explicitly, where either the integer or the label of a choice may be specified. In the above example, if the user wanted "dot", they might specify either "-cycle\_variable 2" or "-cycle\_variable dot". When an optional cycle argument is absent from the command line, it takes on the default value specified. The user will be re-prompted for invalid entries. Both the integer value and the label of the cycle value specified will be made available to the program.

### M.4.2. List

#### Item Description

A list allows the input of one of a number of predefined choices. Each choice is represented by both a number and a label, where the numbers are incremental integers beginning with an integer that may be specified. For example, you might have a list that offered the choices, "red (1),

orange (2), yellow (3), green (4), blue(5) indigo (6)". Lists may be NOT be part of a toggle, but they may be members of a mutually inclusive or mutually exclusive group. They may be optional or required. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### **Use by the GUI**

When used by the GUI, the *list selection* consists of an optional box (if the selection is optional), a title, and a button which displays a pulldown menu containing the possible values of the list. Live list selections have the stylized carriage return symbol appended to the right side. Because of their presentation, lists are recommended in situations where there are a large number (more than 5) choices. The *list selection* differs from the *displayed list selection* in that the *list selection* features a pulldown menu that is accessible via a button, while the *displayed list selection* features a list the contents of which are always displayed.

#### **Use by the CLUI**

When used by the CLUI, the list specifies a *list argument*. The list value is provided on the command line explicitly, where either the integer or the label of a choice may be specified. In the above example, if the user wanted "red", they might specify either "-list\_variable 1" or "-list\_variable red". When an optional list argument is absent from the command line, it takes on the default value specified. The user will be re-prompted for invalid entries. Both the integer value and the label of the list value specified will be made available to the program.

### **M.4.3. DisplayList**

#### **Item Description**

A list allows the input of one of a number of predefined choices. Each choice is represented by both a number and a label, where the numbers are incremental integers beginning with an integer that you can specify. For example, you might have a list that offered the choices, "red (1), orange (2), yellow (3), green (4), blue(5) indigo (6)". Lists may be NOT be part of a toggle, but they may be members of a mutually inclusive or mutually exclusive group. They may be optional or required. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### **Use by the GUI**

When used by the GUI, the *displayed list selection* consists of an optional box (if the selection is optional), a title, and a list of items from which the user may select. The list may have a scrollbar depending on the geometry of the selection. Live displayed list selections have the stylized carriage return symbol appended to the right of the label. Because of their presentation, displayed lists are recommended in situations where there is a large number (more than 5) choices. Depending on context, the actual size of the displayed list selection might be limited and the scrollbar used rather than allowing the displayed list to grow large enough to display all entries. The *displayed list selection* features a list the contents of which are always displayed, while the *list selection* features a pulldown menu which is accessible via a button.

### **Use by the CLUI**

When used by the CLUI, the displayed list specifies a *list argument*. The list value is provided on the command line explicitly, where either the integer or the label of a choice may be specified. In the above example, if the user wanted "red", they might specify either "-list\_variable 1" or "-list\_variable red". When an optional list argument is absent from the command line, it takes on the default value specified. The user will be re-prompted for invalid entries. Both the integer value and the label of the list value specified will be made available to the program.

## **M.4.4. StringList**

### **Item Description**

A stringlist allows the selection of predefined strings, or typing of a new string if the desired string is not in the predefined list. For example, you might have a "color" parameter where you know that the basic ROYGBIV colors are used most often, but you don't want to rule out the option of a less frequently used color. Stringlists may be NOT be part of a toggle, but they may be members of a mutually inclusive or mutually exclusive group. They may be optional or required. StringLists are used in the \*.form file for *xvroutines*, and in the \*.pane file for any VisiQuest program.

### **Use by the GUI**

When used by the GUI, the *stringlist selection* consists of an optional box (if the selection is optional), a title, and a text box in which a string may be entered. The title is a button that will display a pull-down menu with the predefined values of the list. Live stringlist selections have the stylized carriage return symbol on the right side.

### **Use by the CLUI**

When used by the CLUI, the StringList specifies a *stringlist argument*. The stringlist argument is treated identically to the string argument; the benefit of using a stringlist only applies to GUIs.

## **M.5. File Variables**

Using the "I/O" submenu, you can create Input File, Output File, Stdin, and Stdout selections.

### **M.5.1. InputFile**

#### **Item Description**

This allows the user to specify an input file; the file will be checked for existence and read permission. An input file may be part of a toggle, or a member of a mutually inclusive or mutually exclusive group. It may be optional or required. It may be used in the \*.form file for an



*xvroutine*, or in the \*.pane file for any VisiQuest program.

### **Use by the GUI**

When used by the GUI, the *input file* selection allows the user to enter an input file directly or to use the file browser to select a file. The input file selection consists of an optional box (if the selection is optional), a title, and a text box in which the user may enter a filename. The title of the input file selection is actually a button that can be used to bring up the file browser.

### **Use by the CLUI**

When used by the CLUI, the input file specifies an *input file argument*, as in "-i1 my\_input\_image.viff". If an invalid input file argument is entered, the user will be re-prompted. When an optional input file argument is absent from the command line, it takes on the default value specified (the default value may be NULL).

## **M.5.2. OutputFile**

### **Item Description**

The OutputFile allows the user to specify an output file that will be checked for write permission. OutputFile lines may be part of a toggle, or members of a mutually inclusive or mutually exclusive group. They may be optional or required. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

### **Use by the GUI**

When used by the GUI, the *output file selection* allows the user to enter an output file directly or to use the file browser to select an existing output file. The output file selection consists of an optional box (if the selection is optional), a title, and a text box in which the user may enter a filename. The title of the output file selection is actually a button that can be used to bring up the file browser.

### **Use by the CLUI**

When used by the CLUI, the output file specifies an *output file argument*, as in "-o1 my\_output\_file". The user will be re-prompted for invalid output file arguments. When an optional output file argument is absent from the command line, it takes on the default value specified (the default value may be NULL).

## **M.5.3. Stdin**

### **Item Description**

VisiQuest can be used as an *integration system*. This is appropriate when there exists a number of non-VisiQuest programs upon which one wishes to enforce standardized documentation, a

consistent user interface, and accessibility from the VisiQuest visual language, VisiQuest. The procedure that is followed when one is doing such an integration is to create a *pane object* for each program that is to be integrated; a pane object provides a VisiQuest GUI for each non-VisiQuest program. The stdin and stdout selections provide a mechanism whereby non-VisiQuest programs depending on stdin and stdout may be integrated into VisiQuest.

It is important to understand that programs written under the VisiQuest development system do not depend on stdin or stdout; instead, they have a formalized command line user interface where input and output files are specified using input file and output file arguments, as in "-i image:ball" or "-o my\_image.viff".

However, if the programs to be integrated into VisiQuest depend on stdin and stdout for input and output, then some mechanism must be provided from within VisiQuest to accommodate them. The stdin and stdout selections were created for this reason.

#### **Use by the GUI**

The stdin and stdout selections are only used in the \*.pane files for *pane objects* that are created in order to integrate non-VisiQuest, stdin- and stdout-dependent programs into VisiQuest.

#### **Use by the CLUI**

The lines in the UIS file representing stdin and stdout are ignored by the CLUI.

### **M.5.4. Stdout**

#### **Item Description**

VisiQuest can be used as an *integration system*. This is appropriate when there exists a number of non-VisiQuest programs upon which one wishes to enforce standardized documentation, a consistent user interface, and accessibility from the VisiQuest visual language, VisiQuest. The procedure that is followed when one is doing such an integration is to create a *pane object* for each program that is to be integrated; a pane object provides a VisiQuest GUI for each non-VisiQuest program. The stdin and stdout selections provide a mechanism whereby non-VisiQuest programs depending on stdin and stdout may be integrated into VisiQuest.

It is important to understand that programs written under the VisiQuest development system do not depend on stdin or stdout; instead, they have a formalized command line user interface where input and output files are specified using input file and output file arguments, as in "-i image:ball" or "-o my\_image.viff".

However, if the programs to be integrated into VisiQuest depend on stdin and stdout for input and output, then some mechanism must be provided from within VisiQuest to accommodate them. The stdin and stdout selections were created for this reason.

#### **Use by the GUI**

The stdin and stdout selections are only used in the \*.pane files for *pane objects* that are created in order to integrate non-VisiQuest, stdin- and stdout-dependent programs into VisiQuest.

### Use by the CLUI

The lines in the UIS file representing stdin and stdout are ignored by the CLUI.

## M.6. Toggle

Using the "Toggle" submenu, you can create a variety of toggles, including Flag, Logical, Integer, Float, Double, String, Input File, and Output File toggles.

### Item Description

The Toggle provides a method of allowing the user to choose between a finite number of predefined choices, where all choices are of the same data type. Supported toggle types include InputFiles, OutputFiles, Integers, Logicals, Flags, Floats, and Strings. Usually, the value of the toggle as a whole is determined by the default value of the chosen toggle member. For example, if a toggle has three string members with default values of "red," "green," and "blue," the value of the toggle as a whole will be "red" when the first toggle member is selected. The exception to this is with Logical and Flag toggles, where the value of the toggle is the integer number of the toggle member; for example, if a flag toggle has 5 members and the fifth member is selected, the value of the flag toggle will be 5. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

### Use by the GUI

On the GUI, a *toggle selection* consists of an optional box (if the toggle as a whole is optional), a title, and a backplane containing its toggle members. Each toggle member consists of an optional box and a title. When the optional box of a toggle member is highlighted, then that toggle member determines the value of the toggle as a whole.

### Use by the CLUI

On the CLUI, a toggle forces the user to enter one of a set of predefined choices, either by value or by title. For example, if a toggle has three string members with default values of "red," "green," and "blue," the user will have to enter "red" or "1" for the first option, "green" or "2" for the second option, and so on. They will not be allowed to enter anything other than the predefined choices. The program will have access to both the number and the label associated with toggle value entered.

## M.7. Button

Using the "Button" submenu, you can create Action buttons, Routine buttons, Help buttons, and Quit Buttons.

### M.7.1. Action Button

**Item Description**

The purpose of an action button is to return software control to the application program; it is used exclusively in the \*.form file by *xvroutines*. In general, when a pane contains two or more non-live selections, an action button should be placed at the bottom of the pane so that the user can tell the application, "I've finished setting values of selections now, go ahead and perform (some action)." Alternatively, an application may be able to perform a particular operation without any additional information provided by the user besides the fact that the user now wants the operation performed; this is the other case in which an action button is appropriate. A mouse click on the action button causes software control to be diverted from the graphical user interface to the application's subroutine that is associated with the action button, where the name of the subroutine in question is determined by the variable associated with the action button, as well as the variable associated with the pane, guide pane, or master form on which the action button appears.

**Use by the GUI**

The user clicks on an action button to request a particular action from the application. Control is immediately returned from the GUI to the application so that the action may be performed.

**Use by the CLUI**

The line in the UIS file representing the action button is ignored by the CLUI.

**M.7.2. Routine Button****Item Description**

The Routine Button is used in the \*.pane UIS file for all VisiQuest programs so that they may be integrated into the VisiQuest visual language. Routine buttons are used exclusively in the \*.pane files of VisiQuest programs.

**Use by the GUI**

When the user clicks on this button, the specified program is immediately executed with the arguments specified by the values of all other selections on the pane.

**Use by the CLUI**

The line in the UIS file representing the routine button is ignored by the CLUI.

**M.7.3. Help Button****Item Description**

The help button is a standard device on the GUI; all programs in the VisiQuest system are required to provide online help that can be accessed directly via the user interface. The help button provides access to a help file involving no additional work by the application program; all that

is needed is the correct path to the help file. VisiQuest standards specify that help buttons always provide help pages specific to their location. Thus, on a GUI with all three levels of hierarchy, the help button located on the master form will give information about the program as a whole, the help button located on a subform will give information about the general options offered by that subform, and the help button located on a pane will give detailed information about the I/O located on that pane. One help button should be used for each pane, each subform, and the master form (if any) in the \*.form file for *xvroutines*. Every \*.pane file should include one help button to access the man page for the program.

#### **Use by the GUI**

The help button provides access to online help from the GUI.

#### **Use by the CLUI**

The UIS line representing the help button is ignored by the CLUI.

### **M.7.4. Quit Button**

#### **Item Description**

The quit button allows the program to exit. In addition, when a GUI has a master form with several subforms, quit buttons on the subforms allow the user to unmap them and thus reduce clutter on the screen. When used to exit the program, the quit button is by convention labeled "Exit;" when used to close a subform, the quit button is by convention labeled "Close."

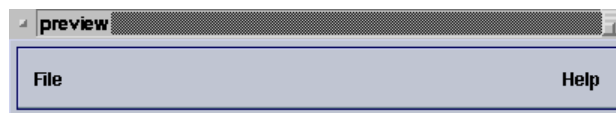
#### **Use by the GUI**

Allows the user to close a subform or exit the program.

#### **Use by the CLUI**

The UIS line specifying the Quit button is ignored by the CLUI.

## **N. Preview**



---

**Figure 4:** The preview graphical user interface display tool allows the user to input and display a UIS file. There is no capability to add new selections, but existing ones may be modified.

---

`preview` is a tool for displaying the GUI defined by a User Interface Specification (UIS) file. Its functionality

is a subset of the **guise** direct-manipulation GUI design tool; **preview** allows you to display and modify a UIS file which defines the GUI and/or Command Line User Interface (CLUI) of your program, but it has no capability for creation of new user interface items. Preview can be run independantly, or can be accessed via composer.

The preview command line options are shown below.

```
% preview
[-i]      (infile)  input UIS file
          (default = {none})
[-noform] (flag)     display GUI of specified UIS file only, not preview's GUI
[-gui]    (flag)     run from GUI as defined in *.pane file
[-V]      (flag)     gives the version number of the program
[-U]      (flag)     gives brief usage
[-usage]  (flag)     gives full usage
[-P]      (flag)     interactive prompting for arguments
[-x]      (integer)  x value for GUI autoplacement
          (-1 <= x <= 1000, default = -1)
[-y]      (integer)  y value for GUI autoplacement
          (-1 <= y <= 1000, default = -1)
[-A]      (outfile)  Creates an answer file
          (default = {none})
[-a]      (infile)  Uses an answer file
          (default = {none})
[-ap]     (infile)  prints answer file values
          (default = {none})
```

**This page left intentionally blank**

# Table of Contents

A. Overview . . . . .	4-1
B. Short Description of Menus & Contents . . . . .	4-1
B.1. The File Menu . . . . .	4-1
B.2. The Edit Menu . . . . .	4-2
B.3. The Simple Menu . . . . .	4-2
B.4. The Button Menu . . . . .	4-2
B.5. The Toggle Menu . . . . .	4-2
B.6. The List Menu . . . . .	4-2
B.7. The I/O Menu . . . . .	4-2
B.8. The Organize Menu . . . . .	4-3
B.9. The Help Menu . . . . .	4-3
C. Start Up . . . . .	4-3
D. GUI Edit Mode . . . . .	4-5
E. Adding Selections . . . . .	4-6
F. Modifying Size & Location Of GUI Items . . . . .	4-7
G. Modifying Other Attributes Of GUI Items (Menuforms) . . . . .	4-7
H. Widget-specific Information . . . . .	4-8
I. The File Menu . . . . .	4-8
I.1. Starting over ('New') . . . . .	4-8
I.2. Displaying A New UIS File ('Open') . . . . .	4-8
I.3. Reloading The GUI ('Reload') . . . . .	4-8
I.4. Saving Changes ('Save' and 'Save As') . . . . .	4-9
I.5. Saving changes to a new file ('Save As') . . . . .	4-9
J. The Edit Menu . . . . .	4-9
J.1. The Options Subform . . . . .	4-9
K. The Organize Menu . . . . .	4-10
K.1. Creation of Submenus and Groups ('Organize') . . . . .	4-10
K.1.1. Create ME Group (required) . . . . .	4-10
K.1.2. Create ME Group (optional) . . . . .	4-10
K.1.3. Create MI Group . . . . .	4-10
K.1.4. Create Loose Group . . . . .	4-10
K.2. Submenus . . . . .	4-11
K.3. Deleting Groups . . . . .	4-11
K.4. Multiple Panes on Subforms ('New Pane') . . . . .	4-12
K.5. GUIs with Multiple Subforms ('New Subform') . . . . .	4-12
L. Editing The UIS File Manually ('Edit File Manually') . . . . .	4-13
M. Specifying GUI Item Destination ('Create Selection On:') . . . . .	4-13
M.1. Simple Variables . . . . .	4-13
M.1.1. Integer . . . . .	4-13
M.1.2. Float . . . . .	4-14
M.1.3. Double . . . . .	4-14
M.1.4. String . . . . .	4-15
M.1.5. Flag . . . . .	4-15
M.1.6. Logical . . . . .	4-16
M.1.7. Arrays . . . . .	4-16
M.2. Workspace . . . . .	4-18
M.3. Label . . . . .	4-18



M.4. List Variables . . . . .	4-19
M.4.1. Cycle . . . . .	4-19
M.4.2. List . . . . .	4-19
M.4.3. DisplayList . . . . .	4-20
M.4.4. StringList . . . . .	4-21
M.5. File Variables . . . . .	4-21
M.5.1. InputFile . . . . .	4-21
M.5.2. OutputFile . . . . .	4-22
M.5.3. Stdin . . . . .	4-22
M.5.4. Stdout . . . . .	4-23
M.6. Toggle . . . . .	4-24
M.7. Button . . . . .	4-24
M.7.1. Action Button . . . . .	4-24
M.7.2. Routine Button . . . . .	4-25
M.7.3. Help Button . . . . .	4-25
M.7.4. Quit Button . . . . .	4-26
N. Preview . . . . .	4-26

**Chapter 5**

**Writing Software With VisiQuest**



# Chapter 5 - Writing Software With VisiQuest

## A. Introduction

Chapter 1 of the *Toolbox Programmer's Manual* gave an overview of the VisiQuest 2001 software development environment; Chapters 2, 3, 4, and 5 detailed the use of the programming tools used to create and modify toolboxes, software objects, and GUI's. However, there are still a number of important issues to be addressed: specifically, the purpose and use of each of the files that are automatically generated, as well as a number of general issues pertaining to the development of code using VisiQuest 2001. This chapter covers those details that were deliberately omitted from the previous chapters of the *Toolbox Programmer's Manual*.

## B. Review Of Toolbox Objects

As explained in Chapter 1, software in VisiQuest 2001 is organized into toolbox objects, each of which contains a number of software objects.

A toolbox object provides a convenient way of presenting VisiQuest developers with an encapsulated collection of information processing programs, interactive applications, and/or libraries designed for a specific application area. The toolbox object enforces a pre-defined directory structure in which its software objects are located; it manages both itself and its software objects via an object-based interface to a software database. It should be noted that some of the directories below will not be created until they are needed.

A toolbox contains the following directories:

**bin** This directory is where all the executable programs from your toolbox are located.

### **config**

This directory contains directory configuration files (Dir.\*) and machine configuration files (Site.\*). 1item "data" Data files that can be used with the programs in the toolbox are stored in this directory. Often, this directory may contain subdirectories indicating general categories of data, such as "images," "sequences," "signals," and so on. See the **sampledata** toolbox for an example of data directory layout.

### **examples**

This directory contains unsupported example programs that can be distributed. Example programs are generally used to demonstrate the proper use of public library calls for a library contained in the toolbox. Note that these programs are *not* formalized software objects created with **composer**, but simply user-created directories containing a main program (no code generation involved). They generally have no documentation other than comments inside the code. For examples of the layout and implementation of example programs, see the **envision** toolbox.

**include**

This directory contains the public include files associated with libraries in the toolbox. The include directory contains one include file named *{toolbox}.h* for the entire toolbox. It will also have one subdirectory for each library object in the toolbox, where each subdirectory is named after its library and contains the *installed* versions of the include files for the library. In other words, the include files in these directories are never modified by hand; they are simply copies of the original include files which are located in each library and maintained by the library developer. They are copied here from the originals when the library developer executes "kmake install". This set up allows the original include files to be easily packaged with the library object, but provides copies of include files at the toplevel toolbox location where they are needed in order to simplify compile rules.

**lib** This directory contains the compiled archives for each library object that exists in the toolbox.

**mach**

This directory is only used when the toolbox is supporting compiles on multiple architectures. In this case, it will contain one subdirectory for each supported architecture (solaris 2.6, irix 6.5, etc) in which symbolic links to the toolbox source code are used to build a separate compile for each architecture. See Section ? for more details.

**manual**

This directory contains the manual for the toolbox. It will contain one subdirectory for each chapter in the manual, plus a README, and directories for the glossary, index, and hardcopy.

**objects**

Software objects are located in this directory. There will be one directory for each class of software object that exists in the toolbox, named after the type; thus, there may be one or more of the standard VisiQuest *kroutine*, *xvroutine*, *script*, *library* and *pane* directories, as well as directories for user-defined software object types (if applicable). Under the classtype directories will be one subdirectory for each software object of that type, named after the software object itself.

**repos**

This directory is a repository for toolbox configuration files and the toolbox object database file.

**testsuite**

This directory is the location for any testsuites that are created in order to test the correctness of programs or libraries in the toolbox.

## C. Issues Specific to Toolbox Objects

There are a few issues that pertain specifically to toolbox objects. These issues include:

1. Steps for developing a toolbox

2. The Aliases file
3. The public toolbox include file
4. Dependencies of one toolbox on other toolboxes
5. The toolbox manual
6. Testsuites
7. Copyrights

This section discusses issues surrounding toolbox objects.

## C.1. Steps For Developing a Toolbox Object

### Create Toolbox

Use `craftsman` to create a new toolbox. See Chapter 2 in this manual for details on toolbox creation with `craftsman`.

### Create Software Objects

Use `craftsman` to create new software objects in the toolbox. See Chapter 3 in this manual for details on using `Composer`.

### Package Toolbox

Package the toolbox for distribution to other VisiQuest users. See Chapter 10 for details.

## C.2. The ToolboxInfo File

The `ToolboxInfo` file provides information about the overall purpose of the toolbox. Every toolbox has a `ToolboxInfo` file located in the `repos/config/setup` directory. The `ToolboxInfo` file should contain a few paragraphs of plain ASCII text giving an overview of the toolbox. The explanation given in the `ToolboxInfo` file should include:

1. a brief explanation of the purpose of the toolbox
2. mention of all applications (xvroutines) provided in the toolbox
3. mention of all operators (kroutines) provided in the toolbox; if there are too many to list separately, a group summary will be sufficient

4. overview of all libraries provided in the toolbox
5. a sentence listing the other toolboxes on which this toolbox depends.

An example of a ToolboxInfo file is as follows:

The Image toolbox contains general image processing operators. The library functions that implement the algorithms for the image processing operators are found in the `kimage_proc` library. Operators include:

```
irootate - Rotate image plane by arbitrary angle about an arbitrary point
idarken - Darkens or brightens an image.
ipostscr - Convert input image object to postscript
itexture - Texture feature extraction using LAW metrics
imedian - Perform median filtering
ifiltdesign - Generate 2-dimensional frequency filter
igeowarp - Perform direct bilinear warping in the WxH plane
auss_gen - Gaussian generator for a line of data
icomposite - Compositing routine for images
iopaque - Makes an image more opaque or less opaque
icreate_alpha - Creates an alpha channel
igradient - Perform image differentiation using gradient operator
igauss_func - Generate 2D gaussian function data
idissolve - Dissolves an image.
```

These routines provide processing of multiband images (images with pixel vectors). The programs adhere to the data relationships defined by the image model, which directly maps onto the polymorphic data model. All `kimage_proc` functions are written to operate on width-height planes of the polymorphic model. If the depth, time, or elements dimensions of data object are greater than one, the operation is repeated for each width-height plane. Since the image model is a subset of the polymorphic model, Image toolbox programs are interoperable with other polymorphic model-based toolbox programs. The Image toolbox is implemented using the polymorphic data services, which is contained in the `dataserv` toolbox.

The Image toolbox requires that the `bootstrap`, `dataserv`, and `datamanip` toolboxes be installed.

### C.3. The Aliases File

If the toolbox includes sample data in the *data* directory, it is good to provide an "Aliases" file in the *repos* directory which will specify shorthand aliases for the files found in the *data* directory. For example, suppose a medical imaging toolbox provided a variety of MRI images in its *data* directory. It is be very helpful to users if the author of the toolbox would also provide an Aliases file in the toolbox, giving aliases for the different MRI images provided, such as "mri:brain1," "mri:brain2," "mri:lungs," and so on.

Note that use of the Aliases capability and the syntax of the Aliases file is covered in Chapter 1 of the VisiQuest 2001 User's Guide. In short, the Aliases file consists of lines, where each line defines an alias for a

data file. Each line contains two parts:

1. The desired alias
2. The toolbox-relative path to the data file location in the toolbox

An excerpt from the `sampledata` toolbox Aliases file is as follows:

```
image:maskedcircles $SAMPLEDATA/data/images/maskedcircles.kdf
image:fat           $SAMPLEDATA/data/images/myelin.kdf
image:tools         $SAMPLEDATA/data/images/tools.xv

masks:ball          $SAMPLEDATA/data/masks/ball.bit.xv
masks:ball_&_mask   $SAMPLEDATA/data/masks/ball.with.mask.kdf

kernel:avg2x2       $SAMPLEDATA/data/kernels/avg2x2.asc
kernel:avg3x3       $SAMPLEDATA/data/kernels/avg3x3.asc
kernel:frei_chen_w6 $SAMPLEDATA/data/kernels/frei_chen_w6.asc
kernel:frei_chen_w7 $SAMPLEDATA/data/kernels/frei_chen_w7.asc
kernel:frei_chen_w8 $SAMPLEDATA/data/kernels/frei_chen_w8.asc
kernel:frei_chen_w9 $SAMPLEDATA/data/kernels/frei_chen_w9.asc

volume:5D_head      $SAMPLEDATA/data/volumes/5dhead.kdf
volume:sagittal     $SAMPLEDATA/data/volumes/sagittal.kdf
volume:boxvol       $SAMPLEDATA/data/volumes/boxvol.kdf

sequence:bushes     $SAMPLEDATA/data/sequences/bushes.kdf
sequence:hheart     $SAMPLEDATA/data/sequences/hog_heart.kdf
sequence:baby       $SAMPLEDATA/data/sequences/marine_movie.xv

plot2d:brain_stroke $SAMPLEDATA/data/signals/stroke.c.xv
plot2d:pulse        $SAMPLEDATA/data/signals/pulse.r.xv
```

## C.4. The Public Toolbox Include File

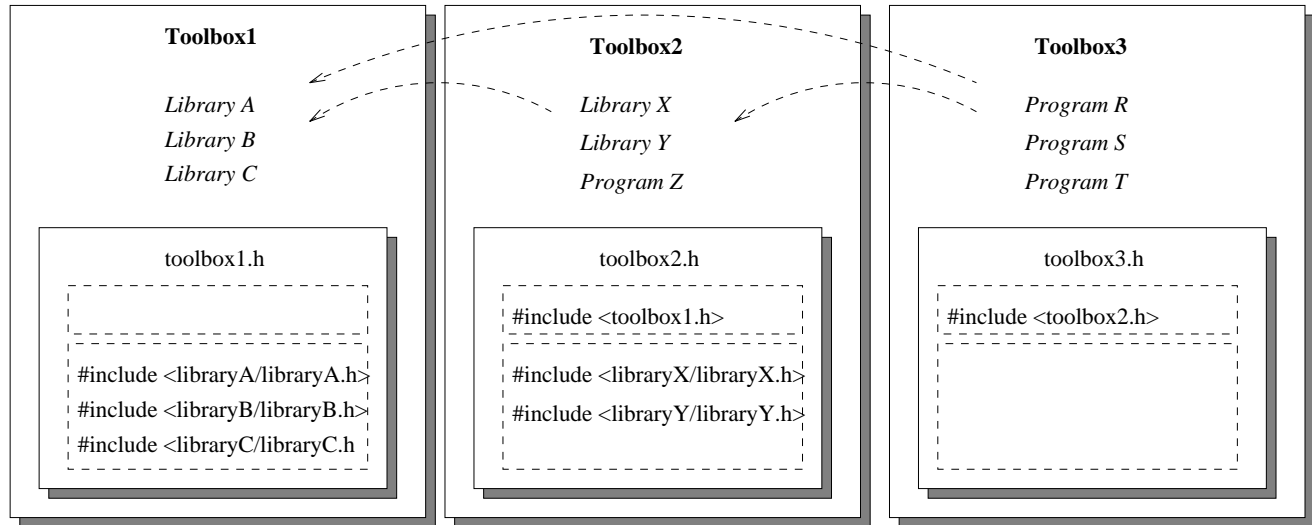
Every toolbox has a toolbox include file in its *include* directory. The toolbox include file is always named "*toolbox.h*". It serves two purposes:

1. It lists the include files of any other toolboxes on which this toolbox is dependent. Thus, the software objects in this toolbox can obtain routine prototypes, structure definitions, etc, from the other toolbox(es).
2. It lists the include files of all public libraries contained in this toolbox. This way, prototypes, structure definitions, etc, are available to other developers who may want to make their toolboxes depend on this one.



## C.4.1. How Include File Dependencies Work

For those who are not familiar with how toolbox dependencies work, this section gives an overview.



**Figure 1:** In this figure, Toolbox1 is not dependent on any other toolbox (its libraries depend only on each other). Toolbox2 is dependent on Toolbox1 (its libraries and/or programs use functions from libraries in Toolbox1), while Toolbox3 is dependent on both Toolbox1 and Toolbox2 (its programs use functions from libraries in both Toolbox1 and Toolbox2).

The figure above provides an example to illustrate toolbox dependencies and how they are handled in the toolbox include file. In the illustration, Toolbox2 is dependent on Toolbox1. This implies that Library X, Library Y, and/or Program Z use public library routines from Library A, Library B, and/or Library C. Toolbox3 is dependent on both Toolbox1 and Toolbox2; ie, Program R, Program S, and/or Program T must use public routines found in libraries A, B, or C. In addition, they must also use public routines in libraries X and/or Y. Toolbox1 is not dependent on any other toolbox.

Because Toolbox1 is not dependent on any other toolboxes, "toolbox1.h" does not have #include statements for any other toolbox files. However, programs or libraries in other toolboxes may take advantage of public routines provided by Toolbox1 in libraries A, B, and C. For this reason, the "toolbox1.h" file must include the public include files of libraries A, B, and C.

Toolbox2 is dependent on Toolbox1; therefore, "toolbox2.h" must have a #include statement for Toolbox1's include file, "toolbox1.h". Toolbox2 also provides two libraries that may be used by software in other toolboxes; to make them available to other toolboxes, "toolbox2.h" must also include the public include files for Library X and Library Y.

Toolbox3 is dependent on both Toolbox1 and Toolbox2; "toolbox3.h" only has a #include statement for "toolbox2.h", since "toolbox2.h" already includes "toolbox1.h". Toolbox3 does not offer any libraries for use by other toolboxes; it therefore has no include statements for library public include files.

## C.4.2. Syntax and Contents of the Public Toolbox Include File

The upper part of the *toolbox.h* file is maintained by *craftsman*. There are four sections that *craftsman* updates.

### Other toolbox includes

This section is for the first purpose listed above. You can make your toolbox depend on another toolbox by using *Craftsman* (see the following section). When you do this, one of the things that *craftsman* does is update your *toolbox.h* file. The following is an excerpt from the *datamanip.h* file:

```
/* other toolbox includes go here */
#include <dataserv.h>
```

In this way, the *datamanip* toolbox has access to definitions made in the include files of the *dataserv* toolbox.

### C library includes

This section is for the second purpose listed above. It lists the public include file for each public C library in the toolbox. The following is an excerpt from the *bootstrap.h* file:

```
/*-- includes for C based VisiQuest libraries --*/
#if defined(INCLUDE_CLIBS)
#include <klibc/klibc.h>
#include <kutils/kutils.h>
#include <klibm/klibm.h>
#include <kcm/kcm.h>
#include <kexpr/kexpr.h>
#include <kforms/kforms.h>
#include <kclui/kclui.h>
#include <ktestutils/ktestutils.h>
#endif
```

In this way, the *bootstrap* library makes the different Foundation Program Services libraries available to all other toolboxes in the VisiQuest system.

### X library includes

Like C library includes, X library includes also make libraries in this toolbox available for use by other toolboxes. However, the libraries here are only those that are dependent on X Windows. The following is an excerpt from the *design.h* file:

```
/*-- includes for X based VisiQuest libraries --*/
#if defined(INCLUDE_XLIBS)
#include <xvwidgets/xvwidgets.h>
#include <kwidgets/kwidgets.h>
#include <xvobjects/xvobjects.h>
#include <xvutils/xvutils.h>
#include <xvforms/xvforms.h>
#endif
```

In this way, the *design* library makes GUI Visualization Services libraries available to other toolboxes in VisiQuest.

### C++ library includes

These are similar to C and X library includes, except that they list C++ libraries. The following is

an excerpt from a C++ toolbox:

```
#if defined(INCLUDE_CXXLIBS)
#include <camlib/camlib.h>
#endif
```

*Important Note: in each of the sections above, the "#if defined" and "#endif" lines are keys for automatically update of the toolbox.h file. Do not delete them or change them for any reason!*

While the upper part of the *toolbox.h* file consists of the automatically generated sections described above, the lower part of the *toolbox.h* file is for the developer's use. It looks like this:

```
/*-----*
|         #defines
|-----*/

/*-----*
|         typedefs
|-----*/

/*-----*
|         global variable declarations
|-----*/

/*-----*
|         macros
|-----*/

/*-----*
|         routine definitions
|-----*/
```

In these sections, you may add definitions, typedefs, global variable declarations, macros, or routine definitions (prototypes) as desired. These definitions will be available to all software objects in the toolbox.

## C.5. Toolbox Dependencies

When writing new software objects, you may want to use library functions that are provided by other toolboxes in VisiQuest. In order to gain access to the libraries of another toolbox, you must make your toolbox *depend* on the other toolbox. By default, all new toolboxes depend on the **design** and the **dataserv** toolboxes (because the **design** toolbox depends on the **bootstrap** toolbox, it is implied that all new toolboxes also depend on **bootstrap**).

Craftsman is used to set toolbox dependencies. Select "Toolbox Attributes" from the Toolbox menu of **craftsman**, and then use the "Select Attribute Type" to select the "Dependencies" pane. Use the "Dependencies" pane to specify the other toolbox(es) on which you want your toolbox to depend.

The remainder of this section provides an explanation of how toolbox dependencies are implemented. Those who are not interested in this level of detail may skip to the next section.

When toolbox dependencies are set, **craftsman** makes four changes to your toolbox:

1. The toolbox dependency is specified in your toolbox.mk file
2. Your toolbox's public include file is updated with a line including the other toolbox's public include file
3. All .depend files in your toolbox's objects/ directory are regenerated

First, the *toolbox/repos/config/mk/toolbox.mk* file is updated to list the new toolbox dependencies. **Craftsman** does this by adding a single line that includes the toolbox.mk file of the other toolbox, in the "Include dependent toolbox.mk rules" section. Having this reference in your toolbox.mk file will allow software objects in your toolbox to correctly compile and link against the other toolbox by causing the correct "-I" segments to appear in the compile line and accurate "-L" segments to appear in the link line. The following excerpt is from the toolbox.mk file of a toolbox that has been set to depend on the **datamanip** toolbox:

```
#####  
#   Include dependent toolbox.mk rules:  
#####  
# -toolbox_include_toolboxes  
  .include <${DESIGN_CONFIG}/toolbox.mk>  
  .include <${DATAMANIP_CONFIG}/toolbox.mk>  
# -toolbox_include_toolboxes_end
```

Next, your toolbox's public include file, *toolbox/include/toolbox.h*, is modified with a `#include` statement so that it includes the public include file of the other toolbox. Referencing the public include of the other toolbox allows software objects in your toolbox to obtain structure definitions, `#defines`, and library prototypes from the other toolbox. The following excerpt is from the public include file of a toolbox that depends on the **datamanip** toolbox:

```
/*-----*  
|           #include  
-----*/  
/* other toolbox includes go here */  
#include <design.h>  
#include <datamanip.h>
```

Then, the `KCMU_TOOLBOX_DEPENDENCIES` attribute of your toolbox is updated. This is necessary so that **craftsman** can keep track of your toolbox dependencies.

The last step is only performed if your toolbox already contains one or more software objects when its toolbox dependencies are set. If it does, all .depend files in the toolbox are re-generated according to the updated *toolbox.mk* file. These .depend files list each of the include files that are needed by the different source code file of your software object; they make sure that your software objects will be forced to recompile if any of those include files are changed.

**IMPORTANT NOTE:** The toolbox.mk file and .depend files are updated automatically by `craftsman`. They should never be edited by hand.

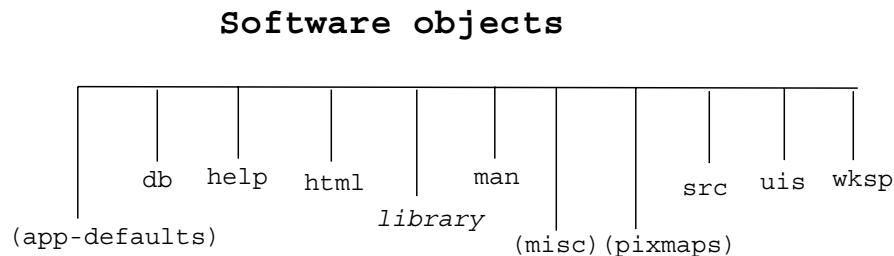
## C.6. Copyrights

Use `craftsman` to change the copyright for your code. Set the "Select Attribute Type" menu to "Copyright" on the *Toolbox Attributes* subform. The "Copyright" pane allows you to set both a short copyright statement and a long copyright statement. The short copyright is put at the bottom of documentation files, while the long copyright is put at the top of source code files.

## D. Review Of Software Objects

You may recall from Chapter 1 that a *software object* is simply a collection of files that are managed as a unit. The files in the software object are organized in a standardized directory structure that is defined by the code generator for the software object classtype.

The flexible code generation system of VisiQuest allows new software object class types to be defined as desired by their code generators. VisiQuest supports seven standard types of software objects, including *code generator objects*, *library objects*, *kroutine objects*, *xvroutine objects*, *pane objects*, *script objects* and *workspace objects*. The different types of software objects were outlined in Chapter 1; this chapter goes into further detail on the purpose and contents of each file in each of the software objects. As with toolbox objects, some directories in the software object are not automatically created; they may be created by the developer if and when they are needed.



---

**Figure 2:** Each software object supported by the VisiQuestPro2001 system has some subset of the directories depicted above. The base set of directories contained by a software object is determined by its class type; in some cases, it may be extended by the developer with optional directories such as *misc/* and *pixmaps/*.

---

Chapter 1 of this manual details the contents of each directory in each type of software object, so that information will not be repeated here. At this point, it is assumed that you have a clear understanding of the different types of software objects and are familiar with the standard locations of files within the directory structures corresponding to software objects. It is further assumed that you are comfortable with the purpose of each type of software object and have a good idea of how to create and maintain software objects using the

craftsman, composer, and guise software development tools from the preceding chapters.

This chapter "fills in the blanks" left by previous chapters by detailing the contents of each of the files in a software object. By familiarizing yourself with the purpose and contents of each file in a software object and becoming aware of the various rules governing software objects outlined in this chapter, you ensure that you will be able to use the VisiQuest software development environment correctly and efficiently.

## **E. Common Issues**

There are a number of issues that are common to most or all types of software objects. These issues include:

1. The Graphical User Interface (GUI) of VisiQuest programs
2. The Command Line User Interface (CLUI) of VisiQuest programs
3. The User Interface Specification (UIS) file
4. Automatically Generated Code and Documentation
5. Prototyping of Subroutines & Functions
6. Software Object Database files
7. The VisiQuest Data File (KDF) Format
8. File Naming Conventions
9. Argument Naming Conventions
10. Tips for Writing Successful Software

This section discusses each of these issues. The following sections then go on to detail the specifics of each different type of software object.

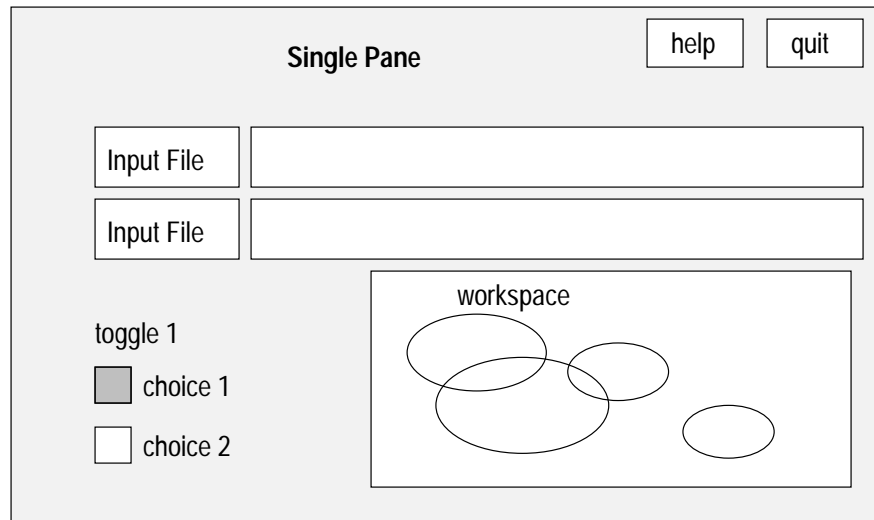
### **E.1. The Graphical User Interface (GUI)**

From the user's point-of-view, the VisiQuest GUI is organized in a three-level hierarchy:

1. A *master form* (optional),

2. One or more individual *subforms*,
3. One or more *panes* on each subform.

The use of such a hierarchical interface aids users in categorizing problems into classes and subclasses. A key technique to solving problems lies in categorization and identification of the problem so that you may then reference the relevant parameters and operations; a hierarchical user interface aids in the visualization and recall of an effective tree-structure organizational system.



---

**Figure 3:** For *xv* routines with simpler GUI's, a single pane, such as the one shown, is sufficient.

---

A pane is where most actual I/O takes place -- here, *selections* are provided for entering strings, integers, floats, doubles, boolean values, input files, output files, cycles, lists, and so on. Any such selection may be made optional, in which case a small box will appear to the left of the selection; the user highlights this box to indicate a wish to utilize the parameter described by the text selection.

For most situations in an application program, it is necessary for the user to enter several values in text selections before initiating an action. For these cases, the user may enter values in a pane's text selections at leisure; when assured that all parameters are correct, they click the mouse on an *action button* to register the parameters and perform the desired operation. If, in the application program, the situation arises that only one input (entered via a text selection) is needed in order to perform a certain action, the text selection may be made "live" -- that is, the desired action is initiated by the user entering the applicable parameter and hitting the carriage return. Such "live" selections are marked by a stylized lightning bolt pixmap that appears to the right of the selection, and in situations where they are appropriate, will reduce the amount of human I/O necessary to initiate an action by a factor of two.

Action buttons, which may be located anywhere on the GUI, allow users to perform a particular action. Specialized action buttons called *routine buttons* are used to execute a program.

For entering input files, a *file browser* option is provided for the convenience of the user who does not wish to remember or type a long path name. Feedback messages are provided in pop-up form for error checking and

general user information.

*Toggles* are also provided so that the user may set a parameter to one of a finite number of values. Toggles may be of type integer, float, string, input file, or output file. When the functionality of a toggle is desired, but elements within the proposed toggle are of different data types, *groups* are used. A *mutually exclusive group* allows the user to select one and only one of a number of selections; a *mutually inclusive group* allows the user to select all or none of a number of selections; a *loose group* forces the user to select at least one of a number of selections.

*Help buttons* provide a mechanism for the user to access on-line help appropriate to the location in the GUI on which it is found. Since a master form is at the top of the user hierarchy, help here should be of the most general kind. Help found on each subform and purpose of that particular subform, while help buttons on each subordinate pane will explicitly describe the proper use of the parameters on that pane.

*Kroutines* always have GUI's that consist of a single pane on a single subform; that is, they utilize only the lowest level of the three-part GUI hierarchy. Many *xvroutines* also have simple GUI's using single panes on single subforms.

The image shows a GUI window titled "Pane". At the top right, there are three buttons: "Run", "Help", and "Quit". Below these are several input fields: "options" (a rounded rectangle), "Input File" (a rectangle with a text box), "Integer" (a rectangle with a text box), "Float" (a rectangle with a text box), and another "Float" (a rectangle with a text box and a small grey square to its left). At the bottom, there is a "Logical" label and a rounded rectangle containing the text "true".

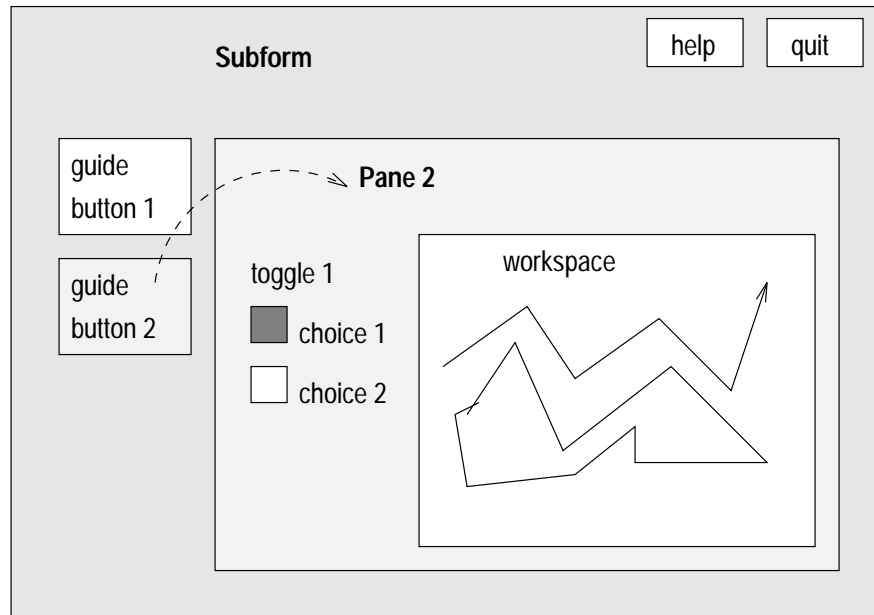
---

**Figure 4:** A GUI for a *kroutine* always consists of a single pane on a single subform.

---

With *xvroutines*, a subform may have multiple panes, only one of which may be displayed at any one time. Such subforms contain *guide buttons* with titles reflecting the pane that they will cause to display.



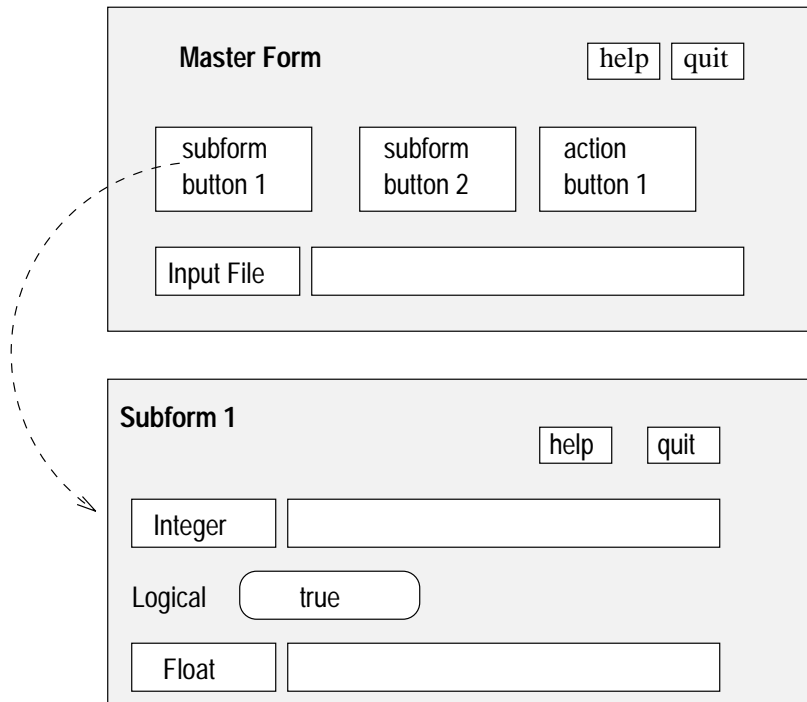



---

**Figure 5:** An *xvroutine*'s subform may have guide buttons that the user can click to display different *panes* on the subform.

---

At the highest level of the VisiQuest GUI is the *master form*. Only when an *xvroutine* has an extensive GUI requiring multiple subforms is a master form used. This master form contains *subform buttons*, each labeled with the broadest category of operation in the application. When the user clicks the mouse on the subform button labeled with the desired category, a subform will be mapped separately from the master form and from other subforms.




---

**Figure 6:** A master form has subform buttons that the user can click to display independent subforms.

---

## E.2. The Command Line User Interface (CLUI)

The Command Line User Interface (CLUI) is supported by all VisiQuest programs. It is the only GUI for those who do not have access to graphics workstation supporting X Windows. The CLUI is modeled after the basic style for most UNIX commands. The program name is typed in after the prompt, followed by appropriate *flags*. Standard flags, such as [-U], [-V], [-P], [-A], [-a], and [-gui] are automatically supported.

Program objects developed using VisiQuest need not do anything to support these standard arguments. Additional program arguments are specified as necessary.

## E.3. The User Interface Specification (UIS) file

Kroutine, xvroutine, pane, and some script software objects have at least one UIS file. The UIS file is comprised of lines made up of fields in a strictly defined syntax. Each UIS line describes either a program argument on the CLUI, an item on the GUI, or both. The UIS was designed at a high level of abstraction; each type of line contains the information necessary to completely describe that type of item in the GUI, as well as in the CLUI. Thus, from a single UIS file for a particular application, the code for both the CLUI and the GUI may be generated directly. The UIS file is used for three distinct purposes in VisiQuest:

1. It defines the CLUI of all VisiQuest programs.
2. It defines the GUI of the program that is used when the program is run using the [-gui] option; this same GUI is used to integrate the program into the VisiQuest visual language.
3. For interactive X-Windows based applications (xvroutines), it defines the interactive GUI of the application.

For *kroutines*, a single UIS file will suffice for all purposes. *Xvroutines* often use two UIS files: one to define the CLUI of the program and to integrate it into VisiQuest, and another to define the more sophisticated independent GUI of the program. For *pane objects*, the UIS file provides the alternate interface to the base program of the pane object. For *script objects*, the UIS file is used to integrate the script into VisiQuest.

## E.4. Automatically Generated Code and Documentation

The VisiQuest software development system uses code generators to automatically generate code and documentation for VisiQuest software objects. The six types of software objects supported by the VisiQuest 2001 system include:

1. kroutine objects
2. xvroutine objects
3. library objects
4. pane objects
5. script objects
6. workspace objects

There is a different code generator for each class type. The code generator for the class type determines the names, contents, and locations of the files generated for the software objects of that type. <sup>1</sup> *Composer* invokes correct code generator for the software object based on its class type when "Generate Code" is selected from its *Make* menu. The correct code generator for a software object can also be invoked manually by typing:

```
% kmake regen
```

in the *src/* directory of the software object.

---

<sup>1</sup> Rather than dictating the structure of the various types of software objects as was done in earlier versions of VisiQuest, the flexible code generation system of VisiQuest 2001 allows software object class types to be defined by the author of the code generator. Thus, the different types of software objects that can be supported is unlimited. No documentation on the writing of a code generator is available; however, VisiQuest Research, Inc. does offer services to deliver support for new software object class types.

The `kregenobj` command may also be used to invoke the code generator for a software object. The command line arguments to `kregenobj` are as follows:

**tb** The [-tb] flag specifies the name of the toolbox in which the software object exists

**oname**

The [-oname] flag specifies the name of the software object on which the code generator is to be run.

**force**

Setting this flag to TRUE will force over-write of files; that is, it will suppress prompting before a file is over-written.

The code generators create code and documentation for software objects based on the user interface specification in the \*.pane file. For kroutines, scripts, and pane objects, the code generators automatically generate the main program, code to obtain command line arguments and store the values in a structure, and documentation. For xvoutines, code is also generated to obtain user input from the GUI and pass it to the application. For library objects and workspace objects, the files generated are primarily documentation.

### E.4.1. Code generation

The code generators also register a software object within a toolbox and maintain the database file for the software object. The database file of a software object stores crucial information such as the location of files and the values of software object attributes.

In contrast to earlier versions of VisiQuest, which produced files that mixed automatically generated code with code to be edited by the developer, the VisiQuest 2001 code generators create two different types of source code files:

1. Auto-generated files
2. Files for the developer

The first type of file contains *only* automatically generated code. These files may be viewed, but they should never be edited by hand. They are created with read-only permissions and are over-written every time the code generator is executed.

The second type of file is created only once. These files contain skeleton code and/or documentation and serve as starting points for the developer. Once they are created, they are never again touched by the code genera-

tors.

## E.4.2. Documentation generation

As with earlier versions of VisiQuest, documentation source files in KP2001 mix automatically generated text with user-written text. For this reason, *tags* are still used so that the code generators can differentiate between documentation that was written by the user and documentation that is automatically generated. However, there are two important differences with documentation generation in KP2001 as opposed to earlier versions of VisiQuest:

- Earlier versions of VisiQuest used the software object man page as the source for documentation text. Documentation was added to the man page and was propagated by the code generators to the HTML file and online help file.

In contrast, KP2001 uses the HTML page as the documentation source rather than the man page. Thus, documentation is added to the HTML page and is propagated by the code generators to the man page and the online help file.

- Earlier versions of VisiQuest required the user to write documentation between tags that marked different sections of user-defined text. Thus, the rule for editing these files *was*: "Don't delete or change the tags. Always put your documentation (or code) between the tags, or it will be lost the next time you run the code generator."

KP2001 takes the opposite approach: tags are used to mark automatically generated text. So, the rule in KP2001 for editing these files is: "Don't delete or change the tags. You may edit any part of the file except the tags and the auto-generated documentation between them (to which any changes will be lost the next time you run the code generator)."

Later sections of this manual give specifics on the code and the documentation that is generated for each class type of software object.

## E.5. Prototyping of Subroutines & Functions

Because VisiQuest is ANSI-compliant, all routines and functions must be *prototyped*. Prototypes are statements that define the number and data types of arguments that are passed to a function; they are quite helpful as they will cause an error message to be produced during compile time if a calling routine calls a function with the wrong number of arguments, or with arguments having the wrong data type. All functions, whether defined in a library or in a program, must be prototyped.

A prototype is structured as follows:

```
data_type routine_name(data_type1, data_type2, ...);
```

Parsed item by item, the prototype breaks down as:

1. the return type of the function (specify `void` if no value is returned)
2. the routine name
3. the data types of each of the arguments to the routine, separated by commas

An example of a prototype for the VisiQuest program *lkarith* is:

```
int lkarith1 (kobject, char *, kobject);
```

Note: the `PROTO` macro used in earlier versions of VisiQuest will still work, but it is unnecessary.

Also note: the `mkproto` program, available in the `migration` toolbox, is very useful for generating prototypes automatically.

## E.6. Software Object Database Files

Every software object has a `db/` directory. In this directory will be a file named "obj.kcm". The VisiQuest configuration management system stores in this file all the information it needs in order to keep track of the software object. In fact, it can be said that it is this file that *makes* the software object an object; it allows the software object to be managed as a single entity by software development tools such as `craftsman`, `composer`, `VisiQuest`, and so on.

These simple access database files contain key/content strings. They are created and maintained by the software development system. Although they are ascii files, they are not designed for direct viewing or editing, and should never be edited manually.

The main function of the "obj.kcm" database file is to keep track of the locations of all the files associated with a software object; it also stores various and sundry other pieces of information about the software object. For example, it stores the location of the `UIS` files, source code, include files, and documentation. It also stores the category and subcategory under which the program may be accessed via `VisiQuest`. It stores special programming language dependencies of the software object (if any), major and minor release numbers of the software, and so on.

If necessary, to verify the keys associated with a software object and their values, the contents of the "obj.kcm" database file for a software object may be viewed in a somewhat more readable form by going into the `db` directory and executing the command:

```
% kcmcat -i obj.kcm
```

## E.7. The VisiQuest Data File (KDF) Format

The general format for a KDF file consists of a header, a series of attribute blocks, and a series of data blocks. It can be pictured roughly as follows:

KDF Header
Attribute block for object
Attribute block for segment 1
Attribute block for segment 2
....
Data for segment 1
Data for segment 2
.....

### E.7.1. KDF Header

The KDF header consists of 7 bytes followed by three integers. Note that all information contained in a KDF file will be stored in a machine specific format. The seventh byte indicates the machine storage type for the particular file.

4 bytes	2 bytes	1 byte	1 int	1 int
Magic #	Version #	Machine Type	Num Data Sets	Num Attr Blocks

The magic number and version: 01 03 19 94 00 02

The machine type follows the numbers defined in:

`$BOOTSTRAP/include/machine/kmachine.h`

The number of data sets is currently hardwired to 1. This will be used in the future to store multiple data objects per file.

There is always at least one attribute block for storing the object level attributes. Beyond that, there is one attribute block for each segment in a data object. The attribute blocks follow sequentially from the header.

### E.7.2. Attribute Blocks

Null Terminated String	1 Int
Segment Name	Num of Attributes for Segment

The segment name will simply be a \0 for the object-level attribute block. The number of attributes describes the number of user-defined attributes for a segment and may be 0.

If the segment name is not NULL, then all the physical segment characteristics will be stored next. If the segment name is NULL, then none of the following information will be present. The segment characteristics are the dimension, datatype, size, and index order. They are stored as follows :

1 Int	Null-terminated String
Dimension	Datatype

Dimension Ints for Size			
Size [0]	Size [1]	...	Size [dim-1]

Dimension Ints for Order			
Order [0]	Order [1]	...	Order [dim-1]

1 Int	1 Int
Fixed_dim	Dim_index

These last two should be hardwired to -1 for now. In the future, these fields will be used for storing multiresolution data.

Datatypes are stored as a string representation of the datatype. The routine `kdefine_to_datatype` can be used to convert from the standard VisiQuest type identifiers to strings and the routine `kdatatype_to_define` can be used to convert the other direction.

Valid datatypes are: bit, byte, unsigned byte short, unsigned short, integer, unsigned integer, long, unsigned long, float, double, complex, and double complex.

Following this will be all other attributes for this segment. There will be exactly the number of attributes as was indicated at the top of the attribute block. Each attribute will follow the format described in the next section.



```

Attribute 0
Attribute 1
...

```

**E.7.3. Attributes**

1 String	1 Int	1 Int	1 String
Attribute Name	Number of Args	Arg Size	Datatype
Attribute Data			
EOA Tag			

The attribute name is a string. The number of attribute arguments and the size of each argument follow. The datatype of the attribute is stored next. Datatypes are stored in a string. The routine `kdefine_to_datatype` can be used to convert from the standard VisiQuest type identifiers to strings and the routine `kdatatype_to_define` can be used to convert the other direction.

Valid standard datatypes are: bit, byte, unsigned byte, short, unsigned short, integer, unsigned integer, long, unsigned long, float, double, complex, double complex, and string. Any arbitrarily-defined datatype is also allowed, although it is dependent on the datatype being defined in order to be read.

The attribute data follows, stored in a large contiguous block. The attribute data is followed by an end of attribute tag.

The end of attribute tag (eoa) is a string: " $\langle \rangle$ ". This tag is only really used when reading past attributes of an unknown data type (an undefined structure datatype for instance).

## E.7.4. Segment Data

Segment 1 Data
Segment 2 Data
....

The segment data is stored following the attribute blocks in a series of blocks with all the data for that particular segment. There is one block per segment. The order of the segment data blocks must match the order of the corresponding attribute blocks. The amount of data present for each segment must be consistent with the physical segment characteristics specified in the attribute block that corresponds to this segment. For instance, a segment with the characteristics of dimension = 2, size = [32, 24], and datatype = "byte," dataset would be expecting 32x24 bytes of data to be stored for its segment data block.

## E.8. File Naming Conventions

The table below summarizes the file naming conventions used in VisiQuest.

File suffix	Description
*.a	library archive
*.asc	ascii file
*.ans	CLUI answer file, use with [-a], [-A], [-ap]
*.awk	awk script file
*.bugs	bugs log for software object
*.c	C source code
*.C	C++ source code
*.cc	C++ source code
*.classified	image previously classified using spectrum
*.cmdlog	VisiQuest command execution log file
*.csh	csh script file
*.disp_env	display environment file for spectrum
*.doc	online help pages for xroutines (NOT Word files)
*.eps	encapsulated postscript files ready to be printed
*.facet	Geometry FACET data (only understood by kimport_facet)
*.grouped	grouped image output from spectrum
*.h	C include files
*.hlp	version of man page specially formatted for online help.
*.f	fortran source code
*.form	UIS file describing GUI for xroutine
*.l	lex source code

File suffix	Description
*.lgd	legend file for spectrum
*.man	source files for manuals
*.ms	manual files containing ms macros produced by kgenmanual.
*.o	object file produced by compiler
*.pane	UIS file describing CLUI & GUI of a software object
*.pnm	Files that are in Portable Any Map (*.ppm and *.pbm) format
*.pl	perl script files
*.ps	postscript files ready to be printed
*.sec	chapters or sections of a manual
*.sh	bourne script files
*.subform	modularized UIS file describing subform for an xvroutine
*.txt	text file (not used that often)
*.wk	saved VisiQuest workspace
*.wksp	saved VisiQuest workspace
*.xbm	Files that are in X11 Bitmap format
*.xpm	Files that are in X11 Pixmap format
*.xwd	Files that are in X11 Window Dump format
*.y	yacc source
*.1	section one man page.
*.3	section three man page.

There are also some files in VisiQuest which always have the same name and the same purpose. Some of these are listed below:

File suffix	Description
Aliases	file containing VisiQuest Alias definitions
Pmakefile	file containing rules for compiling the software
ToolboxInfo	file describing toolbox
cms.db	toolbox database file containing toolbox info
VisiQuest_env	VisiQuest environment file
toolbox.mk	toolbox configuration file
obj.kcm	software object database file
objects.db	toolbox database file listing software objects
manpage.db	toolbox database file listing library functions w/ man page locations
toolbox.sh	file containing installation instructions for toolbox
whatis	automatically generated file containing library function descriptions

## E.9. Argument Naming Conventions

The following is a list of standard arguments used in the VisiQuest application programs. A hash mark (#) after the argument name indicates that a number is appended to the name. A question mark (?) before or after the argument name indicates that the name may be prefixed or appended with a character or string.

### Input Files

The following variables are used with arguments that specify input files.

*i* - single input file

*i#* - multiple input files  
*igate* - operation gating input  
*cmap* - input file containing colormap

## Output Files

The following variables are used with arguments that specify output files.

*o* - single output file  
*o#*, or *o?* - multiple output files  
*ov* - output file containing overlay  
*f* - output ascii file  
*f#* - multiple output ascii files

## Dimension Parameters

The following variables are used with arguments that carry information about data dimensions. In general, all dimension variables start with the first letter of the dimension (w,h,d,t,e).

### Dimension Flags

Dimension flags specify operation on the dimension. For an example, see *kstats*.

*w* - include width in operating unit  
*h* - include height in operating unit  
*d* - include depth in operating unit  
*t* - include time in operating unit  
*e* - include elements in operating unit  
*whole* - perform operation on entire data set

### Size Parameters

The dimension size parameters are *integer* selections.

*wsiz*e - size of width dimension  
*hsiz*e - size of height dimension  
*dsiz*e - size of depth dimension  
*tsiz*e - size of time dimension  
*esiz*e - size of elements dimension

### Offset Coordinates

The dimension offset coordinates are *integer* selections.

*woff* - offset or position in width dimension  
*hoff* - offset or position in height dimension  
*doff* - offset or position in depth dimension  
*toff* - offset or position in time dimension  
*eoff* - offset or position in elements dimension

### Center of Action Coordinates

The dimension center of action coordinates are *integer* selections.

*wc* - width coordinate for the center of action  
*hc* - height coordinate for the center of action  
*dc* - depth coordinate for the center of action  
*tc* - time coordinate for the center of action  
*ec* - elements coordinate for the center of action

### **Magnification Parameters**

*wmag* - magnification in the width dimension  
*hmag* - magnification in the height dimension  
*dmag* - magnification in the depth dimension  
*tmag* - magnification in the time dimension  
*emag* - magnification in the elements dimension

### **Components of Data Model**

The following flags are used with arguments that specify particular data model components.

*loc* - operate on location data  
*map* - operate on map data  
*mask* - operate on mask data  
*time* - operate on time data  
*val* - operate on value data  
*segment* - string variable where the data segment name can be specified (see *kimportasc*).

### **Constants**

*real* - real constant for input when complex data is supported  
*imag* - imaginary value for input when complex data is supported  
*r?* - parameter associated with real value of complex number  
*i?* - parameter associated with imaginary value of complex number  
*scale* - scale factor  
*tval* - Value assigned to output if evaluation is TRUE  
*fval* - Value assigned to output if evaluation is FALSE  
*?val* - value for input when complex data is not supported (*val* is not used alone to specify a constant, since *val* is reserved for the *value* component of the data model).  
*tol* - tolerance  
*rtol* - real tolerance when complex data is supported  
*itol* - imaginary tolerance when complex data is supported

### **Other Commonly Used Variables**

*type* - data type (a *stringlist* selection, see *kconvert*"")

- 0 Propagate Input Type
- 1 Bit
- 2 Byte
- 3 Unsigned Byte
- 4 Short
- 5 Unsigned Short
- 6 Integer
- 7 Unsigned Integer
- 8 Long
- 9 Unsigned Long
- 10 Float
- 11 Double
- 12 Complex
- 13 Double Complex

*norm* - (flag) perform normalization

*valid* - (logical) identify padded data added by program as either valid or invalid in the object's

mask  
*attr* - (flag) use objects attributes (for example, subobject position in kinset)  
*fg* - foreground  
*bg* - background  
*lw* - line width  
*lt* - line type

### Reserved For Predefined Command Line Arguments

The following variables are reserved as standard command line arguments, and *may not be used* for another purpose by any VisiQuest program

*V* - give version number  
*U* - give usage of program  
*P* - do interactive prompting  
*gui* - display GUI as defined by \*.pane file of program  
*a* - read in CLUI answer file for specification of argument values  
*ap* - prints CLUI answer file values  
*A* - write out CLUI answer file to save argument values  
*x* - X screen coordinate for automatic GUI placement for xvroutines  
*y* - Y screen coordinate for automatic GUI placement for xvroutines  
*jr* - create a journal recording of an xvroutine  
*jp* - run a journal playback of an xvroutine  
*form* - use alternate \*.form file to specify GUI of xvroutine

## E.10. Tips for Writing Successful Software

### Before Writing a Utility Function

Before writing a utility function, check VisiQuestgram Services to see if one is already provided for you. The command,

```
% kman -k {keyword(s)}
```

is very useful for searching for such utility routines.

### Use Operating System Services For All I/O

You *must* use the VisiQuest counterparts to all *libc* I/O routines in order to correctly support the various data transport mechanisms. For example, always use *ksprintf()* rather than *sprintf()*, and always use *kfread()* instead of *fread()*. You will find that there is a VisiQuest replacement for virtually every *libc* I/O routine. NEVER MIX Operating System Services routines with *libc* routines - your program will core dump!

### Free Memory After Use

Always free memory after use, especially with programs that use large amounts of memory or allocate memory from within a loop. *purify* is a good tool to help you identify memory leaks, as well as general misuse of memory, such as memory overwrites.

## NULL termination of Set/Get Attributes Routines

One of the most common bugs when using Data Services, Software Services, and GUI & Visualization Services is neglecting to NULL-terminate a get or set attribute routine call. For example,

```
xvw_set_attributes(image_object,
                  XVW_BELOW,          label_object,
                  XVW_IMAGE_IMAGEOBJ, data_object,
                  NULL);
```

is correct, since it is NULL-terminated. In contrast,

```
xvw_set_attributes(label_object,
                  XVW_LABEL, "this is the label",
                  XVW_FOREGROUND_COLOR, "yellow");
```

will cause serious problems. This is true of all the plural set- and get- attribute routines, including *kpds\_set\_attributes()*, *kpds\_get\_attributes()*, *kcolor\_set\_attributes()*, *kcolor\_get\_attributes()*, *kgeom\_get\_attributes()*, *kgeom\_set\_attributes()*, *xvw\_set\_attributes()*, *xvw\_get\_attributes()*, *xvf\_set\_attributes()*, *xvf\_get\_attributes()*, etc.

## Use Correct Data Type with Variable Argument Functions

Whenever using a routine that takes a variable argument list, including ALL get- and set- attribute calls, remember to match the data type of the values passed with the data types that the routines are expecting. Specifically, integers will not be automatically cast to doubles or floats as they would be in a standard function call.

## Provide Correct Number of Parameters with Variable Argument Functions

With variable argument routines, including ALL set- and get- attribute calls, no error checking can be performed to ensure that the right number of parameters are being passed. In GUI & Visualization Services, values match attributes on a one-to-one basis. For example,

```
xvw_set_attribute(image_object, XVW_BELOW, label_object);
```

In contrast, Data Services often has calls where a single attribute will take many parameters, as in:

```
kpds_set_attribute(data_object, KPDS_MAP_SIZE, w, h, d, t, e);
```

It is *imperative* that you send the correct number of parameters to a variable argument function, or you will introduce a bug that is very difficult to find.

## Remember to Close Output Data Objects

You can use Polymorphic Data Services to create and modify an output object, but it will *not* be written out to disk until you call *kpds\_close\_object()*.

## Don't Free Strings Returned From Get Attribute Routines

When a string is returned with the use of a get attribute routine, it will simply be a pointer to memory being used internally by the relevant program services library. You *may not* free the string, or modify the string in any way; it is for inspection purposes only.

## Data Management Services

Before using any *dms\_\**() calls, carefully evaluate whether the functionality you need through Data Management Services is not available from Polymorphic Data Services (*kpds\_\**). Polymorphic Data Services is recommended for application development as it is specifically intended to provide an application programmer's interface. In contrast, Data Management Services is

intended to provide an *infrastructure* programmer's interface. Data Management Services does not enforce a data model, polymorphism, interoperability, or any domain-specific features; there is a great deal more work involved when writing to DMS than there is when using PDS.

### **Edit Files Via *composer*, or From the Command Line, not Both**

*composer* offers a mechanism with which you may select the desired file from a software object and edit it by selecting *Edit* from the *File Operations* menu, which invokes a session with your editor. Alternatively, you may `cd` to the directory in which the file is located, and invoke a session with the editor by hand. Either way is fine; *just don't mix the two at the same time*. For example, if you have used `composer` to invoke a session with the editor on the `*.c` file of a program object, and then in another window you bring up another editor session on the same file (even if you write the file carefully from either window before modifying it in the other window), you are asking for trouble. The software object database will become confused during this procedure, and you are practically guaranteed to lose changes.

## **F. About Library Objects**

There are a number of issues that are particular to library objects. These issues include:

1. Steps for developing a library object,
2. The difference between *unattached* library routines and *lk* routines,
3. The difference between *public*, *private*, and *static* library routines,
4. The standard items that appear in a library source code file,
5. Man pages for library routines and the library as a whole,
6. Private and public library include files.

This section discusses these issues which are specific to library objects.

### **F.1. Steps For Developing a Library Object**

#### **Select Toolbox for Library**

Use `craftsman` to create a new toolbox, or to choose an existing toolbox in which to create the library (see Chapter 2 of the Toolbox Programming Manual for more details).



### Create Library Object

Create the library object using `craftsman` (see Chapter 2 of the Toolbox Programming Manual).

### Develop Library

Develop the library using `composer` (see Chapter 3 of the Toolbox Programming Manual) or manually (`% cd` to the library object directory).

1. Add files containing public library routines to the library using `composer` (see Chapter 3 of the Toolbox Programming Manual).
2. Edit code in public library routines. Editing of `*.c` files can be done manually, or through `composer` (see Chapter 3 of the Toolbox Programming Manual). Private and static library routines may also be added to augment the public library routines.
3. Prototype any "unattached" public library routines in the public library include file. Prototype all private library routines in the "internals.h" file. Prototype any static library routines at the top of the file they appear in. Editing the `*.h` files can be done manually, or with `composer`.
4. Compile the library routines. Use `composer` to execute the compiler, or type:  

```
% kmake
```

in the `src` directory of the library. See Chapter 3 of the Toolbox Programming Manual for more details on compiling from within `composer`.
5. Test and debug library routines; iterate from step 4 as necessary.
6. Fill in library headers carefully. After editing library headers, generate man pages for the public library routines by clicking on the "Generate Code" button of `composer`, or by running  

```
% kmake regen
```

in the `src` directory of the library.
7. For library routine maintenance and modification, repeat process from steps 1, 2 or 6 as necessary.

## F.2. Library Routine Types: Public, Private, Static

*Library routines* will fall into one of three types: *static*, *private*, or *public*.

### Public Routines

A *public* routine is a function that is specifically intended for use by other software objects. Software objects that utilize the public routines in a library may be in the same toolbox as the library,





```

*           This is the header for a PUBLIC function
*
*       Input: argument1 - explanation
*           argument2 - explanation
*           argument3 - explanation
*
*       Output: argument4 - explanation
*           argument5 - explanation
*
*       Returns: TRUE (1) on success, FALSE (0) otherwise
*
* Restrictions: Restrictions on data or input as applicable
*   Written By:
*       Date: Nov 15, 1994
* Declaration:
*
*****/

```

Fields may appear in any order, but they must appear as shown above. The "Routine Name" field is the only *required* field that *must* be filled out. If any other field does not apply, you may delete it, or leave it blank. For the "Input" and "Output" fields, be sure to list arguments in the same order as they appear in the routine declaration.

**Routine Name:**

This line must have the name of the library routine, *followed by a dash*, which is in turn followed by a brief description of the library routine. This brief description will be used in a variety of places; in the man page for the library as a whole, in response to queries of % man -k, and so on. It is important that the brief library routine description not be omitted. If necessary, you may continue the description on another line; adding another star and spacing over will not hurt anything, as in:

```

* Routine Name: myroutine - the brief explanation of my library
*                   routine will not fit on a single line but that's ok
*

```

**Purpose:**

Fill out the purpose field with a *complete* explanation of the library routine. If you want to force a line break, you can use the bang (!) symbol before the words that you want to appear on a new line. For example:

```

* Purpose: Here's the description. Perhaps I am listing something.
*           !1) this text appears on its own line
*           !2) this text appears on its own line, too.

```

**Input:**

List each input argument that appears in the parameter list, *followed by a dash*, followed by the explanation of the argument. Multiple lines can be used if desired.

**Output:**

List each output argument that appears in the parameter list, *followed by a dash*, followed by the explanation of the argument. Multiple lines can be used if desired.

**Returns:**

Explain the return value of the function here; leave blank if the function is *void*. If the function can return more than one value, always state the conditions under which each value is returned.

**Restrictions:**

List any restrictions of the function here. For example, "this routine does not yet deal with complex numbers." Leave blank if there are no restrictions.

**Written By:**

Put your name here. Acknowledge any other people who might have helped you.

**Date:**

The date when the routine was written.

**Declaration:**

This optional field may be manually added if it is needed. In general, the declaration of the routine will be pulled directly into the man page. However, in some cases, this may not be desirable, as the parser will pull everything from the end of the header to the first open bracket. For example, if you have defined external variables, #defines, etc. between the header and the library declaration, these will be pulled into the man page. To avoid this, you can provide your library routine declaration in the header; when you do this, the version in the header will override the actual version. A classic example of when this is useful is when presenting a macro to the user as if it was a function. Use the declaration field to define the API to the macro as you would like it to appear; otherwise the whole macro will be pulled in as the declaration.

### F.3.2. The Private/Static Library Routine Header

While there are no man pages automatically generated from private or static routine headers, it is good practice to fill them out completely and correctly, and keep them up to date. When created via **composer**, a new library file is created with a private/static header already included. This header is as follows:

```

/*-----
|
| Routine Name: foo - short description of foo()
|
| Purpose: This should be a complete description that anyone
|          could understand; it should have acceptable grammar
|          and correct spelling.
|
|          This is the header for a PRIVATE or STATIC function
|
| Input:  argument1 - explanation
|        argument2 - explanation
|        argument3 - explanation
|
| Output: argument4 - explanation
|        argument5 - explanation
|
| Returns: TRUE (1) on success, FALSE (0) otherwise

```

```
|
|   Written By:
|   Date:
| Modifications:
|-----*/
```

The fields in the private routine header are a subset of those listed above for the public library routine header, so they will not be repeated again here. As with public routine headers, if fields are in the private/static header are not used, they may be deleted.

## F.4. Documentation For Libraries

Libraries have two types of documentation:

1. a man page and an html page for each of the public routines in the library,
2. a man page an an html page for the library as a whole.

The documentation is created and maintained as HTML pages. The text is then automatically propagated to the man pages by the code generator.

### F.4.1. Documentation for Public Library Routines

In the preceding section, we covered headers for public library routines. An HTML page and a man page for each public library routine is automatically generated from the header by the code generator. All of the sections in this pair of documents are generated from the library routine header. Each field in the header should be filled out carefully and kept up to date. Whenever a change is made to the header, the code generator should be re-run on the library to update the HTML pages and man pages for the public library routines.

HTML pages for public library routines are generated in the *html/* directory of the library object, and are named "*routine\_name*.htm." Man pages for public library routines are generated in the *man/* directory of the library object, and are named "*routine\_name*.3."

Because they are automatically generated in their entirety, neither the HTML file or the man page that are generated for the library routine should be manually edited. They may, however, be viewed in either their pre-formatted or post-formatted state, either from the command line or from `composer`.

### F.4.2. Documentation for the Library as a Whole

The library object code generator will also generate an HTML page for the library as a whole named "*library.htm*" in the *html/* directory of the library and a man page counterpart named "*library.3*" in the *man/* directory of the library. Unlike the documentation for individual functions, the HTML page for the library should be filled out. After making changes, re-run the code generators to update the man page according to the text in the man page.

There are sections within the HTML page that are automatically generated, however, and these must be left untouched.

The section delineated by the tags

```
<!-- begin_function_list --!>
<!-- end_function_list --!>
```

is an automatically generated function list which should not be edited.

In addition, the copyright section delineated by the tags

```
<!-- begin_short_copyright --!>
<!-- end_short_copyright --!>
```

should be left untouched. The code generators will produce errors if you modify any of these tags, or if you delete them. The code generators will over-write the sections between the tags if you modify their contents.

A summary of the fields in the library HTML file are as follows:

**Library Name**

Automatically generated from software object database information; do not touch

**Description**

Fill this in. Give an overview of the library, as well as detail about the library and its use.

**List of Library Functions**

Automatically generated from software object database information; do not touch.

**Additional Information**

Fill this in. Give any additional information pertaining to the library and its use.

**Location of Source Files**

Initially, automatically generated; change only if necessary

**Location of Public Include Files**

Initially, automatically generated; change only if necessary

**You must include:**

Initially, automatically generated; change only if necessary

**See Also**

Fill this in. Reference the documentation for any other related libraries or programs.

### See Manual

Fill this in. There should be a chapter on the library in the toolbox manual; cite the chapter and manual name here.

### Copyright

Automatically generated from software object database information; do not touch. If you would like to use another copyright, you can use `craftsman` to specify it.

## F.5. Include Files For Libraries

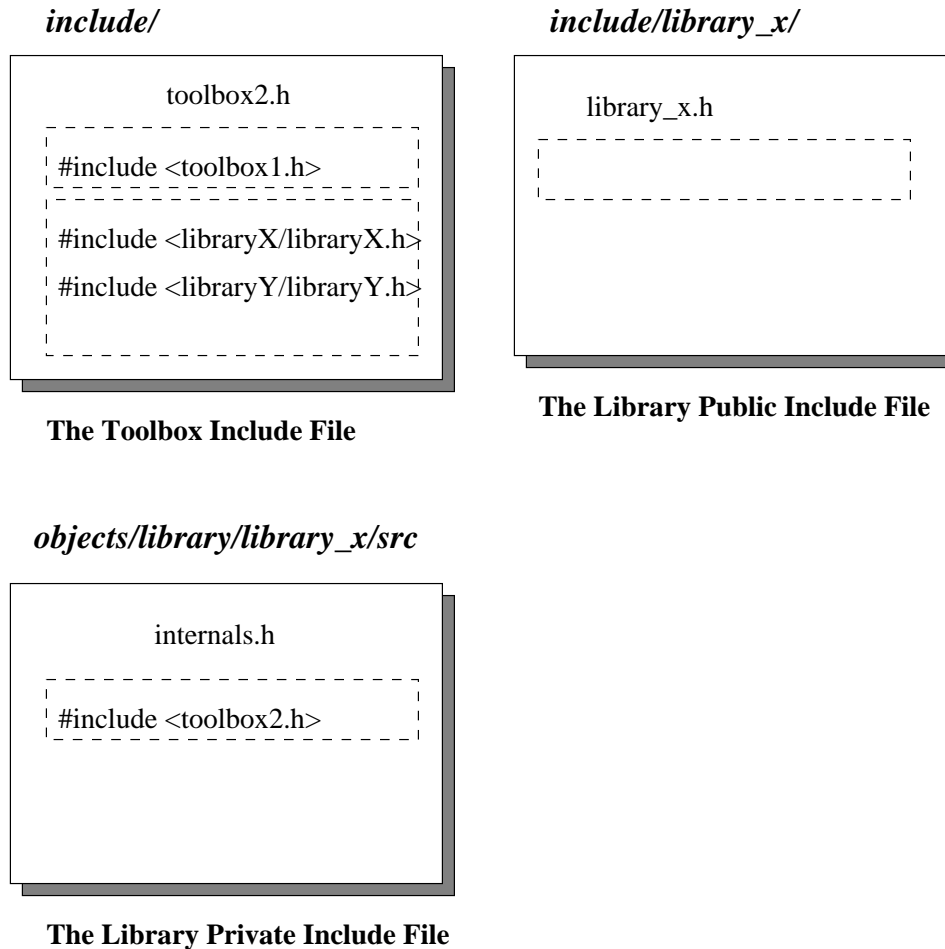
Libraries two include files associated with them:

1. the *private library include file*, named "internals.h." This file is found in the `src/` directory of the software object, and is used for private definitions and declarations.
2. the *public library include file*, named "`library_name.h`." This file is found in the `{library_name}/` directory of the library, and is used for public definitions and declarations. When a "kmake install" is done in that directory, the file is copied to the `$TOOLBOX/include/{library_name}/` directory where it is referenced when needed by other software objects.

Note that the word *private* refers to definitions and declarations that are made available only to the library in question. In contrast, the word *public* refers to definitions and declarations that are made available to other software objects that will be using the library.



## Toolbox2



**Figure 7:** The relationships between the toolbox include file (upper left), the public library include file (upper right), and the private library include file are depicted (lower left).

In the example illustrated above, it is assumed that the "toolbox2" toolbox contains two libraries, X and Y, and that routines in either library X or library Y or both make calls to public library routines provided by toolbox1. The "toolbox2.h" toolbox include file must include the public include files for each of its libraries, X and Y. Because of X and Y's on library routines in the "toolbox1" toolbox, "toolbox2.h" must include "toolbox1.h."

The private "internals.h" file of library X includes "toolbox2.h;" thus, library X routines will have access to prototypes, #defines, and data structures of public routines in both library X and library Y. Because the "toolbox2" include file includes, in turn, the "toolbox1" include file, library X will also have access to public routines offered by "toolbox1." Because of the inclusion of the toolbox include file by the "internals.h" file, *neither of the include files for library X need to include the public include files of the libraries they depend on.* Note that the public include file for library X doesn't need to include any other include files at all.

Include files always have the same elements in the same order. They begin with an RCS header, which is followed by the copyright, and a header for library include files. Special `#ifndef/#define/#endif` statements must

appear in every include file. Locations in the include file are specifically earmarked for #defines, typedefs, global variable declarations, macros, and routine definitions. The elements common to both the "internals.h" private library include file and the "*library\_name.h*" public library include file are as follows:

### RCS Header

Like all other files in VisiQuest, library include files begin with an RCS header.

### Copyright

The copyright (if any) appears after the RCS header. Use *craftsman* to change the copyright if desired.

### Include File Header

Fill out the purpose, the toolbox name, and the authors name. Modifications to the include file can be added to the header as needed.

### ifndef/define/endif statements

Every library include file has its header followed by the lines:

```
#ifndef _library_name_h
#define _library_name_h
```

At very end of the file is the endif:

```
#endif _library_name_h
```

Your library name will be substituted for *library name* automatically. The reason for including these lines is that with a large and complicated software system such as VisiQuest, it sometimes happens that an include file gets included more than once. These lines protect against the actual definitions in the include file getting redefined (resulting in warnings and/or errors) if the include file gets accidentally included more than once.

## F.5.1. The Library's Private Include File

Every library should have a *private include file*; this file is by convention named "internals.h," and is found in the *src/* directory of the library object. The private include file for the library will be included by every \*.c file in the library. The private include file for a library contains the prototypes of the private routines in the library, #defines used by the private library routines, definitions of structures used in private library routines, private macros written for the library, and so on. The contents of the "internals.h" file will be completely determined by the private routines in the library. Elements of the "internals.h" file include:

### #include

Under the comment that reads,

```
/*-----*
|           #include
|-----*/
```

should always be a #include statement that includes the "*toolbox.h*" file:

```
#include <toolbox.h>
```

This frees the "internals.h" file from having to include the public library routines of all other libraries to which calls are made. In addition, if there are other private include files used by the library, these should also have include statements here.

### **#defines**

Under the comment that reads,

```
/*-----*  
|         #defines  
-----*/
```

define any constants that are used by private library routines, but not used in the arguments of public library routines.

### **typedefs**

Under the comment that reads,

```
/*-----*  
|         typedefs  
-----*/
```

declare any data structures that are used by private library routines, but not used in the arguments of public library routines.

### **global variable declarations**

Under the comment that reads,

```
/*-----*  
|         global variable declarations  
-----*/
```

declare any global variables used by the library. Note that global variables should be used sparingly in a library.

### **macros**

Under the comment that reads,

```
/*-----*  
|         macros  
-----*/
```

declare any macros that are used in the library, but are not to be used outside the library.

### **routine definitions**

Under the comment that reads,

```
/*-----*  
|         routine definitions  
-----*/
```

you should insert prototypes for all private library routines.

## F.5.2. The Library's Public Include File

Every library must have a *public include file*; this file is by convention named "*library\_name.h*" and is found in the *library\_name* directory of the library object.

When a "kmake install" is done on the library, this file will be copied to the *\$TOOLBOX/include/{library\_name}/*, where it will be referenced by the "toolbox.h" file for the toolbox in which the library appears. Thus, it is important that a "kmake install" be done any time this public include file is changed. Otherwise, other software that uses the public include will not be getting up-to-date information and may fail to compile.

Elements of the public include file include:

### **#include**

Under the comment that reads,

```
/*-----*
|         #include
|-----*/
```

you should only have #include statements if you have divided up the definitions, #defines, prototypes, etc, into two or more files. For example, suppose a "image analysis" library has a dual purpose, such as "clustering" and "classification." It might be neater to have the definitions for "clustering" in one include file, and the definitions for "classification" in another include file. In this case, there might be three include files associated with the library: "clustering.h," "classification.h," and the public library include "analysis.h," where "analysis.h" used the statements:

```
#include <analysis/clustering.h>
#include <analysis/classification.h>
```

Dependencies on other toolboxes such as the *datamanip* toolbox will be taken care of in the toolbox include file; the toolbox include file is, in turn, included by the private include file.

### **#defines**

Under the comment that reads,

```
/*-----*
|         #defines
|-----*/
```

define any constants that are used as arguments to public library routines.

### **typedefs**

Under the comment that reads,

```
/*-----*
|         typedefs
|-----*/
```

declare any data structures that are passed as arguments to public library routines.

### **global variable declarations**

Under the comment that reads,

```
/*-----*  
|         global variable declarations  
-----*/
```

there should not be anything. Any variables that are global to a library belong in the *private* library include file. Global variables should not be made available to calling routines.

### **macros**

Under the comment that reads,

```
/*-----*  
|         macros  
-----*/
```

declare any macros that are to be presented to the outside world as public functions. Remember that such macros should have a complete public library routine header, so that they will have man pages generated for them. Also note that macros in a public library include file must have the library identification prefix. Please see the section on "Programming Standards" for more information.

### **routine definitions**

Under the comment that reads,

```
/*-----*  
|         routine definitions  
-----*/
```

you must manually insert prototypes for all public library routines.

## G. About Kroutines

Kroutines are the most common form of software object in VisiQuest. Most of the material in this section applies to xvroutines as well as kroutines. Therefore, the material that is common to both kroutines and xvroutines is presented here, with respect to kroutines. The subsequent sections on xvroutines will simply reference this material when applicable.

### G.1. Steps For Developing a Kroutine Object

#### Select Toolbox For Kroutine

Use `craftsman` to create a new toolbox, or to choose an existing toolbox in which to create the kroutine (see Chapter 2 of the Toolbox Programming Manual for more details on `craftsman`).

#### Create Kroutine Program Object

Create the kroutine object using `craftsman` (see Chapter 2 of the Toolbox Programming Manual). At this time, you must decide whether the kroutine will have its functionality contained in an function of a library (making that functionality available to other software objects), or whether the kroutine itself will contain the functionality (making that functionality unavailable outside the kroutine).

#### Develop Kroutine

Develop the kroutine object using `composer` (see Chapter 3 of the Toolbox Programming Manual) or manually (`% cd` to the kroutine program object directory).

1. Specify the command line arguments of the kroutine in the `*.pane` file; use `Guise` to interactively create the GUI (see Chapter 4 of the Toolbox Programming Manual for more information). After changing the `*.pane` file, always regenerate code by using the "Generate Code" button from `composer`'s Make menu or by running

```
% kmake regen
```

in the `src/` directory of the kroutine.

2. Edit the code of the kroutine to add functionality. If the kroutine functionality is to be developed in a library function, most of the coding will be done in the library. Editing the `*.c` files can be done manually or through `composer`.
3. Compile the kroutine. This can be done via `composer` or by typing

```
% kmake
```

in the `src` directory of the kroutine.
4. Test and debug kroutine; iterate from step 1 as necessary.
5. Write the HTML page carefully. After editing the HTML page, generate the help and man pages for the kroutine by clicking on the "Generate Code" button of `composer`, or by

running

```
% kmake regen
```

in the *src/* directory of the kroutine.

6. When you are ready, install your kroutine in the bin directory of your toolbox. This can be done in *composer* or by typing

```
% kmake install
```

in the *src* directory of the kroutine.

7. For kroutine maintenance and modification, repeat process from steps 1, 2 or 5 as necessary.

## G.2. The \*.pane UIS File

The "*program.pane*" file, or \*.pane file, describes the Command Line User Interface (CLUI) of the program. It also describes the Graphical User Interface (GUI) of the program that is displayed when the program is accessed from VisiQuest, or when it is executed with its [-gui] option. The \*.pane file is used as input to the code generators, which generate code according to the program arguments specified therein.

A \*.pane file should always define a GUI that consists of a *single pane* on a *single subform*. Thus, while the VisiQuest GUI can support three levels of hierarchy (master form / subform / pane), the \*.pane file always defines only a single pane.

The image shows a graphical user interface window titled "Pane". At the top left is a pull-down menu labeled "options". To its right are three buttons labeled "Run", "Help", and "Quit". Below these are four text input fields. The first is labeled "Input File", the second "Integer", the third "Float", and the fourth "Float" with a grey square icon to its left. At the bottom is a text input field labeled "Logical" containing the text "true".

---

**Figure 8:** The \*.pane defines a GUI that consists of a single pane on a single subform. In the *Options* pull-down menu, *Run*, *Help* and *Quit* buttons are standard items included on GUI's defined by \*.pane files. The selections on the pane correspond to the arguments of the program.

---

If a program does not require any special arguments of its own, it must have an "empty" \*.pane file; such a \*.pane file contains only those UIS lines that are required in every UIS file, but will not have any UIS lines that specify command line arguments. Even programs using an "empty" \*.pane file will still automatically get the [-V], [-U], [-A], [-gui], [-A], and [-a] that are standard to all VisiQuest programs.

**Guise** is used to create the \*.pane file for a program. Recall that **composer** automatically calls **Guise** when you select the \*.pane file from the list of files on the **composer** main form.

When creating your \*.pane file with **Guise**, you should pay special attention to the following fields in each selection description:

**The optional field**

This determines whether a program argument is required or optional on the command line.

**The default value**

A valid default value should be provided for each argument, ESPECIALLY if the argument is optional.

**The description field**

This should be a BRIEF but EXPLICIT description of the argument. Explanations should be limited to a maximum of 40 characters if possible.

**Bounds on integers, floats, and doubles**

Sensible bounds should set on integer, float, and double arguments when applicable (if appropriate, they can be set to be unbounded).

**The variable name**

Variable names may be as many characters long as desired, but variable names are recommended to be from 1 - 10 characters long.

## G.3. Files Generated

Using information specified in the \*.pane file, the code generators create several files for a program object.

**The main driver (main.c)**

This file, automatically generated in the *src/* directory of the program object, contains the main program.

**The developer's driver (*program.c*, or \*.c)**

This file is generated for developer use in the *src/* directory of the program object. It contains skeletons for the routines called by the main driver, to which code is added to give the program its functionality.



**The main include file (main.h)**

The primary purpose of this automatically generated file is to define the *CLUI Information structure*, the C structure in which the values of the command line arguments will be stored for access by your program. A later section covers the CLUI Information structure in detail.

**The developer's include file (program.h, or \*.h)**

This file is generated for developer use in the *src/* directory of the program object.

**The HTML page (program.htm, or \*.htm)**

This file is generated for developer use in the *html/* directory of the program object. The source location for program documentation, text is added to this file and is then propagated by the code generators to the other documentation files. It provides a HTML-formatted version of the documentation for web-based reading.

**The man page (program.1, or \*.1)**

This file is automatically generated in the *man/* directory of the program object. It is a version of the documentation provided in the HTML page containing *roff* formatting commands. This man page is created to make program documentation will be accessible via the *kman* command.

**The help page (program.hlp, or \*.hlp)**

This file is automatically generated in the *help/* directory of the program object. It is a version of the documentation provided in the HTML page containing VisiQuest macro formatting commands. This online help page makes program documentation available when the program is accessed as a glyph via VisiQuest.

## G.4. The CLUI Information Structure

The CLUI Information structure for your program is a single-level data structure, automatically generated in the \*.h file. The automatically generated *program\_get\_args()* routine in the *usage.c* file of your program will fill out this CLUI Info structure with the values provided by the user for command line arguments, so that you may use them as needed in your program.

There will be a pointer to CLUI Info structure globally declared for you in your *program.h* file; this pointer is always named *clui\_info*. Fields in the CLUI Info structure are named according to the variable names provided on the UIS lines of your \*.pane file. The fields that are generated depend on the type of argument used. The following table summarizes, through the use of examples, the variables that are generated for the different command line arguments. Note that the table lists individual fields in the CLUI Info structure as they would be declared in the *program.h* file; these must be referenced properly in your program given the fact that they are members of a structure. For example, the *int choice\_cycle* field listed for the *Cycle* entry in the table would be referred to in actual code as *clui\_info->choice\_cycle*.

Example Argument Information Variables		
Argument Type (UIS Type Flag)	Variable On UIS Line	Variables Generated In CLUI Info Struct
Action Button (-a)	ignored	none
Blank (-b)	ignored	none
Cycle (-c)	choice	int choice_cycle; /* integer value of choice */ char *choice_label; /* string associated w/ int value */ int choice_flag; /* TRUE if [-choice value] on cmd line */
Double (-h)	n	int n_double; /* double value of n */ int n_flag; /* TRUE if [-n value] on cmd line */
Flag (-t)	rev	int rev_flag; /* TRUE if [-flag] on cmd line */
Float (-f)	s	float s_float; /* float value of s */ int s_flag; /* TRUE if [-s value] on cmd line */
Help Button (-H)	ignored	none
Input File (-I)	i1	char *i1_file; /* filename */ int i1_flag; /* TRUE if [-i1 value] on cmd line */
Integer (-i)	x	int x_int; /* integer value of x */ int x_flag; /* TRUE if [-x value] on cmd line */
Lists, Displaylists, (-x), (-z)	mode	int mode_list; /* integer value of mode */ char *mode_label; /* string associated w/ int value */ int mode_flag; /* TRUE if [-mode value] on cmd line */
Logical (-l)	fill	int fill_logic; /* boolean value of fill */ int fill_flag; /* TRUE if [-fill value] on cmd line */
Mutually Exclusive Group (-C)	N/A	No variables are generated for group as a whole; however, variables are generated for each of the members in the Mutually Exclusive group.
Mutually Inclusive Group (-B)	N/A	No variables are generated for group as a whole; however, variables are generated for each of the members in the Mutually Inclusive group.
Loose Group (-K)	N/A	No variables are generated for group as a whole; however, variables are generated for each of the members in the Loose group.
Output File (-O)	o2	char *o2_file; /* filename */ int o2_flag; /* TRUE if [-o2 value] on cmd line */
Quit Button (-Q)	ignored	none
Routine (-R)	N/A	none

Example Argument Information Variables		
Argument Type (UIS Type Flag)	Variable On UIS Line	Variables Generated In CLUI Info Struct
String (-s)	name	char *name_string; /* string value of name */ int name_flag; /* TRUE if [-name string] on cmd line */
String List (-y)	color	char *color_string; /* string value of color */ int color_flag; /* TRUE if [-color string] on cmd line */
Toggle (-T)	pt	int pt_toggle; /* int value (flag, logical, int toggles) */ - or - char *pt_toggle; /* string value (file, string toggles) */ - or - float pt_toggle; /* float value (float toggles) */  int pt_flag; /* TRUE if [-pt value] on cmd line */
Workspace (-w)	ignored	none
Array (Integer) (-c)	ai	int *ai_array; /*-- integer array DATA --*/ size_t ai_width; /*-- integer array WIDTH --*/ size_t ai_height; /*-- integer array HEIGHT --*/ int ai_flag; /*-- integer array FLAG --*/
Array (Float) (-c)	fi	float *fi_array; /*-- float array DATA --*/ size_t fi_width; /*-- float array WIDTH --*/ size_t fi_height; /*-- float array HEIGHT --*/ int fi_flag; /*-- float array FLAG --*/
Array (Double) (-c)	di	double *di_array; /*-- double array DATA --*/ size_t di_width; /*-- double array WIDTH --*/ size_t di_height; /*-- double array HEIGHT --*/ int di_flag; /*-- double array FLAG --*/
Array (Input File) (-c)	ini	kstring *ini_array; /*-- input file array DATA --*/ size_t ini_num; /*-- input file array SIZE --*/ int ini_flag; /*-- input file array FLAG --*/

---

**Table 1:** The table contains examples of the variables generated for the various CLUI arguments.

---

## G.5. The Main Driver (main.c)

The main.c file is an automatically generated file that should not be edited. It contains four elements:

- The *main program*.  
The main calls *VisiQuest\_init()* to do VisiQuest initializations, *kclui\_init()* to do command line user

interface initiations. Finally, it calls the `run_program()` routine, the routine that you will write to give the program its functionality. For xv routines, there is additional code generated to support the GUI.

- The `program_get_args()` routine.  
The `program_get_args()` routine retrieves program arguments from the command line (or from the glyph's pane in VisiQuest) and stores them in the CLUI info structure where they can be easily referenced.
- The `program_usage_additions()` routine  
This routine prints the short description of the program and then calls the `program_extra_usage_additions()` routine (see the following section).
- The `program_free_args()` routine  
This routine frees the `clui_info` structure and then calls the `program_extra_free_args()` routine (see the following section).
- The definition of the `clui_uis`  
This is an array of strings which creates a compiled version of the UIS file(s) for the program, enabling the program to be removed from the VisiQuest system and executed without its UIS file(s).

## G.6. The Developer's Driver (\*.c)

The `program.c` file is the file containing the driver for your program. In this file, you add the code which will give the program its functionality. It contains three routines which will be called at the appropriate times from the automatically generated main program:

### The Run Routine

The header and a skeleton definition of the `run_program()` routine is automatically generated and called from the main program. It takes no arguments and returns a status of TRUE (success) or FALSE (failure). Add code to this routine as desired to make the program functional.

### The Usage Additions Routine

The `program_extra_usage_additions()` routine is called when the program is run with the standard [-U] and [-usage] arguments. In this routine, you may add additional print statements to augment the usage statement for your program. These statements must be in the form:

```
kfprintf(kstderr, "This is the message");
```

### **The Free Args Routine**

The *program\_extra\_free\_args()* routine is called to do memory cleanup before the program exits. In this routine, you may add code to free any memory allocated by your program in the course of execution. Because the *program.c* file is not touched by the code generators after it is initially created, you may add additional routines as desired.

## **G.7. The Main Include File (main.h)**

The main.h file is an automatically generated file. It contains the following elements:

- The copyright
- A warning about not editing this automatically generated file
- A VisiQuest include file header
- A statement to include the public include file for the toolbox
- The definition for the CLUI Info structure
- Prototypes for automatically generated routines.

## **G.8. The Developer's Include File (\*.h)**

The \*.h file is generated for use by the developer. It is a shell containing the RCS header and copyright statement, an include file header and a statement to include the "main.h" file. You may add code to this driver include file as needed.

## **G.9. Program Documentation (\*.htm, \*.1, \*.hlp)**

There are three program documentation files, the HTML file, the man page, and the help file. Text is Elements of the program documentation include:

### **Program Name**

The program name is the name of the software object.

### **Short Program Description**

Contains a concise, one-line (no carriage returns) description of the program. Appears after the program name.

### **Long Program Description**

Contains a complete and detailed program description which may be as long as desired. It should cover such issues as algorithms used, program philosophy, overall program intent, and so on.

### **Required and Optional Arguments**

The arguments sections are automatically generated; do not edit. Arguments to a program are changed by modifying the \*.pane file with `guise` and re-generating the program.

### **Examples**

One or more examples of the program use. A variety of examples covering different situations is best for programs that can be used in different ways.

### **See Also**

Other programs that are related to this program in some way.

### **Program Restrictions**

Any restrictions on the use of the program or limitations of the program.

### **References**

Provide any applicable references for the algorithms used by your program; journal articles, textbooks, conference proceedings, and web pages are appropriate.

### **Copyright**

The copyright is automatically generated. If you would like to use another copyright, you can use `craftsman` to specify it.

The code generator will generate a \*.htm HTML page for the program in the *html* directory of the program object. An HTML page for a program looks like this:

```
<HTML>
<HEAD>
  <TITLE> colortest Online Help Page </TITLE>
</HEAD>
<BODY>
<H2>TESTTB commands</H2><HR>
<H3>PROGRAM NAME </H3>
colortest - Default Compiled Workspace
<H3> DESCRIPTION </H3>
<!-- begin_arguments -->
<H3> REQUIRED ARGUMENTS </H3>
none
<H3> OPTIONAL ARGUMENTS </H3>
none
<!-- end_arguments -->
<H3> EXAMPLES </H3>
<H3> SEE ALSO </H3>
<H3> RESTRICTIONS </H3>
```

```

<H3> REFERENCES </H3>
<!-- begin_short_copyright -->
/*
 * Copyright (C) 1993 - 1999, AccuSoft Corporation, ("AccuSoft Corporation").
 * All rights reserved. See $BOOTSTRAP/repos/license/License or run
 * klicense.
 */
<!-- end_short_copyright -->
</BODY>
</HTML>

```

The HTML file contains tags that are used to delineate automatically generated text blocks. As mentioned before, you should not change these tags in any way, or edit the text between them. The following table describes the two text blocks associated with the program HTML page and their associated tags.

Tags Used in HTML file		
Text Block	Begin Tag	End Tag
Required and Optional Arguments	<!-- begin_arguments -->	<!-- end_arguments -->
Short Copyright	<!-- begin_short_copyright -->	<!-- end_short_copyright -->

Any documentation outside these tags may be edited as desired. The VisiQuest program documentation sections such as description, examples, see also, etc., are conventions, not requirements. A converter is used to translate the HTML page into *roff* for the man page. Another converter is used to translate the HTML page into VisiQuest macros for the online help page. However, please be aware of this IMPORTANT NOTE: translation of HTML tables is not supported! If you use tables in your HTML page, the table will not appear in the man page or in the online help.

## G.10. Program Structure for Kroutines

For code reusability, it is often desirable to divide the functionality of the kroutine from the kroutine itself. For example, you might be writing code implementing a particular algorithm. You may want to design the code such that there is a kroutine that performs that algorithm on its input file and creates an output file; in addition you may also want to write other kroutines that call the code performing that algorithm, as part of a larger operation.

Alternatively, you may decide that the functionality of the kroutine is to be specific to the kroutine, and that it does not make sense to put its functionality in a library routine, since the library routine would probably never be called except by the kroutine that you are writing.

In the latter case, when all code for the kroutine is written in the kroutine itself, the following sections do not apply.

## G.11. Making kroutine functionality re-usable with an lkroutine

When the functionality of a kroutine is to be made available to other kroutines for code reusability, the kroutine is designed such that it is divided into two components:

1. the kroutine driver
2. the *lkroutine*, a library function in a separate library object, containing the functionality for the kroutine

For example, the `karith1` kroutine of the `datamanip` toolbox has its `*.c` file in the `karith1` software object directory; its `lkroutine`, `lkarith1()`, is located in the `kdatamanip` library.

The preliminary work is done in the *driver*. Preliminary work includes argument checking, opening and initializing input and output data objects, and setting up the parameters that will be passed into the library. The `lkroutine` contains the source code for executing the function on the data.

In contrast to earlier versions of VisiQuest, however, the `lkroutine` library file is not managed as part of the kroutine within `composer`. The `lkroutine` file is edited separately, as part of the library object containing it, and not from the the kroutine itself.

The following sections discuss how to divide the code successfully between the kroutine and the `lkroutine` so that the `lkroutine` can be called correctly from other kroutines and `xvroutines`, in addition to providing the complete functionality for the kroutine you are writing.

### **G.11.1. The kroutine driver**

The `run_kroutine()` function, in the `program.c` file of the kroutine, serves as the kroutine driver. It should open all input and output data objects and verify them before passing them into the library routine. It is the responsibility of the driver code to copy any data segments or attributes that are not directly manipulated by the library function to the output object. This can be thought of as pre-initializing the data object. It applies if you wish to preserve and pass through to the output all extra information and data associated with the input object. In other words, the destination object is initialized to be the same as the source object, and the library routine changes only those segments and attributes that it modifies directly. We strongly recommend this approach so that your kroutine will not have detrimental side effects on the users' data, such as removing parts of it. Any time your program will cause unexpected side effects on the data object (for example, removing an object's comment field) this must be documented in the man pages. Transferring segments and attributes is accomplished by using the `<app_serv>_copy_data()` and `<app_serv>_copy_attributes()` function calls.

### **G.11.2. Set Up Parameters**

The kroutine's parameters are defined in its CLUI Information structure parameter. In general, all kroutine parameters should be passed to the associated `lkroutine`. Opening and verification of input and output data objects is done before the call to the library routine so that the input and output data objects may be passed directly into the `lkroutine`. Some additional processing of parameters may also be done before the `lkroutine` is called.



### G.11.3. Call the Lkroutine

The *lkroutine* is developed as a function contained in a separate library object. It does all the processing for the kroutine. All krountines should return TRUE upon success and FALSE upon failure. If the library routine fails, an appropriate error message should be printed using *kerror()*. No library routine should *ever* exit.

### G.11.4. Check Status Returned By Lkroutine

When calling the library routine, the driver should verify that the lkroutine returned TRUE. If it returns FALSE, close all opened objects before exiting. To avoid multiple pop up windows appearing when the kroutine fails, you should *not* print an error in the driver if the corresponding library call fails. It is the library calls responsibility to print the error; the driver just needs to close and exit.

### G.11.5. The Kroutine Driver Return Status

When the driver exits, it must return a status of TRUE or FALSE. If the lkroutine has failed, exit with a status of FALSE. Otherwise, it should return TRUE.

### G.11.6. Example Kroutine Driver Code

Below is a code segment that gives an example of the flow of a kroutine driver. In this example, we use polymorphic data services to open the input object and the output object, and call the lkroutine.

```
int run_kroutine(void)
{
    kobject source, destination; /* input and output data objects      */
    int     integer_argument;   /* example kroutine has one int argument */
    double  double_argument;    /* example kroutine has one double argument */
    char    *routine_name = "run_kroutine";

    /*
     * Open the input object. Give an error and exit on failure.
     */
    if (!(source = kpds_open_input_object(clui_info->i_file))
        {
            kerror(NULL, routine_name, "Cannot open input object.");
            return FALSE;
        }

    /*
     * Open the output object. Give an error, remember to close
     * the source object, and exit on failure.
     */
    if ( !(dest = kpds_open_output_object(clui_info ->o_file) )
        {
            kerror(NULL, routine_name, "Cannot open output object.");
            kpds_close_object(source);
            return FALSE;
        }
}
```

```

/*
 * Pre-initialize the output object so that it is the same as the input
 * object by using the application service's copy object function. This
 * function copies both the data and the attributes from the source object
 * to the destination object.
 */
if (!kpds_copy_object(source, destination))
{
    kerror(NULL, routine_name, "Cannot open copy input to output.");
    kpds_close_object(source);
    kpds_close_object(destination);
    return FALSE;
}

integer_argument = clui_info->var1_int;    /* perhaps do something with */
double_argument  = clui_info->var1_double; /* args before passing them */

/*
 * Call the lkroutine with open input & output data objects,
 * plus values provided for all command line arguments. If the lkroutine
 * fails, don't call kerror(); the lkroutine should have already called it.
 */
if (!lkroutine(source, integer_argument, double_argument, destination))
{
    kpds_close_object(source);
    kpds_close_object(destination);
    kexit(KEXIT_FAILURE);
}

kpds_close_object(source);
kpds_close_object(destination);

return TRUE;
}

```

The lkroutine contains the source code for performing operations on the data. This enables the lkroutine to be called from different programs as well as from the original kroutine for which it was written.

### G.11.7. Return Status

The lkroutine should always be declared to return an integer. Consequently it will return FALSE (zero) upon failure or TRUE (non-zero) upon success. The lkroutine should be very robust; it should not crash for any reason. In order to obtain this objective, all the necessary error checking, such as verifying that the data objects passed in are valid and that parameters are within valid ranges, should be done in the lkroutine. The lkroutine should always return to the kroutine driver, whether it completed its work successfully or unsuccessfully. That is, the library should *never* call *kexit()*, but should call *return(FALSE)*.

### G.11.8. Calling Other Functions

If the lkroutine needs to call any existing library routines, the calling format should be the same as that of the main program; that is, the lkroutine should check if the subordinate routine returns TRUE (non-zero) or FALSE (zero) upon success or failure, respectively. Do not print an error message if the library routine that

you called failed (the function should already have printed the message before returning).

### G.11.9. Error Messages and Information

If an error occurs in a lkroutine, the error message should be printed from the library using *kerror()*. If the lkroutine needs to print a warning, use *kwarn()*. Use *kinfo* to print general information. *Kinfo* has several modes of verbosity that can be set by the `VisiQuest_NOTIFY` environment variable. For example, if you want to print large amounts of information for debugging purposes, set the *notify\_type* in the *kinfo* function call to *KVERBOSE*. The messages will be printed only when users set their `VisiQuest_NOTIFY` environment variable to `VERBOSE`. Most often, *kinfo()* is called with the *KSTANDARD* verbosity level.

### G.11.10. Side Effects

Library routines should *never* alter the input or source data object (unless the source is also the destination, which is not a recommended practice), and library routines should not have *side effects* on the output or destination object. Alterations associated with the function, such as changing the size attributes of the destination object during a resize operation, are *not* considered side effects. But changing the data type, or copying, creating or altering data segments that are not associated with the operation, would be side effects. If side effects are produced by the library, they must be described in the *Side Effects* field of the library header. This information will then be propagated to the library man page that is displayed by *kman*.

The general guidelines for insuring no side effects on an output data object are to:

- First set all attributes that are a direct result of the library function, on the output object. For example, in the *karith2* routine, the data type and the size of the output object may change, depending on the attributes of the input object. These attributes should be set on the output object.
- Then, create a reference to the output object, and perform all other attribute sets on the reference object. You can put your data into either the reference object or the object itself — the results will be the same. It is better to put data into the reference object so that output position attributes are not altered.
- Reference the input object and use the reference when doing data and attribute manipulation and when getting data.
- Close the reference objects before returning to the calling routine.

### G.11.11. Example Lkroutine

The lkroutine is developed in a file in a separate library object. An example library pseudo-code sequence that prevents side effects is given below. In this example, polymorphic data services is used to operate on the value segment of the input data object.

```
int lkroutine(  
    kobject source,  
    int v1,
```

```

double v2,
kobject destination)
{
    kaddr    data = NULL;
    kobject source_ref, dest_ref;

    /*
     * Make preliminary checks to ensure that valid parameters and
     * data objects were passed in.
     */
    if (!source)        return(FALSE);
    if (!destination)  return(FALSE);

    /*
     * The existence of the data with which you will be working (such as value,
     * mask, map, etc) in the destination object should be checked because the
     * destination object may not already have that type of data.  If that type
     * of data does not already exist, it must be created, otherwise future
     * function calls that operate on it will fail.  In this example, we are
     * operating on the value segment.
     */
    if (!kpds_query_value(destination)) {
        if (!kpds_create_value(destination))
            return(FALSE);
    }

    /*
     * If you are going to set any attributes on the source object,
     * set up a reference to the source object so that the library
     * does not have any side effects on it.
     */
    if (!(source_ref = kpds_reference(source)))    return(FALSE);

    /*
     * Attribute changes that are necessary for the internal operation
     * of the library function, for example, POSITION, should be applied to
     * the _reference_ source object.
     */
    if (!kpds_set_attribute(source_ref, KPDS_VALUE_POSITION, 0, 0, 0, 0, 0))
        return(FALSE);

    /*
     * Set those attributes on the destination object that should
     * change as a result of library program.
     */
    if (!kpds_set_attribute(destination, <KPDS_ATTRIBUTE>, v1, v2))
        return(FALSE);

    /*
     * Set up a reference to the destination object so that the
     * library does not have any side effects on other destination
     * object attributes.  Then set attributes that are necessary
     * for the internal operation of the library function on the
     * referenced destination object.
     */
    if (!(dest_ref = <kpds_reference(destination)))    return(FALSE);
    if (!<kpds_set_attribute(dest_ref, KPDS_VALUE_POSITION, 0, 0, 0, 0, 0))
        return(FALSE);
}

```

```

    /*
    * Finally, get the data from the source object, process it, and put it to
    * the destination object. Passing in NULL as the last parameter causes
    * the get data function to allocate memory for the data.
    */
    data = kpds_get_data(i, <PRIMITIVE>, NULL);

    kpds_put_data(o, <PRIMITIVE>, data);

    /*
    * Before returning, free memory that was allocated within this function.
    */

    kfree(data);

    /*
    * Close the referenced input and output objects.
    */
    kpds_close_object(source_ref);
    kpds_close_object(dest_ref);

    return(TRUE);
}

```

## G.12. Continuous Run Programs

### G.12.1. Introduction to Continuous-Run Programs

Continuous-run programs are different from conventional programs in the way that they operate when executed as glyphs within VisiQuest. Typically, VisiQuest executes each operator, or glyph, by executing its associated program as a separate process. Conventional programs are written to start up, gather their command-line arguments, perform some processing, and then exit. Upon exit, the scheduler within VisiQuest will schedule all operators downstream from the completed process.

Continuous-run programs are slightly different from conventional programs. Like conventional programs, continuous-run programs also start up, gather their command-line arguments, and perform some processing. But then, rather than exiting, they will sit "idle", until they are scheduled again with new input to process. When they are re-scheduled because of a change in their input data, they will re-gather their command-line arguments, and repeat the processing. They will run like this, continuously, until VisiQuest tells them to exit, or some internal condition causes them to exit.

When executed outside VisiQuest, continuous-run programs behave identically to conventional programs.

### G.12.2. How Continuous-Run Programs Work

Communication between operators and VisiQuest is done via a bi-directional Inter-Process Communication (IPC) mechanism that is embedded in the VisiQuest transport abstraction. This abstraction is utilized by Data Services for all of its file formats. Output data objects automatically use the IPC to notify VisiQuest when they

are closed and new input is available for the next glyph in the visual program. Therefore, the IPC is transparent to the VisiQuest developer. You need never make any explicit IPC calls yourself; however, use of data objects and Data Services is necessary to achieve IPC and continuous run support.

VisiQuest initiates operators in a workspace to run based on data availability. If new data is available at the input ports of a glyph, the glyph will indicate that it is ready to run. In turn, VisiQuest will execute the operator if it is not already running. Each operator signals output file availability back to VisiQuest, which uses the information for subsequent scheduling.

The IPC mechanism provides great flexibility in directing the VisiQuest scheduler. For example, you can write a kroutine that gates a single input to one of several outputs, causing only the branch of the workspace connected to that one output to fire. You can also connect direct feedback within a workspace using a Merge glyph.

### **Kroutines**

In the automatically generated main of a continuous-run kroutine, the *kclui\_init()* call handles the initialization of your CLUI structure with the initial arguments from the kroutine's pane. After each iteration of the while loop containing the *kprocess\_contrun()* call, your kroutine will pause to wait for another the signal from VisiQuest. When the signal arrives, it will gather the latest input parameters from the kroutine's pane, and continue for another iteration through the while loop. The generation of new output is signaled back to VisiQuest, which uses the information to schedule downstream operators. When VisiQuest signals that the routine should end, the *kprocess\_contrun()* routine returns FALSE, ending the loop and allowing the program to exit.

### **Xvroutines**

Kroutines, read data, process it, write it, and end; thus, in order to make a kroutine continuous-run, its main must be generated with a special loop to prevent it from exiting as soon as its operations are finished. In contrast, xvroutines operate in an event loop which is driven by user input; this loop prevents them from exiting until the user performs an action which exits the program (such as clicking on the Quit button). Thus, there is no distinction between an xvroutine which is "continuous-run" and one which is not; however, as the developer, you must structure your xvroutine code properly in order for it to work correctly in VisiQuest and react properly when new input is received. The guidelines for this are covered later.

## **G.12.3. Creating Continuous-Run Programs**

In contrast to earlier versions of VisiQuest, creation of continuous-run kroutines in VisiQuest 2001 is done automatically. To create a continuous-run kroutine in VisiQuest 2001, simply set the "Continuous Run Driver?" option to "Yes" in *craftsman* when you are creating the kroutine. This will cause the kroutine code generator to use a *kprocess\_contrun()* continuous-run loop in the automatically generated main, rather than the conventional call to the program driver. There is nothing further that you need to do, as far as kroutine coding is concerned.

In VisiQuest 2001, all xvroutines have the continuous-run capability. Thus, in *craftsman*, there is no "Continuous Run Driver?" option to set when creating the xvroutine. However, in order to ensure that your xvroutine

supports the continuous-run capability correctly, you must follow some special coding guidelines for collecting and using program arguments. These will be explained in a later section, along with other issues relating to xvoutines.

### G.12.4. Caution! Don't Let Continuous-Run Programs Leak!

Continuous-run programs are much more sensitive to memory leaks. Any memory leaks in your routine will accumulate on each iteration. If the routine runs for a long time, this will cause you to eventually run out of memory. The use of a memory tracking routine like *purify* to clean up all memory leaks is highly recommended.

## H. About Xvoutines

### H.1. Steps For Developing an Xvroutine Object

#### Select Toolbox For Xvroutine

Use *craftsman* to create a new toolbox, or to choose an existing toolbox in which to create the xvroutine (see Chapter 2 of the *Toolbox Programming Manual* for more details).

#### Create Xvroutine Program Object

Create the xvroutine object using *craftsman* (see Chapter 2 of the *Toolbox Programming Manual*).

#### Develop Xvroutine

Develop the xvroutine object using *composer* (see Chapter 3 of the *Toolbox Programming Manual*) or manually (% cd to the xvroutine program object directory).

1. Specify the command line arguments of the xvroutine in the \*.pane file; use *Guise* to interactively create the GUI (see Chapter 4 of the *Toolbox Programming Manual* for more information). After changing the \*.pane file, always regenerate code by selecting "Generate Code" from the Make menu of *composer* (see Chapter 3) or by running

```
% kmake regen
```

in the src/ directory of the xvroutine.

2. Specify the GUI of the xvroutine in the \*.form file use *Guise* to interactively create the GUI (see Chapter 4 of the *Toolbox Programming Manual* for more information). After changing the \*.form file, always regenerate code by selecting "Generate Code" from the Make menu of *composer* (see Chapter 3) or by running

```
% kmake regen
```

in the `src/` directory of the `xvroutine`.

3. Edit the code of the `xvroutine` to add functionality. Editing of `*.c` files can be done manually, or through `composer` (see Chapter 3 of the *Toolbox Programming Manual*).
4. Compile the `xvroutine`. Use `composer` to execute the compiler, or type

```
% kmake
```

See Chapter 3 of the *Toolbox Programming Manual* for more details on compiling from within `composer`.

5. Test and debug the `xvroutine`; iterate from step 3 as necessary.
6. Write the HTML page. After editing the HTML page, generate the man page and help page for the `xvroutine` by selecting "Generate Code" from the Make menu of `composer`, or by running

```
% kmake regen
```

in the `src/` directory of the `xvroutine`.

7. Write the online help pages; there must be one `*.doc` file for each help button on the GUI of the `xvroutine` as specified by the `*.form` file. The `*.doc` files must be created by hand. Verify that each help button on the GUI references the correct path to the appropriate `*.doc` help page.
8. For `xvroutine` maintenance and modification, repeat process from steps 1, 2, 3, 6, or 7 as necessary.

## H.2. The `*.form` UIS File

In addition to the standard `*.pane` file, `xvroutines` also have a UIS file named "`program.form`" (shorthand `*.form`) that describes their independent 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 its CLUI, thus prompting the creation of the `*.form` file.

`Guise` is used to create the `*.form` file for a program. Execute `guise` by double-clicking on the `*.form` file from the `composer` files list, or by running

```
% guise -tb toolbox -oname program -uis program.form
```



## H.2.1. Special Notes on \*.subform Files

For xv routines having a master form that can display multiple subforms, the \*.form file frequently becomes large and unwieldy. In this case, "\*.subform" files may be used if desired to modularize the UIS of the xv routine. The following is the procedure to be followed when creating \*.subform files.

### Creating \*.subform Files

Use **guise** to display each subform of the GUI in turn. Put each subform in edit mode. Note that if you only have a single pane on the subform, you may have to use your window manager to make the subform just a little bigger so that you are able to put the *subform* in edit mode, rather than the *pane*. Bring up the internal menuform for the subform. You can verify from the title of the menuform that it is indeed the menuform of the subform, and not the menuform of a pane. Use the selection at the bottom of the subform's menuform, entitled, "Write \*.subform file" to enter the name of the \*.subform file; hpress <Return> to write out the \*.subform file. A message will pop up, verifying that the \*.subform file was written out. Be sure to also write out the \*.form file itself before exiting **guise**. To verify that everything has been done properly, look at the \*.form file. It should contain one [-k] reference to each \*.subform file you created. If you like, you may use **preview** or **guise** to look at each subform separately.

A similar procedure applies if a subform with multiple panes is to be modularized into multiple \*.pane files.

## H.3. Files Generated For Xv routines

All the files created for kroutines are also created for xv routines (see previous section). Recall that these files include:

**The main driver (main.c)**

**The developer's driver (*program.c*, or \*.c)**

**The main include file (main.h)**

**The developer's include file (*program.h*, or \*.h)**

**The HTML page (*program.htm*, or \*.htm)**

**The man page (*program.1*, or \*.1)**

**The help page (*program.hlp*, or \*.hlp)**

Unless otherwise stated, the contents of these files is the same for xv routines as it is for kroutines, so that material will not be repeated again here.

In addition, xv routines have other files created to contain the GUI drivers and the code necessary to mediate between the application program and its GUI. Also, there will be files created for you to write the code that will react to user input to the GUI of the xv routine.

Accordingly, the additional files generated for an xv routine are as follows:

**The `form_program.c` file**

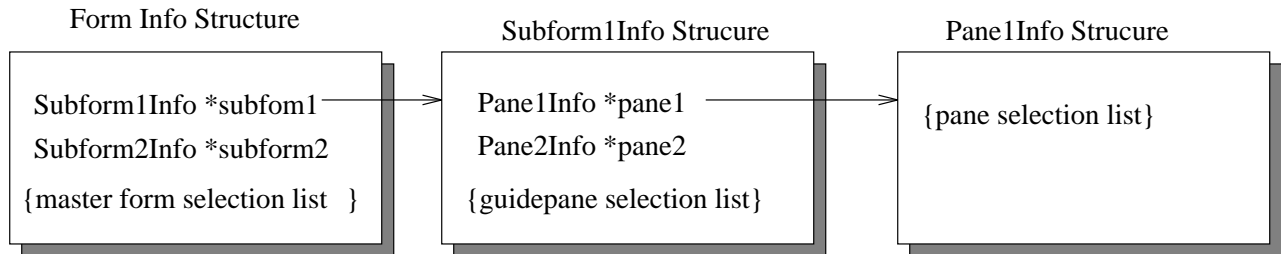
This file contains the GUI drivers for the xv routine and the code necessary to mediate between the application program and its GUI.

**Do files**

In addition to the "`form_program.c`" file, there will be one or more "do" files created for your xv routine. These are the files in which you add the functionality for your xv routine.

## H.4. The GUI Information Structure

The current values of the selections on the GUI of an xv routine are stored in the GUI Info structure. This structure is generated along with the CLUI Info structure (see previous sections) in the "`main.h`" file. The GUI Info structure is a tri-level data structure, reflecting the tri-level hierarchy of the VisiQuest GUI (master form / subforms / panes).



**Figure 9:** The organization of the tri-level GUI Info structure reflects the tri-level hierarchy of the VisiQuest GUI.

### H.4.1. The Form Info Structure

The highest structure in the GUI Info structure hierarchy is the Form Info structure. The actual (typedef'ed) name of the Form Info structure for your xv routine will be `xvroutine_formvar`. The Form Info structure will contain:

1. A set of variables corresponding to each subform on the form, including:

**forminfo->subformvar**

A pointer to the Subform Info structure corresponding to the subform. The Subform Info structure is discussed in the next section.

**forminfo->subformvar\_selected**

A logical for the subform which will be set to TRUE if that subform is currently receiving input from the user.

**forminfo->subformvar\_struct**

A generic *kform\_struct* representation of the subform, which can be passed to *xvf\_set\_attribute(s)()* if desired.

If the GUI has no master form, a single set of variables for the single subform on the GUI will be generated.

2. A set of selection information variables corresponding to each selection on the master form, *if and only if* the GUI has a master form, and *if and only if* that master form contains any selections. A table containing example selection information variables is given later.

**forminfo->selvar**

A variable of appropriate data type in which the current value of the selection is stored throughout the execution of the xvroutine.

**forminfo->selvar\_selected**

Generated when the selection is "live," this variable is set to TRUE when the selection has just been changed. It is used exclusively by the GUI subform driver to direct software flow to the *subformvar\_selvar()* routine in the "do\_subformvar.c" file.

**forminfo->selvar\_optsel**

Generated when the selection is optional, this variable is set to TRUE when the user highlights the optional button of the selection. It is for your use, so that you may detect when the user wants to use an optional selection.

**forminfo->selvar\_struct**

A generic representation of the selection, this selection variable may be passed to *xvf\_set\_attribute(s)()* in order to change the selection during runtime.

**forminfo->selvar\_label**

This variable is generated for those selections that have both an integer and a string value, such as cycles, lists, and logicals. It will hold the string value that corresponds to the current integer value.

## H.4.2. The Subform Info Structure

The middle structure in the hierarchy is the Subform Info structure. The actual (typedef'd) name of the Subform Info structure for your xvroutine will be *formvar\_subformvar*. The Subform Info structure will contain:

1. A set of variables corresponding to each pane on the subform, including:

**subforminfo->panevar**

A pointer to the Pane Info structures corresponding to the pane. The Pane Info structure is discussed in the next section.

**subforminfo->panevar\_selected**

A logical which will be set to TRUE if that pane is currently receiving input from the user.

**subforminfo->panevar\_struct**

A generic *kform\_struct* representation of the pane that can optionally be passed to *xvf\_set\_attribute(s)()*.

If the subform has no guidepane, a single set of variables for the single pane on the subform will be generated.

2. A set of variables corresponding to each selection on the guidepane, *if and only if* the subform has a guidepane, and *if and only if* that guidepane contains any selections. Assuming a "subforminfo" pointer, declared of type *formvar\_subformvar*, then variables are as follows:

**subforminfo->selvar**

A variable of appropriate data type in which the current value of the selection is stored throughout the execution of the xvroutine.

**subforminfo->selvar\_selected**

Generated when the selection is "live," this variable is set to TRUE when the selection has just been changed. It is used exclusively by the GUI subform driver to direct software flow to the *subformvar\_selvar()* routine in the "do\_subformvar.c" file.

**subforminfo->selvar\_optsel**

Generated when the selection is optional, this variable is set to TRUE when the user highlights the optional button of the selection. It is for your use, so that you may detect when the user wants to use an optional selection.

**subforminfo->selvar\_struct**

A generic representation of the selection, this selection variable may be passed to *xvf\_set\_attribute(s)()* in order to change the selection during runtime if desired.

**subforminfo->selvar\_label**

This variable is generated for those selections that have both an integer and a string value,

such as cycles, lists, and logicals. It will hold the string value that corresponds to the current integer value.

### H.4.3. The Pane Info Structure

The lowest structure in the hierarchy is the Pane Info structure. The actual (typedef'd) name of the Pane Info structure for your xvroutine will be *subformvar\_panevar*. For each selection on the pane, the Pane Info structure will contain the following set of information variables:

1. A variable of appropriate data type in which the current value of the selection is stored throughout the execution of the xvroutine.
2. Generated when the selection is "live," this variable is set to TRUE when the selection has just been changed. It is used exclusively by the GUI subform driver to direct software flow to the *subformvar\_selvar()* routine in the "do\_subformvar.c" file.
3. Generated when the selection is optional, this variable is set to TRUE when the user highlights the optional button of the selection. It is for your use, so that you may inquire when the user has indicated a desire to use an optional selection.
4. A generic representation of the selection, this variable may be passed to *xvf\_set\_attribute(s)()* in order to change the selection during runtime if desired.
5. This variable is generated for those selections that have both an integer and a string value, such as cycles, lists, and logicals. It will hold the string value that corresponds to the current integer value.

### H.4.4. The Selection Information Variables

Selection Info variables are generated as fields in the Form, Subform, and Pane Info structures, and directly correspond to selections on a master form, guide pane, or pane, respectively. Selection variables contain all the current information about the corresponding selection throughout the run of the program. The general types of selection variables were listed in each of the preceding three sections, in context of their appearance in the Form, Subform, and Pane Info structures, respectively. Some points worth reiterating, however, include:

1. The "*selvar\_selected*" variables are ONLY generated where noted if the selection in question is "live" or if it is an action button.
2. The "*selvar\_optsel*" variables are ONLY generated where noted if the selection in question has been specified as optional.

3. The "selvar\_struct" variables are generated for passing to `xvf_set_attribute(s)()`. These variables can also be passed to `xvf_add_extra_call()` if desired.

Example Selection Information Variables		
Selection (UIS Line Type)	Variable	Information Variables Generated For Selection w/ Variable Name Given
Action Button (-a), (-m), (-n)	reset	int reset; /* TRUE if user clicks on button */ kform_struct *reset_struct; /* pass to xvf_set_attribute() */
Quit Button (-Q)	n/a	int quit; /* TRUE if user clicks on quit */ kform_struct *quit_struct; /* pass to xvf_set_attribute() */
Input File (-I)	i1	char *i1; /* filename */ kform_struct *i1_struct; /* pass to xvf_set_attribute() */ int i1_selected; /* TRUE for <cr> or browser use */ int i1_optsel; /* TRUE if optional box is ON */
Output File (-O)	o2	char *o2; /* filename */ kform_struct *o2_struct; /* pass to xvf_set_attribute() */ int o2_selected; /* TRUE for <cr> */ int o2_optsel; /* TRUE if optional box is ON */
Integer (-i)	x	int x; /* integer value of x */ kform_struct *x_struct; /* pass to xvf_set_attribute() */ int x_selected; /* TRUE for <cr> or scrollbar motion */ int x_optsel; /* TRUE if optional box is ON */
Float (-f)	s	float s; /* float value of s */ kform_struct *s_struct; /* pass to xvf_set_attribute() */ int s_selected; /* TRUE for <cr> or scrollbar motion */ int s_optsel; /* TRUE if optional box is ON */
String (-s)	name	char *name; /* string value of name */ kform_struct *name_struct; /* pass to xvf_set_attribute() */ int name_selected; /* TRUE for <cr> */ int name_optsel; /* TRUE if optional box is ON */
StringList (-y)	color	char *color; /* string value of color */ kform_struct *color_struct; /* pass to xvf_set_attribute() */ int color_selected; /* TRUE for <cr> or list selection */ int color_optsel; /* TRUE if optional box is ON */
Logical (-l)	fill	int fill; /* boolean value of fill */ kform_struct *fill_struct; /* pass to xvf_set_attribute() */ int fill_selected; /* TRUE when user changes value */ int fill_optsel; /* TRUE if optional box is ON */ char *fill_label; /* string associated w/ boolean value */
Flag (-t)	rev	int rev; /* boolean value of rev */ kform_struct *rev_struct; /* pass to xvf_set_attribute() */ int rev_selected; /* TRUE when user clicks optional box */

Example Selection Information Variables		
Selection (UIS Line Type)	Variable	Information Variables Generated For Selection w/ Variable Name Given
Cycle (-c)	choice	<pre>int choice;           /* integer value of choice */ kform_struct *choice_struct; /* pass to xvf_set_attribute() */ int choice_selected; /* TRUE when user changes value */ int choice_optsel; /* TRUE if optional box is ON */ char *choice_label; /* string associated w/ int value */</pre>
List (-x)	mode	<pre>int mode;           /* integer value of mode */ kform_struct *mode_struct; /* pass to xvf_set_attribute() */ int mode_selected; /* TRUE when user changes value */ int mode_optsel; /* TRUE if optional box is ON */ char *mode_label; /* string associated w/ int value */</pre>
Workspace (-w)	canvas	<pre>xvobject canvas; /* address of canvas xvobject */ kform_struct *canvas_struct; /* pass to xvf_set_attribute() */</pre>
Blank (-b)	blk	<pre>kform_struct *blk_struct; /* pass to xvf_set_attribute() */</pre>
Toggle (-T)	type	<pre>int type_val; /* int value (flag, logical, int toggles) */ - or - char *type_val; /* string value (file or string toggles) */ - or - float type_val; /* float value (float toggles) */ int type_num; /* index of current value, 1 to N */ kform_struct *type_struct; /* pass to xvf_set_attribute() */ int type_selected; /* TRUE when user changes value */ int type_optsel; /* TRUE if toggle's optional box is ON */</pre>
MutExcl (-C)	N/A	No variables are generated for MutExcl line directly; however, variables are generated for each of the members in the Mutually Exclusive group.
MutIncl (-B)	N/A	No variables are generated for MutIncl line directly; however, variables are generated for each of the members in the Mutually Inclusive group.

---

**Table 2:** The table contains examples of the variables generated for the various GUI selections.

---

## H.5. The Main Driver (main.c)

The main program generated for an xvroutine differs in a variety of ways from the main program generated for a kroutine. Like the main program generated for a kroutine, the main program generated for an xvroutine calls *VisiQuest\_init()* to do general VisiQuest initializations, and calls *kclui\_init()* to do command line user interface initiations. As with kroutines, the main of an xvroutine calls "run\_program()", the routine that you will write to give the program its functionality.

In contrast to the main program for a kroutine, however, the main program for a xvroutine must perform a number of additional steps. It initializes GUI and Program Services libraries with a call to *xvw\_initialize()*. It finds and opens the \*.form file defining the GUI of your xvroutine, and creates the GUI with a call to *xvf\_build\_form()* using the full path to the \*.form file. It allocates the GUI Info structure for the xvroutine, and initializes the GUI Info structure with a call to the automatically-generated *\_xvf\_init\_program()* routine.

## H.6. The Developer's Driver (\*.c)

As mentioned before, the *program.c* file is the file containing the driver for your xvroutine that you will modify to give the program its functionality. The usage additions routine and the free args routine have already been covered, so that material will not be repeated here. However, there are additional elements in the run routine of an xvroutine, as well as an extra routine to which you must add code. This section covers these additional elements of the xvroutine developer's driver.

### The Run Routine

Whereas the run routine of a kroutine is generated as an empty routine, the run routine of an xvroutine has three function calls in it. You may add code to the run routine as desired, but new code should be placed at the top of the routine. Code to handle command line arguments will be placed in the *program\_init()* routine. The run routine of an xvroutine looks like the following:

```
int run_program_main(void)
{
    /*-- Put additional code here, at the top --*/

    /*-- enable automatic update handler to be fired on CLUI changes --*/
    xvw_add_clui_update(program_init, NULL);

    /*-- start up, given values provided on command line --*/
    program_init(NULL);

    /*-- run application --*/
    xvf_run_form();

    return TRUE;
}
```

Recall, from the earlier section on continuous run programs, that all xvroutines have the continuous-run capability. Xvroutines operate in an event loop which is driven by user input; software control flow is directed to the appropriate "do" routines according to the action taken by the user. When executed from VisiQuest, the xvroutine may already be running when the user opens up its glyph's pane, and change an argument. In this case, you want the xvroutine to properly update according to the new argument value. This goal is achieved by:

1. Installing a *clui update* routine, that will be called via the interprocess communication system (IPC) when the user changes any program arguments on the glyph pane in VisiQuest
2. Placing all code that handles program arguments (ie, all code that references the CLUI info structure) in the *clui update* routine that was installed.



Thus, the code above uses a call to `xvw_add_clui_update()` to install the `program_info()` as the CLUI update routine. It then calls `program_info()` directly, since the command line arguments will need to be processed for the first time in any case, and the xvroutine may or may not be executed from VisiQuest. Finally, the call to `xvf_run_form()` puts the xvroutine into the event loop that will display the graphical user interface defined by the \*.form file, and wait for user input.

### The CLUI Update Routine

The CLUI Update routine, called `program_info()`, is placed at the end of the `program.c` file. By isolating all references to the CLUI info structure in this routine, you ensure that your xvroutine will work properly as a continuous run program within VisiQuest. Any time the IPC detects a change in program arguments to the xvroutine, the CLUI update routine will be called, and re-initialize the xvroutine with the new argument values. For example, the following code is an excerpt from the `editimage` CLUI update routine:

```
void editimage_init(
    kaddr client_data)
{
    /*-- process [-i] argument --*/
    if (clui_info->i_file != NULL)
    {
        /*-- open data object associated with input file --*/
        if ((current_image = kpds_open_object(clui_info->i_file,
                                             KOBJ_READ)) == NULL)
        {
            kerror(NULL, routine, "Unable to open specified data file '%s'",
                   clui_info->i_file);
            return;
        }

        /*-- do everything associated with new image input --*/
        edimg_input_newimage();
    }
}
```

Note that the input file argument of the xvroutine is opened as a data object and displayed as an image using a utility routine. Since this is done within the CLUI update routine, the xvroutine will correctly update with a new incoming image if the user changes the input to the program. Similarly, code for all other CLUI arguments should also be added to this routine.

## H.7. The Main Include File (main.h)

In addition to the CLUI Info structure and other definitions generated in the "main.h" file that were covered in previous sections, the "main.h" file for an xvroutine contains the definition of the GUI Info structure (see previ-

ous sections).

## H.8. The Developer's Include File (\*.h)

As with routines, this \*.h file is generated for use by the developer. file. You may add code to this driver include file as needed.

## H.9. The *form\_program.c* File

The "*form\_program.c*" file contains all the additional generated code necessary for the xvroutine to operate as an event-driven application responding to user input to its GUI. This is an automatically generated file that you may not edit manually. It is updlts contents include the following:

- The GUI drivers.
- The code to store GUI values in the GUI Info structure.
- The code to initialize and free the GUI Info Structure.

The "*form\_program.c*" file contains your main GUI driver as well as subordinate GUI subdrivers in the GUI driver hierarchy. There will be a single main GUI driver named *run\_formvar()* that runs the GUI of your xvroutine as a whole. There will be one GUI subdriver for each subform on your GUI named *run\_subformvar()*, and one GUI subdriver for each pane on the subform named *run\_panevar()*. The sole purpose of the GUI drivers in the "*form\_drv.c*" file is to redirect software control flow to code in which you will provide the functionality for your xvroutine. This is accomplished by the GUI subdrivers making appropriate calls to the *do* routines (explained in the following section) in which you will do the "real" coding for the application.

Also, the "*form\_program.c*" file also contains the C code which will extract information from the GUI of your xvroutine and store it in the GUI Info structure so that you may access the information as needed. You need not be too worried about understanding this code, which will be a series of extraction routines that reflect the tri-level hierarchy of the VisiQuest GUI. The first routine extracts information from the GUI as a whole and is named *\_xvf\_get\_formvar()*. Following that, there will be one routine for extracting information from each subform on your GUI named *\_xvf\_get\_subformvar()*. A pane extraction routine will be generated for each pane on each subform named *\_xvf\_get\_panevar()*.

The "*form\_program.c*" file contains the C code which will initialize the GUI Info structure according to the default values that were specified in the \*.form file. The initialization routine generated will be named *\_xvf\_init\_formvar()*.

Finally, the "form\_program.c" file contains a routine code to free the GUI Info structure before the xvroutine exists. This cleanup routine will be named `_xvf_free_formvar()`.

## H.10. Do Files

*Do* files are generated to contain the "do" routines in which you will give your xvroutine its functionality. A *do* routine is generated for each action button and "live" selection of your GUI. Skeletons for these routines are created initially; the *do* files will never be touched again unless you add new action buttons and/or "live" selections to your \*.form file, in which case new *do* routine skeletons will be appended to the end of the appropriate *do* file. Remember that the *xvforms* library passes software control to your xvroutine *only* when the user clicks on an action button or changes the value of a "live" selection.

The *do* routines are found in appropriately named files, and organized according to the location of the corresponding action buttons and live selections whether they appear on a pane, on the guidepane of a subform, or on a master form. A "do\_panevar.c" file is created for the *do* routines that correspond to the action buttons and "live" selections on a pane. A "do\_subformvar.c" file is created for each guidepane with action buttons and/or live selections. A "do\_formvar.c" file is created for a master form containing action buttons or live selections. Collectively, these files are referred to as *do* files.

An xvroutine with a simple uni-level GUI consisting of a single pane on a single subform would only have one *do* file. A more complex GUI involving a master form which brings up subforms with guidepanes will have many *do* files. Since *do* routines are *only* generated for action buttons and "live" selections, an xvroutine having a GUI without any action buttons or "live" selections would have no *do* files at all, regardless of its size and complexity. Note that since action buttons and "live" selections provide the only mechanism<sup>2</sup> through which the *xvforms* library passes software control to the xvroutine, an xvroutine with NO action buttons or "live" selections could have no functionality added to it.

Calls to the subroutines whose skeletons appear in the *do* files are generated in the GUI drivers. Because calls to these subroutines are generated automatically, the subroutines must have predefined names derived from the variable names on the UIS lines of the \*.form file; you may *not* change the names of the subroutines. Subroutine names have two parts. The first part of the name indicates the location of the selection within the GUI, on the master form, subform, or pane. The second part of the name is taken from the variable name on the UIS line defining the GUI selection itself. Thus, functions corresponding to GUI selections on a pane will be named *panevar\_selvar()*, functions corresponding to GUI selections on the guidepane of a subform will be named *subformvar\_selvar()*, and functions corresponding to GUI selections on a master form will be named *formvar\_selvar()*.

To illustrate, suppose we have an action button on a pane; assume the action button is defined by a [-a] UIS line with the variable name *redisplay*, and the pane is defined by a [-P] UIS line with the variable name *plot*. In this case, a file named "do\_plot.c" will be generated, containing a skeleton for a subroutine named *plot\_redisplay()*. The *plot\_redisplay()* subroutine would be called by the *run\_plot()* GUI subdriver in the "form\_drv.c" file. Again, remember that this code is all generated automatically for you in the GUI drivers;

---

<sup>2</sup> An exception can be made with the use of *xvf\_add\_extra\_call()*; see the *xvforms* section of the *Program Services III Manual* for more details.

you only need to add the appropriate functionality to the *do* routines themselves.

## H.11. Example of Files & Functions Generated



**Figure 10:** Above is a very simple GUI for a "superplot" program. The GUI for "superplot" contains a workspace, an action button, a help button, and a quit button. Below is the UIS file that describes the "superplot" GUI.

```
-F 4.3 1 1 10x5+10+15 +0+0 ' ' plot
-M 1 1 20x5+0+0 +14+0.5 'Super Plot Package' display
-P 1 0 53x7.68182+0+0 +2+0 'Linear Plots' linear
-H 1 5x1.5+33+0.2 'HELP' 'help page for superplot' $TOOLBOX/help/superplot help
-Q 1 0 5x1.5+42+0.2 'QUIT'
-w 410x90+1+2 ' ' 'contains'
-a 1 0 8x1.5+1+6 'Reset' 'resets the axes' reset
-E
-E
-E
```

Example Files and Functions			
UIS Line	Variable	File	Function Generated {or other effect of variable on contents of file}

Example Files and Functions			
UIS Line	Variable	File	Function Generated {or other effect of variable on contents of file}
StartForm (-F)	<i>plot</i>	form_init.c  form_info.c  form_drv.c	<i>_xvf_init_plot()</i> : Initializes the "superplot_plot" GUI Info structure.  <i>_xvf_get_plot()</i> : Main GUI info extraction routine; calls <i>_xvf_get_display()</i> for [-1 1]  <i>run_plot()</i> : Main GUI driver for "superplot"; calls <i>_xvf_get_plot()</i> & <i>run_display()</i>
StartSubform (-M)	<i>display</i>	form_init.c  form_info.c  form_drv.c	{code to extract values on <i>display</i> subform in <i>_xvf_init_plot()</i> }  <i>_xvf_get_display()</i> Extracts GUI info from "display" subform; calls <i>_xvf_get_linear()</i> for [-1 (1, 2) ]  <i>_xvf_run_display()</i> GUI subdriver for "display" subform; calls <i>run_linear()</i> .
StartPane (-P)	<i>linear</i>	form_init.c  form_info.c  form_drv.c  do_linear.c	{code to extract values on <i>linear</i> pane in <i>_xvf_init_plot()</i> }  <i>_xvf_get_linear()</i> Extracts GUI info from <i>linear</i> pane.  <i>run_linear()</i> GUI subdriver for "linear" pane;  calls <i>linear_reset()</i> {no effect on code; simply determines filename}
PaneAction (-a)	<i>reset</i>	form_init.c  form_info.c  form_drv.c  do_linear.c	{code to initialize integer "reset" variable in the Pane Info structure to FALSE}  {code to set integer "reset" variable in the Pane Info structure to TRUE when user clicks on action button}  {produces a call to <i>linear_reset()</i> in <i>run_linear()</i> }  <i>linear_reset()</i> You add your code to reset the axes here.

**Table 3:** The table contains examples of the files and the functions that are generated by conductor. If the UIS file above was provided as the \*.form file for the "superplot" program.

---

## H.12. Xvroutine Documentation

Like kroutines, xvroutines have an HTML page which should be filled out and used to generate a man page and an online help page. See the relevant sections earlier in this chapter for details. Xvroutines also have additional documentation opportunities that the developer won't want to miss!

For an xvroutine, there should be a *Help* button on each pane defined in the \*.form file. For GUI's with multiple subforms, there should be a *Help* button on each subform. For GUI's with master forms, there should also be a *Help* button on the master form. The *Help* buttons should each reference a \*.doc file, which you create from scratch in the *help/* directory. There is a set of specialized documentation macros that are used in place of *roff* commands in \*.doc files. These macros were created especially for use with VisiQuest documentation and are significantly easier to use than *roff* commands. A summary of those commands is available in Appendix B, *Documentation Macros*.

## I. About Pane Objects

### I.1. Steps For Developing a Pane Object

#### Select Toolbox For Pane Object

Use *craftsman* to create a new toolbox, or to choose an existing toolbox in which to create the pane object (see Chapter 2 of the Toolbox Programming Manual for more details).

#### Create Pane Object

Create the pane object using *craftsman* (see Chapter 2 of the Toolbox Programming Manual).

#### Develop Pane Object

Develop the pane object using *composer* (see Chapter 3 of the Toolbox Programming Manual) or manually (% *cd* to the pane object directory).

1. Specify the command line arguments of the pane object in the \*.pane file; use *Guise* to interactively create the user interface. See Chapter 4 of the Toolbox Programming Manual for more information on *guise*.
2. After changing the \*.pane file, always regenerate code by using the "Generate Code" button of *composer* (see Chapter 3) or by running

```
% kmake regen
```

in the *src/* directory of the pane object.

3. Compile the pane object script. Use **composer** to execute the compiler, or type

```
% kmake install
```

in the *src* directory of the pane object. See Chapter 3 of the Toolbox Programming Manual for more details on compiling from within **composer**.

4. Write HTML page for pane object carefully. After editing the HTML page, generate the man and help pages for the pane object by clicking on the "Generate Code" button of **composer**, or by running

```
% kmake regen
```

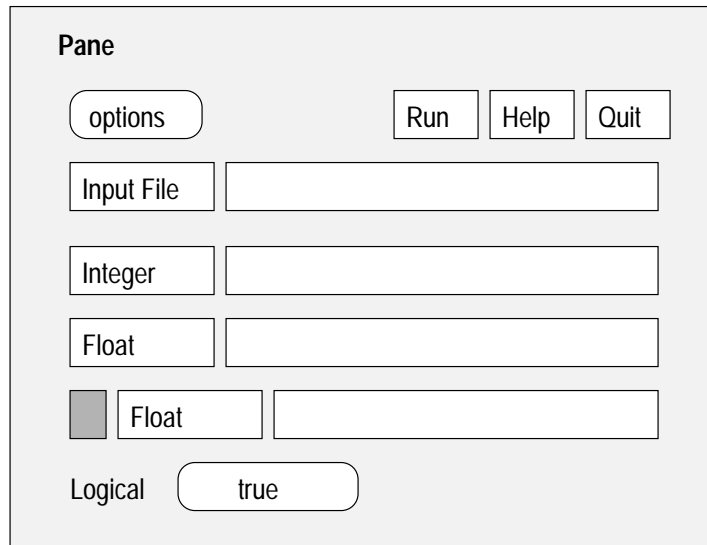
in the *src/* directory of the pane object.

5. For pane object maintenance and modification, repeat process from steps 1 or 4 as necessary.

## I.2. \*.pane File

The *pane object* depends on another software object (a kroutine or an xvroutine) for its functionality; the program that provides the functionality for the pane object is referred to as the *base program* of the pane object. Because the point of the pane object is to present the user with a simplified interface to a particular aspect of the base program's functionality, the most significant component of the pane object is its GUI, as specified by its \*.pane file.

Like the \*.pane files of kroutines and xvoutines, the \*.pane file of a pane object is used as input to the code generator. However, the code generator does not generate any C code for a pane object. Instead, it generates a simple shell script that uses **kexec** to display the GUI of the pane object. When the user clicks on the *Run* button of the pane, **kexec** will execute the base program of the pane object with the arguments specified on the pane object's GUI.




---

**Figure 11:** Like the \*.panes for kroutines and xvoutines, the \*.pane file for a pane object defines a GUI that consists of a single pane on a single subform.

---

As in the panes for kroutines and xvoutine, the *Options* pulldown menu, Run, Help and Quit buttons are standard items included on GUI's of pane objects. The selections on the pane correspond to a *subset* of the arguments of the program that provides the functionality for the pane object. When creating a \*.pane file for a pane object, special attention should be paid to the following items:

### The Title of the GUI

The title that appears on the the pane object's GUI indicates the aspect of the functionality of the base program that is provided by the pane object. For example, the *kabs* pane object provides a simplified user interface to the multiple-operation *karith1* kroutine. While the title of the pane for *karith1* reflects the general nature of *karith1* ("Single Operator Pointwise Arithmetic"), the title of the *kabs* pane object can be much more specific ("Output = Absolute Value of Input"). Note that the title of the GUI is specified for the *subform* of the pane object's GUI; i.e., the title appears on the [-M] UIS line of the \*.pane file. When using *guise*, the title is specified using the internal menuform of the single subform, *not* the single name on the subform. Alternatively, you may edit the \*.pane file manually, entering the desired title between the tic marks on the [-M] StartSubform UIS line.

### The base program

The base program specification is done in association with the "Run" button. The base program must be specified using:

```
$TOOLBOXBIN/base_program [-flag(s)]
```

For example, the base program for *kabs* is specified as follows:

```
$DATAMANIPBIN/karith1 -abs
```

When using *Guise*, bring up the internal menuform for the *Run* button, and enter the base program *with the appropriate command line argument(s)* in the *Program* selection. Alternatively, you may directly edit the \*.pane file, putting the desired base program specification as the last field in the [-R] UIS line (which specifies the *Run* button).



### Other selections on the pane

All selections specified in the \*.pane file for the pane object *must also appear in the \*.pane file for the base program*. Remember that the pane object serves only as a "front end" to the base program. As such, it cannot have any selections on its GUI that will not be recognized by the base program. Titles, of course, may be changed if desired; variables used, however, must match those of the base program.

## I.3. Pane Object Documentation

The documentation files for pane objects are the same as those for kroutines, and work the same way. Rather than repeating the explanation here, we refer you to the section on "Program Documentation" for kroutines.

## I.4. Generated Shell Script File

The code generator creates a script associated with the pane object. After several lines of comments, there is only one line of code in the script:

```
exec kexec -tb TOOLBOX -oname pane_object_name -args "$*
```

This line invokes `kexec` with the object name of the pane object; `kexec`, in turn, will execute the base program of the pane object with the arguments specified for the pane object. You *must not* modify the script as it is regenerated each time the code generator is run.

## J. About Script Objects

### J.1. Steps For Developing a Script Object

#### Select Toolbox For Script Object

Use `craftsman` to create a new toolbox, or to choose an existing toolbox in which to create the script object (see Chapter 2 of the Toolbox Programming Manual for more details).

#### Create Script Object

Create the script object using `craftsman` (see Chapter 2 of the Toolbox Programming Manual). Select the scripting language that you will be using; `csh`, `sh`, `ksh`, and `perl` are supported. If you will be using a \*.pane file to specify a GUI for the script object so that it may be accessed via VisiQuest, set the logical selection "Command-line UIS file?" to "Yes." Note that inclusion of the script object in VisiQuest is the *only* reason to use a \*.pane file.

## Develop Script Object

Develop the script object using `composer` (see Chapter 3 of the Toolbox Programming Manual) or manually (`% cd` to the script object directory).

1. Write the shell script in the scripting language specified when the script object was created. All command line options must be parsed by the script; there is no automatic generation of command line parsing code for script objects.
2. If you are using a \*.pane file so that the script may be integrated into VisiQuest, specify the command line arguments of the script object in the \*.pane file such that they are *compatible* with the ones parsed by the script. You may use `Guise` to interactively create the GUI. See Chapter 4 of the Toolbox Programming Manual for more information on `Guise`.
3. Write HTML page for script object carefully.
4. If you are using a \*.pane file, after editing the HTML page you should generate the help and man pages for the script object by clicking on the "Generate Code" button of `composer`, or by running

```
    % kmake regen
```

in the `src/` directory of the script object.
5. For script object maintenance and modification, repeat process from step 1 as necessary.

## J.2. Shell Script File

## J.3. \*.pane File

As mentioned earlier, \*.pane files are only used with script objects in order to include the script object in VisiQuest. Since the command line parsing code cannot be automatically generated, it is important that the arguments specified by the \*.pane file match those parsed by the shell script itself.

## J.4. Script Object Documentation

The documentation files for script objects are the same as those for kroutines, and work the same way. Rather than repeating the explanation here, we refer you to the section on "Program Documentation" for kroutines.

## K. Ansi C/C++ Software Objects

VisiQuest allows you to include non-VisiQuest based libraries and programs within a VisiQuest toolbox. This is done using Ansi C/C++ software objects. By creating Ansi C/C++ program objects and library objects, you may create programs in C or C++ which are managed by the VisiQuest development tools, but do not actually depend on the VisiQuest software development libraries. This is especially useful for managing legacy source code without requiring undue modification of the code. It also offers a way to integrate of legacy code with the VisiQuest Visual Programming Language, VisiQuest.

### K.1. The Ansi/C++ Standalone Program object

The Ansi/C++ Standalone Program object is similar to a VisiQuest kroutine, except that it contains no calls to VisiQuest within its code, which may be written either in C or C++. If the program is to be accessible as a glyph within VisiQuest, a VisiQuest command line user interface specification (UIS) must be created for it. Unlike normal VisiQuest kroutines, no code to parse the command line will be generated; it is up to the programmer to write the code that will parse the command line options supported by the UIS file.

#### K.1.1. Steps for developing an Ansi/C++ Standalone Program object

##### Select Toolbox For Standalone Program Object

Use `craftsman` to create a new toolbox, or to choose an existing toolbox in which to create the script object (see Chapter 2 of the Toolbox Programming Manual for more details).

##### Create Standalone Program Object

Create the standalone program object using `craftsman` (see Chapter 2 of the Toolbox Programming Manual). Attributes of the standalone software object include the Toolbox, Object Name, Binary Name, Author, Email Address, Short Description, Install in VisiQuest, Icon Name, Category, and Subcategory.

If you want the program to be accessible as a glyph in VisiQuest, set the logical selections "Command-line UIS file?" to "Yes", and set "Install in VisiQuest" to True. Also be sure to specify an Icon Name, Category, and Subcategory. These values determine where in the VisiQuest Glyph menus the glyph will appear, and what its name will be.

The Short description is a one line description of the purpose of the program, it defaults to "No Short Description Specified" if none is specified.

Author and email address are strings to give proper credit and access to the author of the program. These value will inherit the Toolbox values if no Author or email address are specified.

Binary Name defaults to the Object Name is none is specified.

The Ansi/C++ code generator will create the kcm database file, a template html man page, and a Pmakefile. It also creates a man and help page which are just versions of the html document, formatted differently.

### **Develop Standalone program Object**

Develop the standalone program object using `composer` (see Chapter 3 of the Toolbox Programming Manual) or manually (`% cd` to the script object directory).

1. The code for the standalone program object may be written in C or C++. If desired, legacy code may simply be copied into the `src/` directory of the standalone program object if desired; slight modifications may be necessary in order to get the program to compile.
2. If the program is to be integrated into `VisiQuest`, use `Guise` to specify the arguments of the program in the `*.pane` file. See Chapter 4 of the Toolbox Programming Manual for more information on `Guise`. Remember that all arguments must be parsed by the program; there is no automatic generation of command line parsing code for standalone objects.
3. Write HTML page for the standalone program object carefully. Be aware that the html page has special tagged sections which will be filled out each time the code generator is called. The `begin_arguments` and `end_arguments` section is filled out based on the values in the UIS file. If no UIS file exists for an Ansi software object, this section will be left blank. The `begin_short_copyright` and `end_short_copyright` section is filled out with the copyright of the toolbox.

*never* edit the man or help page files, as they are completely auto-generated from the html page. After editing the HTML page, re-generate the help and man pages for the standalone object by clicking on the "Generate Code" button of `composer`, or by running

```
% kmake regen
```

in the `src/` directory of the standalone object.

4. For standalone program object maintenance and modification, repeat process from step 1 as necessary.

## **K.2. Ansi/C++ Standalone Library object**

The Ansi/C++ Standalone Library object allows you to include a non-VisiQuest library within a VisiQuest toolbox. The library may be written in C or C++, and it may be new code or legacy code being integrated using VisiQuest.

## K.2.1. Steps for developing an Ansi/C++ Standalone Library object

### Select Toolbox for Library

Use `craftsman` to create a new toolbox, or to choose an existing toolbox in which to create the library (see Chapter 2 of the Toolbox Programming Manual for more details).

When creating a standalone library object, you must provide a Toolbox Name and an Object Name. Other options include: Archive Name, Author, Email Address, Short Description, Language Dependency, and Add reference to `toolbox.mk`. The Short description is a one line description of the purpose of the library. Author and email address are strings to give proper credit and access to the author of the program. These value will inherit the Toolbox values if no Author or email address are specified. Binary Name defaults to the Object Name is none is specified. The Language Dependency selection allows you to tell the code generator whether or not this is a library with C or C++ dependencies. The Add reference to `toolbox.mk` file determines whether or not the `toolbox.h` file is updated to have a `#include` to the library's `oname.h` and the `toolbox.mk` file is updated to include the library in program linking.

### Create Library Object

Create the library object using `craftsman` (see Chapter 2 of the Toolbox Programming Manual).

### Develop Library

Develop the library using `composer` (see Chapter 3 of the Toolbox Programming Manual) or manually (`% cd` to the library object directory).

1. Add files containing public library routines to the library using `composer` (see Chapter 3 of the Toolbox Programming Manual).
2. Edit code in public library routines. Editing of `*.c` files can be done manually, or through `composer` (see Chapter 3 of the Toolbox Programming Manual). Private and static library routines may also be added to augment the public library routines.
3. Prototype any public library routines in the public library include file. Prototype all private library routines in the "internals.h" file. Prototype any static library routines at the top of the file they appear in. Editing the `*.h` files can be done manually, or with `composer`.
4. Compile the library routines. Use `composer` to execute the compiler, or type:  

```
% kmake
```

in the `src` directory of the library. See Chapter 3 of the Toolbox Programming Manual for more details on compiling from within `composer`.
5. Test and debug library routines; iterate from step 4 as necessary.
6. Fill in library headers carefully.

The Standalone library code generator scans the source code and header files of the library for public functions. Public functions are marked by uniquely formatted C style comment headers. The following is a sample header containing all the fields understood by the code generator.

```

/*****
*
* Routine Name: function_name - short description of function
*
* Purpose: This should be a complete description that anyone
*         could understand; it should have acceptable grammar
*         and correct spelling.
*
* Input: argument1 - explanation
*        argument2 - explanation
*        argument3 - explanation
*
* Output: argument4 - explanation
*         argument5 - explanation
*
* Returns: TRUE (1) on success, FALSE (0) otherwise
*
* Restrictions: Restrictions on data or input as applicable
* Written By: John Doe
* Date: Nov 15, 1994
* Verified: How to verify the routine
* Side Effects: Any side effects caused by calling the routine
* Examples: An explanation on how to use the routine
*           ! Actual code here showing it being used.
*           ! '!' preserve spacing for pre-formatted sections
* See Also: References to other related documents.
* Declaration: ! int function_name(
*              ! char *argument1,
*              ! char *argument2,
*              ! char *argument3,
*              ! char **argument4,
*              ! int *argument5)
*****/
int function_name(
    char *argument1,
    char *argument2,
    char *argument3,
    char **argument4,
    int *argument5)
{
    /* put your code here */
}

```

These headers are parsed by the code generators, and turned into 100% auto-generated html and man 3 pages. The only field above that is required is the *Routine Name*: field. All others can be included or not, and can be in any order. The *Declaration*: field is **required** when

you are putting a public header on a macro. On C/C++ routines, if the header does not include a Declaration field the declaration of the routine itself is used.

All fields above, except *Written By*: result in sections in the html and man pages for the library. As noted above, lines beginning with the '!' symbol preserve all space after the '!'.

7. The html documentation file is the primary file which documents your software object. The man file is a direct roff conversion of the html document. You should **never** edit the man page file, as it is 100% auto-generated from the html page. The only other thing to be aware of is the html page has special tagged sections which will be filled out each time the code generator is called. The *begin\_function\_list* and *end\_function\_list* section is filled out based on the public functions that exist in the library. If no public functions exist, this section will be empty. The *begin\_short\_copyright* and *end\_short\_copyright* section is filled out with the copyright of the toolbox. After editing library headers, generate man pages for the public library routines by clicking on the "Generate Code" button of **composer**, or by running

```
% kmake regen
```

in the *src* directory of the library.

8. For library routine maintenance and modification, repeat process as necessary.

# Table of Contents

A. Introduction	5-1
B. Review Of Toolbox Objects	5-1
C. Issues Specific to Toolbox Objects	5-2
C.1. Steps For Developing a Toolbox Object	5-3
C.2. The ToolboxInfo File	5-3
C.3. The Aliases File	5-4
C.4. The Public Toolbox Include File	5-5
C.4.1. How Include File Dependencies Work	5-6
C.4.2. Syntax and Contents of the Public Toolbox Include File	5-7
C.5. Toolbox Dependencies	5-8
C.6. Copyrights	5-10
D. Review Of Software Objects	5-10
E. Common Issues	5-11
E.1. The Graphical User Interface (GUI)	5-11
E.2. The Command Line User Interface (CLUI)	5-15
E.3. The User Interface Specification (UIS) file	5-15
E.4. Automatically Generated Code and Documentation	5-16
E.4.1. Code generation	5-17
E.4.2. Documentation generation	5-18
E.5. Prototyping of Subroutines & Functions	5-18
E.6. Software Object Database Files	5-19
E.7. The VisiQuest Data File (KDF) Format	5-19
E.7.1. KDF Header	5-20
E.7.2. Attribute Blocks	5-20
E.7.3. Attributes	5-22
E.7.4. Segment Data	5-23
E.8. File Naming Conventions	5-23
E.9. Argument Naming Conventions	5-24
E.10. Tips for Writing Successful Software	5-27
F. About Library Objects	5-29
F.1. Steps For Developing a Library Object	5-29
F.2. Library Routine Types: Public, Private, Static	5-30
F.3. Standard Elements of A Library Source Code File	5-31
F.3.1. The Public Library Routine Header	5-32
F.3.2. The Private/Static Library Routine Header	5-34
F.4. Documentation For Libraries	5-35
F.4.1. Documentation for Public Library Routines	5-35
F.4.2. Documentation for the Library as a Whole	5-35
F.5. Include Files For Libraries	5-37
F.5.1. The Library's Private Include File	5-39
F.5.2. The Library's Public Include File	5-41
G. About Kroutines	5-43
G.1. Steps For Developing a Kroutine Object	5-43
G.2. The *.pane UIS File	5-44
G.3. Files Generated	5-45
G.4. The CLUI Information Structure	5-46
G.5. The Main Driver (main.c)	5-48



G.6. The Developer's Driver (*.c)	5-49
G.7. The Main Include File (main.h)	5-50
G.8. The Developer's Include File (*.h)	5-50
G.9. Program Documentation (*.htm, *.1, *.hlp)	5-50
G.10. Program Structure for Kroutines	5-52
G.11. Making kroutine functionality re-usable with an lkroutine	5-52
G.11.1. The kroutine driver	5-53
G.11.2. Set Up Parameters	5-53
G.11.3. Call the Lkroutine	5-54
G.11.4. Check Status Returned By Lkroutine	5-54
G.11.5. The Kroutine Driver Return Status	5-54
G.11.6. Example Kroutine Driver Code	5-54
G.11.7. Return Status	5-55
G.11.8. Calling Other Functions	5-55
G.11.9. Error Messages and Information	5-56
G.11.10. Side Effects	5-56
G.11.11. Example Lkroutine	5-56
G.12. Continuous Run Programs	5-58
G.12.1. Introduction to Continuous-Run Programs	5-58
G.12.2. How Continuous-Run Programs Work	5-58
G.12.3. Creating Continuous-Run Programs	5-59
G.12.4. Caution! Don't Let Continuous-Run Programs Leak!	5-60
H. About Xvroutines	5-60
H.1. Steps For Developing an Xvroutine Object	5-60
H.2. The *.form UIS File	5-61
H.2.1. Special Notes on *.subform Files	5-62
H.3. Files Generated For Xvroutines	5-62
H.4. The GUI Information Structure	5-63
H.4.1. The Form Info Structure	5-63
H.4.2. The Subform Info Structure	5-65
H.4.3. The Pane Info Structure	5-66
H.4.4. The Selection Information Variables	5-66
H.5. The Main Driver (main.c)	5-68
H.6. The Developer's Driver (*.c)	5-69
H.7. The Main Include File (main.h)	5-70
H.8. The Developer's Include File (*.h)	5-71
H.9. The form_program.c File	5-71
H.10. Do Files	5-72
H.11. Example of Files & Functions Generated	5-73
H.12. Xvroutine Documentation	5-75
I. About Pane Objects	5-75
I.1. Steps For Developing a Pane Object	5-75
I.2. *.pane File	5-76
I.3. Pane Object Documentation	5-78
I.4. Generated Shell Script File	5-78
J. About Script Objects	5-78
J.1. Steps For Developing a Script Object	5-78
J.2. Shell Script File	5-79
J.3. *.pane File	5-79
J.4. Script Object Documentation	5-79

K. Ansi C/C++ Software Objects . . . . .	5-80
K.1. The Ansi/C++ Standalone Program object . . . . .	5-80
K.1.1. Steps for developing an Ansi/C++ Standalone Program object . . . . .	5-80
K.2. Ansi/C++ Standalone Library object . . . . .	5-81
K.2.1. Steps for developing an Ansi/C++ Standalone Library object . . . . .	5-82

**This page left intentionally blank**

**Chapter 6**

**The Compile System**



# Chapter 6 - The Compile System

## A. Introduction

The `kmake` system, provided as part of the *bootstrap* toolbox, is the mechanism that is used for compiling software in the VisiQuest software development environment. VisiQuest is designed to run on a wide variety of platforms, and provides a single approach to makefiles which provides architecture independence for VisiQuest toolboxes.

The system has three parts: the toolbox configuration file, *toolbox.mk*, which defines the rules relating toolbox dependencies and install directories that `kmake` uses to compile the software objects; *rules.mk*, files which define the make targets for the software objects; and architecture-independent *Pmakefiles*, which provide object specific values to the rules.

The `toolbox.mk` file keeps track of the binary install directory, library install directory, libraries in the toolbox to link against, an include directory search path for the toolbox, a library directory search path for the toolbox, and it includes the `toolbox.mk` files for toolboxes the toolbox depends on. The `toolbox.mk` file also conditionally includes a Site specific files which can exist in the `$TOOLBOX/config` directory. This Site file allows special initialization and rules to exist for the toolbox. The `rules.mk` files are unique to each software object classtype, and provide the rules understood when `kmake` is invoked. The `Pmakefiles` contain flags and variables that are used in the rules defined in the `rules.mk` and by conditional statements defined in the "toolbox.mk" file, determining how the software is to be compiled by `kmake`. The `kgenmake` program automatically generates *Pmakefiles* specific to the classtype of the software object it is operating on.

`Kmake` is a modified version of the UC Berkeley's *pmake* program. Since the subject is so extensive, and well documented in many commercial make books, we will not describe `pmake` syntax here. The `pmake` specific modifications to make are described in the paper, "PMake -- A Tutorial" by Adam De Boor..

VisiQuest provides three extension to the `pmake` syntax. These extensions (1) support a prepend variable syntax and (2) a define variable syntax. An example of the prepend syntax is as follows:

```
INCLUDES ^= -I${VSIP}/include
```

The above example inserts `-I${VSIP}/include` to the beginning of the list of values in the `INCLUDES` variable.

An example of the define variable syntax is:

```
.def COMPILE_OPTIMIZED
```

The above example is equivalent to

```
COMPILE_OPTIMIZED = 1
```

VisiQuest also adds a `[-Ovariable]` option to `kmake` that prevents *variable* specified from being set. This is useful when a variable is used as a flag to turn certain make options on or off, as with "kmake debug" or "kmake optimize" (see section on "Kmake Targets").

## B. Kmake Targets

The `kmake` command is used to compile a software object. A current Pmakefile, created with `kgenmake`, must exist in the directory in which `kmake` is executed. If you are in doubt as to the validity of a Pmakefile at any time, running `kgenmake` can never hurt the compile process.

There are a number of compile targets, and the specific targets vary from classtype to classtype. The necessary targets for a specific classtype are available from the composer "Make" menu. Listed below are several of the more common make targets:

```
% kmake
% kmake clean
% kmake debug
% kmake depend
% kmake install
% kmake klint
% kmake lint
% kmake optimize
% kmake proto
% kmake protoize
% kmake purecov
% kmake purify
% kmake purtest
% kmake quantify
% kmake regen
% kmake struct
% kmake tags
% kmake test
% kmake uninstall
% kmake Makefile
% kmake Makefiles
```

### Most Commonly Used Options

The most commonly used `kmake` commands are `kmake install`, `kmake clean`, `make depend`, `kmake debug`, `kmake optimize`, `kmake regen`, and of course, just plain `make`.

#### **% kmake and kmake all**

Compiles the software object according to the rules included in the local Pmakefile. The executable or library archive is left in the local, or ".", directory.

#### **% kmake install**

Compiles the software object according to the rules included by the Pmakefile, and then installs the executable result in the location specified by `LIBDIR` (if you are compiling a library object) or `BINDIR` (if you are compiling a program object). Both `LIBDIR` and `BINDIR` are specified in your `toolbox/repos/config/mk/toolbox.mk` file. Note that `BINDIR` and `LIBDIR` are typically specific to each toolbox.

### **% kmake clean**

Cleans out a directory; it is done (1) to save space, or (2) in preparation for compiling on a new architecture. The `% make clean` procedure will remove all object files (\*.o), core files, archived library files (\*.a), RCS files (\*,\*) tilde files produced by emacs (\*~), files produced by purify (\*.pure), and other such files.

### **% kmake regen**

Executes the appropriate code generator on the software object in question, thus recreating automatically generated source code and documentation for the software object. It is equivalent to running `kregenobj` in the source directory of the software object. The advantage of `% kmake regen` over a direct execution of `kregenobj` is that it used to recursively descend into subdirectories from the `objects` or `objects/{object type}` directory of a toolbox, in addition to being done directly in the directory associated with the software object itself. In this way, multiple software objects may have their code and documentation updated automatically. Note that the `[-force]` option is used when `kregenobj` is executed in this way, so you will *not* be prompted before files are over-written.

## **Other General Kmake Options**

### **% kmake depend**

This updates the ".depend" file in your `src/` directory according to any changes that may have been made in include file status. Any time you touch a \*.h file on which your software depends, the entire software object is recompiled from scratch. This is to ensure that a software object binary cannot become corrupted because of changes in data structures, #defines, and other items that affect memory. Include files in question are not only those directly included in the code with "#include" statements, but also those include files on which libraries used by the software object depend; thus, the include file dependancy rule can be considered transitive (if A depends on B, and B depends on C, then A also depends on C). `% kmake depend` should be used any time a "#include" reference to a new \*.h file is made for the first time in a software object, or any time a \*.h file referenced by a "#include" statement is renamed or moved. Thus, after the .depend file has been properly updated, the software object will be forced to recompile when the new (or moved) include file is touched. Because of the transitive nature of the include file dependancy rule, if it is a *library* which has had its include file status changed, not only does the library have to have its .depend files updated, but any program objects using that library must have their .depend files updated, too.

### **% kmake Makefile**

Recreate a Pmakefile by executing `kgenmake` in a subdirectory.

### **% kmake Makefiles**

Recursively travels through all subdirectories and recreates the Pmakefiles by executing "kmake Makefile" in each subdirectory.



### **% kmake debug**

This compiles a program for use with `dbx`, `gdb`, or other debugger. It is only of use when VisiQuest as a whole has been installed *optimized*, and you wish to compile one or more *particular software object(s)* for use with the debugger. For information concerning installation of VisiQuest as a whole for optimization vs. debugging, see the VisiQuest Installation Guide.

### **% kmake optimize**

This compiles a program with the optimizer. It is only of use when VisiQuest as a whole has been installed for *debugging*, and you wish to optimize one or more *particular software object(s)*. For information concerning installation of the VisiQuest system as a whole for optimization vs. debugging, see the VisiQuest Installation Guide.

### **% kmake uninstall**

`% make uninstall` removes binaries and library archives from the install directory. It is typically used to undo a "kmake install".

### **% kmake struct**

This option is for support of user-defined structures in VisiQuest (see Chapter 6 of the Data Services Manual for details). The "kmake struct" option takes the user defined structure definition in a \*.x file and creates files called "io\_\*.c" and "io\_\*.h"; these files will contain the definition/reader/writer/match/free code for the structure.

## **Use of Integrated Software Development Tools**

There are several software tools that are not only integrated with VisiQuest, but are also highly recommended for use with the VisiQuest software development system. These include `lint`, `purify`, and `quantify`.

### **% kmake klint**

Runs `klint` on the software object. `Klint` is a perl script which will check the integrity of a toolbox or software object. It is particularly useful when you are getting a toolbox ready for release.

### **% kmake lint**

Executes `lint` on your source code. `Lint` is a C program verifier that attempts to detect features of the source code that are likely to be bugs, to be non-portable, or to be wasteful. It also performs stricter type checking than C compilers typically do. `Lint` is available with most operating systems (or, sometimes, with the software developer's kit that is offered with the operating system). Do `% man lint` for more information. Note that this option is only valid if `lint` is available on your system.

### **% kmake proto**

Calls the `mkproto` program, found in the *migration* toolbox, to automatically generate ANSI-C prototypes for each function in the program or library object. It will create a file called, "prototypes", in which there will be a comment containing the name of each file in the `src/` directory, followed by the ANSI-C prototypes for each function in that file. The contents of the "prototypes" file can then be easily inserted into the appropriate \*.h file(s) for the software object. Note that this option is only valid if the \$MIGRATION toolbox has been installed as part of the VisiQuest system at your site.

### **% kmake protoize**

Calls the `gnu` (non-VisiQuest) `protoize` program on the source code in your directory. The `protoize` program will do its best to convert K&R style source code to ANSI C prototype format. Do `% man protoize` for more details. Note that this option is only valid if the `protoize` program is available on your system; it is not distributed with VisiQuest, but with the `gnu` software.

### **% kmake purecov**

Executes `purify` on your object code and program executable. `Purecov` is a tool created by Rational Software that show which lines of code are executed during program execution.

### **% kmake purify**

Executes `purify` on your object code and program executable. `Purify` is a tool created by Rational Software Inc. which detects run-time memory problems; it is a great tool for finding those hard-to-locate memory overwrite bugs and for finding memory leaks. Note that this option is only valid if `purify` is available on your system, and that VisiQuest was installed with the `HasPurify Site.def` file flag set to YES. For more details, see *The VisiQuest Installation Guide*. For more information on `purify`, see the `Purify` manuals, or do: `% man purify`.

### **% kmake purtest**

Compile and run a testsuite under `purify`, then report status of testsuite run to stdout.

### **% kmake quantify**

Runs `quantify` on your program. Like `purify`, `quantify` is also produced by Rational Software Inc; `quantify` is a valuable tool for analyzing CPU usage, software control flow, memory use, and other runtime aspects of an application. Note that this option is only valid if `quantify` is available on your system, and that VisiQuest was installed with the `HasQuantify Site.def` file flag set to YES. For more details, see *The VisiQuest Installation Guide*. For more information on `quantify`, see the `quantify` manuals, or do: `% man quantify`.

### **% kmake tags**

Calls the (non-VisiQuest) `ctags` program on the source code in your directory. The `ctags` program creates a "tags" file for use with the `vi` text editor. A tags file gives the locations of functions in the files that make up your source code; once a "tags" file has been created for a

src/ directory, you can easily edit a function without having to know the name of the file that contains it (and therefore, not having to `grep` for the function if you have forgotten the name of the file in which it is defined). After you have created a "tags" file using `% make tags`, then you can take advantage of it by executing:

```
% vi -t desired_function_name
```

Note that this option is only valid if the `ctags` program is available on your system; it is not distributed with VisiQuest.

### **% kmake test**

Compile and run a testsuite, then report status of testsuite run to stdout.

The VisiQuest make system is set up to be very flexible when it comes to propagating make targets to sub directories. The directory Pmakefiles will pass any target specified to all subdirs which contain a Pmakefile. If the Pmakefile in that particular subdir doesn't understand the target, a warning is printed like the following:

```
% kmake unknown_target
--- 'unknown_target' ---
kmake unknown_target is not supported on a <classtype> object
```

<classtype> is the classtype of the software object the kmake command was run on. This warning can be safely ignored.

## **C. Kgenmake**

**Kgenmake** is the program that is used to generate a Pmakefile for a software development object. The first thing that must *always* be done before a software object is compiled is to create a Pmakefile, this is done as part of the code generation process. The Pmakefile is architecture independent; it has no information that is specific to a particular computer or operating system. Generally, it lists the files in a particular directory and indicates what those files are. For example, in a source code directory for a library object, the Pmakefile will list all the C code source files, all the object files, any \*.h files, Lex & Yacc files, and/or Fortran files, and will specify the library name and toolbox name. Initially, when the Pmakefile is being created in a directory for the very first time, **kgenmake** needs to know the name of the toolbox in which the software object exists, the name of the object, and the type of object it is. For later runs of **kgenmake**, however, information is taken from the software object itself so that command line arguments need not be provided.

The general rule of thumb concerning Pmakefiles is this: whenever you *add a new file, delete an old file* or *change a filename* in a source code directory, you *must* re-run **kgenmake**. Note, that composer will re-run **kgenmake** for you as necessary, so you *only* need to do it by hand if you've added the files to the software object by hand.

## D. Pmakefiles

Pmakefiles provide the information necessary to compile software in VisiQuest. There are a number of different types of Pmakefiles. There are Pmakefiles that compile software; these include Pmakefiles for programs, libraries, and scripts. There are Pmakefiles for directories; these Pmakefiles allow you to compile all software objects which exist at a particular directory level.

In general, the Pmakefile format includes a set of flags at the top of the file, followed by variables specifying object type, toolbox name, object name, include files, source code files, object files, scripts, and binary name. At the end of the Pmakefile are lines which include the files containing the rules and targets that kmake uses when it compiles a software object.

Because the contents of a Pmakefile differs according to the purpose of the Pmakefile, the format explanation below is the superset of all Pmakefiles in VisiQuest.

```
HasClassTypeDependence = 0 or 1
DoNotAutoRecreate      = 0 or 1
HasX11Dependence       = 0 or 1
HasCDependence         = 0 or 1
HasCplusplusDependence = 0 or 1
HasFortranDependence   = 0 or 1
HasOpenGLDependence    = 0 or 1
PosixOverride          = 0 or 1

OBJECT_TYPE = kroutine, library, xvroutine, script or pane

TOOLBOX_NAME = toolbox name

OBJECT_NAME = object name

HEADERS = list of include files
CSRCS = list of C source code files
YSRCS = list of yacc source code files
LSRCS = list of lex source code files
FSRCS = list of Fortran source code files
XSRCS = list of *.x source code files (used by kgenstruct to generate
reader/writer code for user-defined structures)
CXXSRCS = list of C++ source code files

COBJS = list of C object files
LOBJS = list of lex object files
FOBJS = list of Fortran object files
CXXOBJS = list of C++ object files

SCRIPTS = list of script files

BINARY_NAME = binary name

.include <${DESIGN_CONFIG}/toolbox.mk>
.include <Program.mk>
```

The flags at the top of the Pmakefile are as follows:

**HasClassTypeDependence**

This flag is set automatically by `kgenmake` and should not be modified.

**DoNotAutoRecreate**

This should be set to TRUE only if you do not want `kgenmake -recreate` to be able to recreate the Pmakefile from a template. This would only be the case if you have hand modified the Pmakefile to support make rules not otherwise supported.

**HasX11Dependence**

This should be set to TRUE if the source code makes calls to X Windows or the X Toolkit; otherwise, it should be set to 0. This will be automatically set on creation for xvoutines and libraries with an X dependence.

**HasCDependence**

This must be set to 1 if there are C (\*.c) source files to be compiled; otherwise, it should be set to 0. This will be automatically set on creation for kroutines, xvoutines, and libraries.

**HasCplusplusDependence**

This must be set to TRUE if there are C++ (\*.C) source files to be compiled; otherwise, it should be set to 0. This will be automatically set on creation for routines that specify a C++ language dependence.

**HasFortranDependence**

This must be set to TRUE if there are Fortran (\*.f) source files to be compiled; otherwise, it should be set to 0. You must set this flag yourself if you add Fortran source files to your routine.

**HasOpenGLDependence**

This must be set to TRUE if there are any calls to OpenGL in the code; otherwise, it should be set to 0. You must set this flag yourself if you have any dependencies on GL libraries.

**PosixOverride**

If set, this flag tells the compiler to relax strict POSIX compliance standards. Certain system calls are not supported by the POSIX standard; setting this flag will allow you to compile routines using non-POSIX system calls. Note that doing this may adversely affect the portability of your code. By default this flag should be set to 0.

At the end of the Pmakefile will be two ".include" lines. These lines include the configuration files that specify the make targets understood by `kmake` for the type of software object in question.

The first line is the "toolbox.mk" file for the toolbox in which the software object exists. For example, the *image* toolbox would have the line:

```
.include <${IMAGE_CONFIG}/toolbox.mk>
```

This line is automatically inserted by `kgenmake` when the software object is created.

The second line is the "*class\_type.mk*" file, where "class type" is specific to the software object in question. Scripts use a "Script.mk" file. Directories use a "Directory.mk" file. The rest of the software object class types are all supported by other mk files. So, for example a kroutine would have the line:

```
.include <Kroutine.mk>
```

A directory, on the other hand, would have the line:

```
.include <Directory.mk>
```

The next section in this chapter describes the contents of the "toolbox.mk" toolbox configuration file.

## E. The toolbox.mk File

Every toolbox must have a toolbox configuration file. This file is called "toolbox.mk" and is located in \$TOOLBOX/repos/config/mk. This file defines specific variables pertaining to your toolbox. These variables are used by kmake to tailor the generalized kmake rules for use by your specific toolbox. The "toolbox.mk" file consists of size major sections.

### E.1. File header

```
#####  
# toolbox configuration file for toolbox: "bootstrap"  
#####
```

The file header prevents multiple inclusion of this "toolbox.mk" file by other toolboxes. Do not touch.

### E.2. Inclusion of dependent toolbox's configuration files

```
#####  
# Include dependent toolbox.mk rules:  
#####
```

This section is managed completely by craftsman and should NOT be modified. Any manual changes made to this section will be lost the next time craftsman is used. To change toolbox dependencies, use the "Toolbox Dependencies" pane of craftsman, accessible via the the "Toolbox Attributes" menu.

### E.3. Variables section

```
#####  
# Define variables  
#####
```

This section is where the INCLUDES variable is modified to include this toolbox's include directory. In other words, this is what causes the "-I ..." option on the compile line to search the include directory for this toolbox when C and C++ code is compiled. If your code is dependent on third-party software, you may add additional lines to this section. For example, the VSIP toolbox includes the VSIP image includes as follows:

```
INCLUDES ^= -I${VSIP}/include/vsip_image
```

## E.4. Toolbox library definitions

```
#####  
#   Define toolbox libraries  
#####
```

This section contains the specifications for the libraries which exist in this toolbox. Libraries are listed in categories, where categories include ANSI C libraries, VisiQuest C libraries, ANSI C++ libraries, VisiQuest C++ libraries, Fortran libraries, and X/Xt-dependent libraries. Each category has a required *if defined / endif* syntax that must be present. Other classtypes can add sections for their own special library types as necessary, so don't be surprised to find additional sections. **Craftsman** will automatically append and remove libraries from the appropriate sections, but sometimes it is necessary to manually change the order of that the libraries are in because **craftsman** has no way to determine library order dependence. In the specification, if library B depends on library A, you must list A first and then B, so that it will appear in the opposite order on the command line (the libraries are prepended). For example, in the C libraries listed below, the *kexpr* library depends on the *klm* library, so it must be listed after *klm*. The library listings are as follows.

### ANSI C libraries

These are C libraries which have no VisiQuest dependencies. For example,

```
.if defined(HasANSICDependence) && ${HasANSICDependence}  
TBLIBS ^= -lmyclib  
.endif
```

### VisiQuest C libraries

These are C libraries which depend on other VisiQuest C libraries, with the exception of X/Xt libraries. For example,

```
.if defined(HasCDependence) && ${HasCDependence}  
TBLIBS ^= -lkc  
TBLIBS ^= -lku  
TBLIBS ^= -lklm  
TBLIBS ^= -lkexpr  
TBLIBS ^= -lkvf  
TBLIBS ^= -lkclui  
.endif
```

### Fortran libraries

```
.if defined(HasFortranDependence) && ${HasFortranDependence}  
TBLIBS ^= -lapack  
.endif
```

### X Windows and X Toolkit libraries

```

    .if defined(HasX11Dependence) && ${HasX11Dependence}
    TBLIBS ^= -lxvw
    TBLIBS ^= -lkwid
    TBLIBS ^= -lxvobj
    TBLIBS ^= -lxvu
    TBLIBS ^= -lxvf
    TBLIBS ^= -lxvl
    .endif

```

### ANSI C++ libraries

These are C++ libraries which have no VisiQuest dependencies. For example,

```

    .if defined(HasANSICXXDependence) && ${HasANSICXXDependence}
    TBLIBS ^= -lmycxxlib
    .endif

```

### VisiQuest C++ libraries

```

    .if defined(HasCplusplusDependence) && ${HasCplusplusDependence}
    TBLIBS ^= -lSimLib
    .endif

```

## E.5. Normal target rule definitions

```

#####
#   Define normal target rules:
#####

```

This section is where you would add additional `kmake` targets for your toolbox, if desired.

## E.6. System macros

```

#####
#   Low-level system macros: (don't change anything below this line)
#####

```

These macros are automatically generated by `craftsman` on toolbox creation. Do not touch.

This section includes a toolbox-specific Site file if one exists, and sets `${BINDIR}` and `${LIBDIR}` for the toolbox. It also adds the library directory to the `"-L ..."` option of the link line and the `-R` or `-rpath` options.



**This page left intentionally blank**

## Table of Contents

A. Introduction . . . . .	6-1
B. Kmake Targets . . . . .	6-2
C. Kgenmake . . . . .	6-6
D. Pmakefiles . . . . .	6-7
E. The toolbox.mk File . . . . .	6-9
E.1. File header . . . . .	6-9
E.2. Inclusion of dependent toolbox's configuration files . . . . .	6-9
E.3. Variables section . . . . .	6-9
E.4. Toolbox library definitions . . . . .	6-10
E.5. Normal target rule definitions . . . . .	6-11
E.6. System macros . . . . .	6-11

**This page left intentionally blank**

# *Toolbox Programming*

## **Chapter 7**

# **Packaging Your Toolbox**

Copyright (c) AccuSoft Corporation, 2004. All rights reserved.



# Chapter 7 - Packaging Your Toolbox

## A. Introduction

VisiQuest contains packaging tools for toolbox distribution. The *kpacktb* tool is used to package up a toolbox or a set of toolboxes for distribution; the *xpacktb* application provides a convenient graphical user interface front-end to the *kpacktb* tool. The *kpkgadd* ("kpackageadd") tool is used to unpack and install the toolbox(es) that were previously packaged using *kpacktb* or *xpacktb*.

The *kpacktb* tool uses a configuration file called "kpacktb.ini" which specifies site-specific packaging parameters and contains a number of switches that give the developer fine-grain control over how their toolboxes are packaged. It creates a \*.pkg file which is a gzipped tar file containing the desired toolboxes. This file can then be distributed to other users, who will then run *kpkgadd* with the package file as input. *Kpkgadd* will then install the packaged toolboxes contained in the package file.

Before creating packages using *kpacktb* or *xpacktb*, we strongly recommend using *klint* to verify the integrity of your toolbox(es) and software objects prior to distribution. *Klint will detect extraneous files, as well as valid files that may not be registered in the database(s).*

The following sections describe the VisiQuest packaging tools in detail.

## B. **kpacktb** — *package toolbox(es) for distribution*

### Object Information

Category:	<i>Software Management</i>	Subcategory:	<i>Installation</i>
Operator:	<i>Create distributions of toolboxes</i>	Object Type:	<i>Script</i>
In Cantata:	<i>No</i>		

### Description

**kpacktb** is a utility for packaging toolboxes for distribution. A VisiQuest package may be distributed in two forms: a package file which is a gzip-tar file and has the extension .pkg, and a CD image which is simply an unzipped-untarred copy of the .pkg file. Either form of the distribution may then be used with the **kpkgadd** or **ktbsetup** programs to install the package or update a currently installed package on their system.

**kpacktb** operates in a patch create mode as well. When the *-patch* option is specified on the command line, the *-tb* option and the ini file *Toolboxes=* value are ignored. Instead, the list of toolboxes is interpreted from the filename specified in the *-patch* option. This patch file has the following format:

```
TB1 : : oname1
TB1 : : oname2
```

```
TB1/path/to/file_to_include
TB2::oname3
```

This tells **kpacktb** to create a patch package, including the objects oname1 and oname2 from TB1, and oname3 from TB2. It also tells **kpacktb** to include the file path/to/file\_to\_include in TB1. The output produced is a packname.pat file which is interpreted by **kinstpatch**

**kpacktb** is highly configurable. Options may be given on the command line, or in the kpacktb.ini file (see below). With the command line options, the user may direct **kpacktb** to build a distribution consisting of compiled binaries only (statically or dynamically linked), source code only, or both binaries and source code. If a toolbox is under CVS revision control, the 'keepcvs' flag also lets the user include CVS directories in the distribution. More refined control over what files are and are not included in the distribution may be specified in the kpacktb.ini file (see below).

The kpacktb.ini file is stored in the custom-dir. Run "kecho -echo custom-dir" to find the path to that directory on your system. The kpacktb.ini file specifies site-specific packaging parameters and contains a number of switches that give the user fine-grain control over how the toolboxes are packaged.

We now describe arguments in kpacktb.ini. The file syntax is explained by the example below.

#### [general]

The settings in the "general" section establish the default values for the packaging system overall. Any and all of these settings may be reset for individual toolboxes in that toolbox's own section (eg., [bootstrap]).

#### Toolboxes

A comma separated list of toolboxes available for packing. If the "-tb" option is not given on the command line, all the toolboxes in this list are packed.

#### basedirs

A comma separated list of directories to be packed with the set of base directories. These directories are the directories that should be distributed with a "source code" distribution. Eg., objects,include,repos,thirdparty.

exdirs A comma separated list of directories to be packed with the examples. Eg., examples,workspaces.

testdirs A comma separated list of directories to be packed with the testsuite. Eg., testsuite.

sitedir The directory in which Site.\* template files are found. Eg., repos/config/mk/config

bindir The directory in which compiled binaries are found. Eg., bin.

libdir The directory in which compiled libraries are found. Eg., lib.

tb\_ignoredirs

A comma separated list of directories that should not be included in the distribution. Eg., config,mach,bin,lib>manual.

tb\_keepdirs

A comma separated list of directories that should be included in the distribution. This would be any special, custom, or local standard directory. For example, if your company, LewisInc, has a set of toolboxes all of which contain an extra LewisWare directory that you wish to include as part of the distribution, it would be added to this list.

tb\_ignorefiles

A comma separated list of files that should not be included in the distribution. For example, you may keep a MYLOG file in the repos directory that should not be distributed. You may then add "repos/MYLOG" to this list.

exclude\_objects

A comma separated list of object names that should not be included in the distribution. Every object that does not appear on this list will be packaged into the distribution.

exclude\_classtypes

A comma separated list of classtypes that should not be included in the distribution. Eg., if "kroutine" is on this list, no kroutines will be included in the distribution.

obj\_ignoredirs

A comma separated list of object directories that should not be included in the distribution. Eg., if "uis" is on the list, then no <TOOLBOX>/objects/<CLASSTYPE>/<OBJ\_NAME>/uis directories will be included in the distribution.

obj\_ignorefiles

A comma separated list of object files that should not be included in the distribution. Eg., if "man/whatis" is on the list, then no <TOOLBOX>/objects/<CLASSTYPE>/<OBJ\_NAME>/man/whatis files will be included in the distribution.



obj\_keepdirs

A comma separated list of object directories that should be included in the distribution. This would be any special, custom, or local standard directory. For example, if your company, LewisInc, has a toolbox and every object has an extra LewisWare directory that you wish to include as part of the distribution, it would be added to this list.

nobintypes

A comma separated list of Classtypes that do not have a corresponding binary file. Eg., Pane,WorkspaceObject.

libtypes

A comma separated list of Classtypes that are libraries. Eg., Library,ParallelLibrary.

du\_command

The disk usage command to use. Eg., "/usr/local/gnu/bin/du -sk".

libman\_ignore

Should library man pages not be included in the distribution? Takes the value "yes" or "no".

libman\_tb

If libman\_ignore is set to "no", all library man pages (no matter which toolbox the library is located in) will be packaged into the toolbox named by the libman\_tb argument. To bundle library man pages with the same toolbox that the library is in, leave the libman\_tb parameter blank.

formatdoc

Should documentation be formatted before packaging? Takes the value "yes" or "no".

unpacked

Should the toolboxes be packaged in "unpacked" form. Takes the value "yes" or "no". This argument is best explained by example. Normally, this argument is set to "no". The one exception is for the SAMPLEDATA toolbox which contains no software objects but many datasets making the toolbox itself very large. If we wish to distribute SAMPLEDATA on a CD, then we would like to give the user the option of running the toolbox off the CD itself rather than requiring the user to install it on a harddrive. By specifying that SAMPLEDATA be packaged in unpacked form, the CD image of the package is "unpacked" so that users may run directly off CD. If we did not specify unpacked, the toolbox would appear as a directory of gzipped-tarred files on the CD.

[options]

The options section corresponds to the command line arguments. Arguments specified here are overridden by values given on the command line.

output\_dir

Directory in which to build the distribution. This is the directory in which to find the package file and/or CD image after the distribution is built.

packname

Package name. Eg., if this is set to "fromage", the package name will be "fromage.pkg".

keepcvs

If the toolboxes are under CVS revision control, do we include the CVS directories in the distribution? Takes the value "yes" or "no". This argument may also be set in the toolbox specific sections (see below).

bindesc

If the distribution is to include binaries, this argument is a string describing the binaries. Eg., "Solaris 2.3 - gcc 2.8 - glibc 2.5"

binonly

Binary only? Takes the value "yes" or "no". If set to "yes", directories set in the basedirs argument will not be included in the distribution.

standalone

Compile the binaries static? Takes the value "yes" or "no". Ignored if not a binary distribution.

noclean

If the force and noclean are both set to "yes", then the output\_dir is not cleaned before building the distribution. If force is not set, or set to "no", then noclean is ignored.

nopackage

Only build the CD image, no \*.pkg file? Takes the value "yes" or "no".

verbose

Verbose output? Takes the value "yes" or "no".

force If force is set to "yes", then the output\_dir is cleaned before building the distribution (unless noclean is also set to "yes"). If force is set to "no", then the user is prompted to see if output\_dir should be cleaned.

### Toolbox Specific Sections

Following the "[general]" and "[options]" sections are any number of toolbox specific sections (eg., "[bootstrap]") that apply only to that toolbox and override the defaults given earlier in the kpacktb.ini file. Any arguments given in the above sections may be used here. See the Examples section below.

## Examples

Here is an example kpacktb.ini file:

```
[general]
Toolboxes=bootstrap,datamanip,dataserv,db_migrate,design,envision,geometry,image,matrix,migration,retro,support

;
; toolbox directories that make up distributions
;
basedirs=objects,include,repos,data,thirdparty,hardcopy,shared,figures
exdirs=examples,workspaces
testdirs=testsuite
; sitedir, bindir, and libdir may only contain one dir
sitedir=repos/config/mk/config
bindir=bin
libdir=lib

;
; Specific toolbox configuration
;
; don't want these toolbox dirs
tb_ignoredirs=config,mach,bin,lib>manual
; keep these toolbox dirs
tb_keepdirs=
; don't want these toolbox files
tb_ignorefiles=

;
; Object exclude (both classtype and specific names are supported)
;
exclude_objects=
exclude_classtypes>manual

;
; Specific object configuration
;
```

```

; don't want these object dirs
obj_ignoredirs=
; don't want these object files
obj_ignorefiles=
; keep these object dirs
obj_keepdirs=

;
; NoBinary and Library object classtypes
;
nobintypes = Pane,ParallelPane,WorkspaceObject
libtypes = Library,ParallelLibrary,SACLibrary

;
; Specify a du command
;
du_command = /usr/local/gnu/bin/du -sk

;
; Library man page configuration
;
libman_ignore=yes
; library man pages need to go here if ignore is no
libman_tb=devel

;
; Run all doc through kman as part of the pack process
;
formatdoc=yes

[options]
; directory to install into
output_dir=/usr/users/fred/VisiQuest/release
force=yes

;
; specific overrides to the general settings for the DESIGN toolbox
;
[design]
basedirs=objects,include,repos,data,thirdparty,hardcopy>manual/share
tb_ignoredirs=config,mach,bin,lib

;
; specific overrides to the general settings for the SAMPLEDATA toolbox
;
[sampledata]
unpacked=yes
libman_ignore=yes

```

With the above `kpacktb.ini` file installed, we could create a LewisWare distribution consisting of the three `lw` toolboxes with the command:

```
$ kpacktb -packname LewisWare -tb lw_parker,lw_randall,lw_steiner
```

The resulting distribution would be found in `/usr/users/fred/VisiQuest/release`. To distribute this package to others, one would only need to give them the `LewisWare.pkg` file and the `kpkgadd` installation script. Alternatively, one could press a CD with the contents of the `/usr/users/fred/VisiQuest/release/LewisWare` directory and the `kpkgadd` script.

### See Also

`kpkgadd`

## C. `xpacktb` — *graphical user interface for kpacktb*

### Object Information

Category:	<i>Software Management</i>	Subcategory:	<i>Utilities</i>
Operator:	<i>xpacktb</i>	Object Type:	<i>Xvroutine</i>
In Cantata:	<i>No</i>		

### Description

**xpacktb** is a front end for the `kpacktb` script, which packages toolbox(es) for distribution. The main advantage of using **xpacktb** is that it automatically creates a "`kpacktb.ini`" file for you; the "`kpacktb.ini`" is required by **kpacktb** to specify site-specific packaging parameters. Another advantage of using **xpacktb** is that you can build the list of toolboxes to pack by picking from a list of available toolboxes using the mouse, rather than having to type them all in.

### See Also

`kpacktb`

### Command Line Arguments

```
=====  
Usage for xpacktb:  
% xpacktb  
=====
```

## D. **kpkgadd** — *install a package of toolbox(es)*

### Object Information

Category:	<i>Software Management</i>	Subcategory:	<i>Installation</i>
Operator:	<i>kpkgadd</i>	Object Type:	<i>Script</i>
In Cantata:	<i>No</i>		

### Description

**kpkgadd** is a highly configurable package installation program. Its purpose is to install packages built with the `kpacktb` program. These packages may contain any combination of compiled binaries, source code, examples, testsuites, sample data, or none of these things. It may run interactively or in *auto* mode for automated testing.

#### The Five Stages of Operation

First, it unpacks the package file into a *staging* area (directory). If the user is installing off of a CD, the package is already unpacked on the CD, and the CD itself acts as a staging area. Otherwise, the \*.pkg file is gunzipped and untarred into the staging directory.

Second, the user is queried about the system on which it is currently running, the system for which it is installing, and the current VisiQuest installation (if any). Some packages may contain binaries for more than one platform. In this case, the user may, for example, run **kpkgadd** on a linux machine to install SunOS binaries. Therefore, **kpkgadd** would ask about both the machine on which it is currently running (linux), and the machine for which it is installing (SunOS). If VisiQuest is already installed, **kpkgadd** will also ask where the **kecho** program is located.

The third step is to present the user with a menu of configurable installation options. The first menu item is a list of toolboxes scheduled to be installed. The remaining menu items are the installation parameters associated with each of the toolboxes. By selecting item 1, the user may edit the list of toolboxes to install. By selecting the other menu items, the user may edit the installation parameters for the toolboxes (eg., the toolbox installation directory, the bin and lib directories).

Fourth, the system is installed. This should run without requiring additional input from the user. While installing, it prints messages to the screen announcing its progress.

Finally, the staging area is cleaned. If a staging area was created, that area is removed before exiting the program.

#### Toolbox Specific Installation Instructions

Every toolbox contains a `$TOOLBOX/repos/config/setup/toolbox.sh` file. This file contains seven required subroutines and any number of optional support routines that specify how the toolbox is to be installed.

For example, when installing a package that contains the BOOTSTRAP toolbox, **kpkgadd** will *source* BOOTSTRAP's toolbox.sh file, then run the TBQuestions routine to query the user to set parameters that are specific to the BOOTSTRAP toolbox. Later, it will *source* the toolbox.sh again and run some or all of TBInstall, TBSetup, TBCompile, TBCleanUp, and TBFinalMessages. The names of these routines must not be changed.

For most toolboxes the toolbox.sh file does not need to be customized. It will work fine the way it is. Some reasons a toolbox programmer may want to customize this file would include; to install a Site.\* file; to install thirdparty software; and to set environment variables needed for compilation and/or testing. All of these would require some additional customization. If you find that you need to customize your toolbox.sh, it is strongly recommended that you familiarize yourself with some other toolbox.sh that performs a task similar to what you wish to do in your toolbox. Pay special attention to the appearance of the output to maintain a seamless look between the **kpkgadd** program and the various toolbox.sh's that are being run.

The required toolbox.sh routines are:

### TBQuestions

This is the only routine in the toolbox.sh file that is to be run interactively. No other routine should query the user for information of any kind. This routine takes one argument, the queryFlag. If this flag is TRUE, the routine may ask the user for input. If it is FALSE, it must take default values without asking the user for input. NOTE: There are some extreme exceptions to violating the queryFlag rule, but in general this rule must be honored.

The convention for setting parameters is to write a supporting routine for each parameter that needs to be set. Each parameter has a *key* that describes it, and a *value* to which the parameter is set. The *key:value* pair are stored in the file whose path is stored in \$G\_tbParametersFile. This file is *cat*'ed to the screen to show the user the current parameter settings, so it is important that the spacing is consistent with the other menu items.

The DEVEL toolbox's toolbox.sh file is a good example of how one may query the user to choose a toolbox local Site.\* template file.

For a more complicated example, look at the BOOTSTRAP toolbox's toolbox.sh file. It queries the user to choose a Site.\* template, and then stores a series of parameters to substitute into the chosen Site.\* template.

### <DT> TBInstall

This routine unpacks the needed compressed tar files from the staging area for this toolbox. If there are thirdparty compressed tar files in the staging area, you may add instructions to this routine to unpack these files also.

<DT> TBSetup

This routine does anything that should be done after files are untarred, but before compiling begins. For example, this is where the Toolboxes file is edited, and the Dir.\* file is created.

See also DEVEL and BOOTSTRAP's toolbox.sh. They use the TBSetup routine to create the Site.\* file.

<DT> TBCompile

This routine runs the **kmake** commands to compile and install the toolbox's software objects. If this is a *binary only* package, this routine is not run.

Conventionally, one redirects the output of **kmake** to a make.World file so that the compiler output may be examined for debugging purposes.

Any thirdparty software should also be compiled and installed in this routine.

<DT> TBCleanUp

Typically this routine just runs "kmake clean." Do not forget to also cleanup any thirdparty software directories.

<DT> TBFinalMessages

After the installation, this routine prints out any information or instructions that users should be aware of. If any special environment variables need to be set, inform the user here. If any setup/configuration programs need to be run, this is the place to let the user know.

### **Variables and Routines Available to the toolbox.sh Programmer**

Several variables and routines are available to toolbox.sh programmer. Some of the more useful ones are listed here. We start with the variables:

G\_path Set to the directory where the toolbox is installed. Eg., /usr/local/VisiQuest/bootstrap.



<DT>G\_binpath The path to the bin installation directory.

<DT>G\_libpath The path to the lib installation directory.

<DT>G\_tsInstall A flag set to \$G\_TRUE or \$G\_FALSE indicating if testsuites should be installed.

<DT>G\_exInstall A flag set to \$G\_TRUE or \$G\_FALSE indicating if examples should be installed.

<DT>G\_auto A flag set to \$G\_TRUE or \$G\_FALSE indicating if we are running in *auto* mode.

<DT>G\_CALLING\_PROGRAM A switch set to KPKGADD or KTBSETUP depending on which program is sourcing the toolbox.sh file.

<DT>G\_verbose A flag set to \$G\_TRUE or \$G\_FALSE indicating if we are running in verbose mode.

<DT>G\_stagingDir The path to the staging directory.

<DT>G\_packageType A switch set to \$G\_BINARY\_PACKAGE if the package contains compiled binaries. Set to \$G\_COMPILE\_PACKAGE if the TBCompile routine will be called.

<DT>G\_configurationName The configuration name for the current installation. This would be used, for example, in naming the Dir.\* and Site.\* files. For example, if the config-name is 'linux', the Dir and Site files would be Dir.linux and Site.linux.

<DT>PATH The executable path environment variable.

<DT>TMPDIR The temporary directory environment variable.

<DT> Misc unix commands Complete paths to the following unix commands are set up.

sed, csh, pwd, dirname, basename, tr, grep, awk, df, wc, tail, head,  
cat, rm, rmdir, mkdir, tar, ls, more, sort, mv, cp, chmod, gzip,  
basename, make, touch, tee

So, for example, to run 'tee' in the toolbox.sh file, run the command

Some routines that may be run from the toolbox.sh are:

echon echon is the local equivalent of "echo -n", echo with no carriage return.

<DT>BinaryInPath usage: BinaryInPath cmd

This routine checks if the given binary is in the PATH. Returns \$G\_YES or \$G\_NO accordingly. If found, sets \$G\_binaryPath to the location found.

<DT>SetupBinary usage: SetupBinary cmd [args ...]

Given a string, cmd, this routine searches the PATH for a binary by the same name. The path to the first one encountered becomes the value of a new variable, \$cmd (where cmd is the actual string passed in) appended by the args given (if any).

Eg. if /bin/rm is in the PATH, 'SetupBinary "bin" "-f"' will create a variable, \$bin, with the value '/bin/rm -f'.

If no such binary is in the PATH, an error is reported and exits the program.

<DT>YesNoPrompt usage: YesNoPrompt str default

This routine asks a yes or no question (given in the string) with the given default, and keeps looping until it gets an answer of yes or no. Exits with status \$G\_YES or \$G\_NO according to the response.

<DT>SubstituteStr usage: SubstituteStr file oldStr newStr  
limitation: Won't work on strings with '!' in them

This routine substitutes one string for another in the file. The file must be both readable and writable.

<DT>ToolboxesFileAppend usage: ToolboxesFileAppend tb tbStr

This routine appends a line to the Toolboxes file in the installation directory. The line that is appended is "\${tb}:\${tbStr}".

If there is no Toolboxes file in the installation directory, one is created.

If there is already a \$tb line in the file, it is commented out before appending.

<DT>CheckCompile usage: CheckCompile binList binDir libList libDir

This routine checks that the list of binaries are installed in the correct directory and are executable and that the list of libs are in the correct directory.

The binList should be a space separated list of executable names.

The libList should be a space separated list of binary names. I.e., if libkclui.a is the name of a library we expect to find in the lib dir, the kclui should be an item on the list.

Prints warnings for anything that is missing. Returns \$G\_TRUE if everything looks okay, \$G\_FALSE otherwise.

### **toolbox.sh Substitutions**

When kpacktb creates a package, it makes certain substitutions into the toolbox.sh file that it includes in the package. We describe those substitutions here:

%dependency-list%

Is replaced with a whitespace separated list of toolboxes on which this toolbox is dependent.

<DT> %bin-list%

Is replaced with a whitespace separated list of files to be installed in the bin directory.

<DT> %lib-list% Is replaced with a whitespace separated list of files to be installed in the lib directory.

## Examples

To install the VisiQuestBase package, run:

```
$ kpkgadd ./VisiQuestBase.pkg
```

If you wish to run in auto mode in the background, put your responses into a variable (eg. \$resp) and run it as:

```
$ echo $resp | kpkgadd ./VisiQuestBase.pkg >& kpkgadd.output &
```

## See Also

kpacktb, kecho, kmake.

## Required Command-Line Arguments

### package

type: file

desc: package file (\*.pkg) to install

## E. klint — *check for toolbox and object integrity*

### Object Information

Category: *Software Management*

Subcategory: *Object Utilities*

Operator: *Object Lint*

Object Type: *Script*

In Cantata: *No*

## Description

Klint is a perl script which will check to see whether a toolbox or software object is in a good state. It is particularly useful when you are getting a toolbox ready for release, and should be part of the release process you adopt.

You would check the entire SUPPORT toolbox using the following command:

```
% klint -tb support
```

You can specify a particular object with the `-oname` switch:

```
% klint -tb support -oname kdbmedit
```

The script will check to see if the database matches the files in the toolbox and/or object's directories. If no toolbox or object is specified, then klint will check all toolboxes available. Multiple toolboxes can be specified by separating toolbox names with a comma. For example, to check the SUPPORT and MIGRATION toolboxes, you would use the following command:

```
% klint -tb support,migration
```

You can also specify multiple objects to check. For example, to check *ghostwriter*, and *conductor*, in the BOOTSTRAP toolbox, you would use the following command:

```
% klint -tb bootstrap -oname ghostwriter,conductor
```

If you are using cvs and do not want klint to complain about the CVS directory use the `-ignorecvs` flag.

```
% klint -ignorecvs -tb toolbox1
```

## Examples

To check the kdataserv object in the VisiQuest toolbox do,

```
% klint -tb VisiQuest -oname kdataserv
```

or if the kdataserv object is in the VisiQuest toolbox and the VisiQuestdevel toolbox do,

```
% klint -tb VisiQuest,VisiQuestdevel -oname kdataserv
```

## Table of Contents

A. Introduction . . . . .	7-1
B. kpacktb — <i>package toolbox(es) for distribution</i> . . . . .	7-1
C. xpacktb — <i>graphical user interface for kpacktb</i> . . . . .	7-8
D. kpkgadd — <i>install a package of toolbox(es)</i> . . . . .	7-9
E. klint — <i>check for toolbox and object integrity</i> . . . . .	7-14

**This page left intentionally blank**

## **Chapter 8**

# **Test Suites**



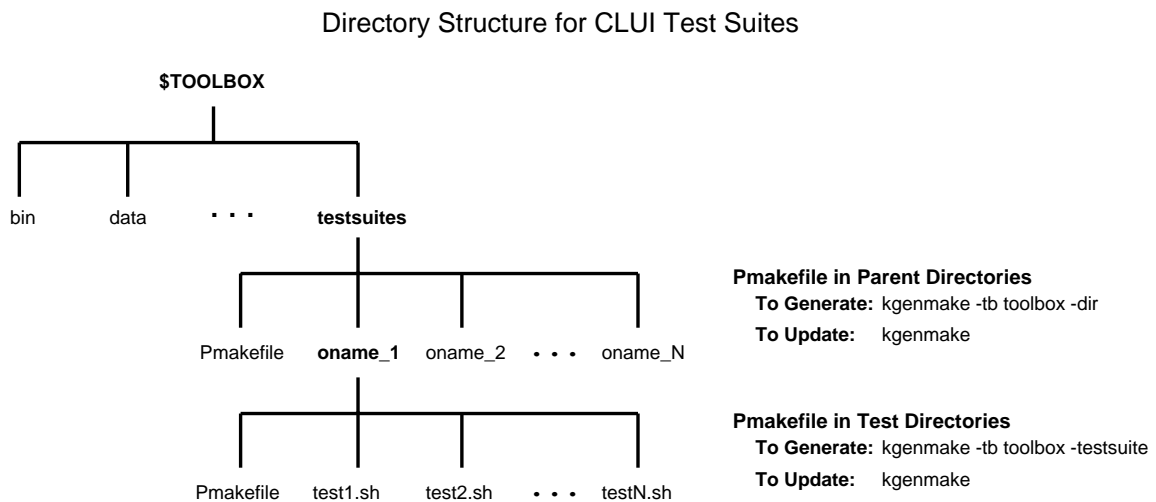


# Chapter 8 - Test Suites

## A. Introduction

*Test suites* are an important component of a toolbox and should be an integrated part of the software development process. They provide a reproducible, non-interactive method of algorithm verification that helps guarantee the integrity, robustness, and portability of your code. If you intend to distribute your toolbox, the test suites will simplify the process of porting your code to other architectures and system configurations, and they allow users who are unfamiliar with your code to verify it. Test suites also test the VisiQuest infrastructure, and therefore are useful in determining whether infrastructure changes have introduced unexpected bugs. Though it can be time consuming to write, a good test suite will save time in the long run. Currently, there is support in VisiQuest for writing test suites for library and non-X-based routines (kroutine, pane, and script objects).

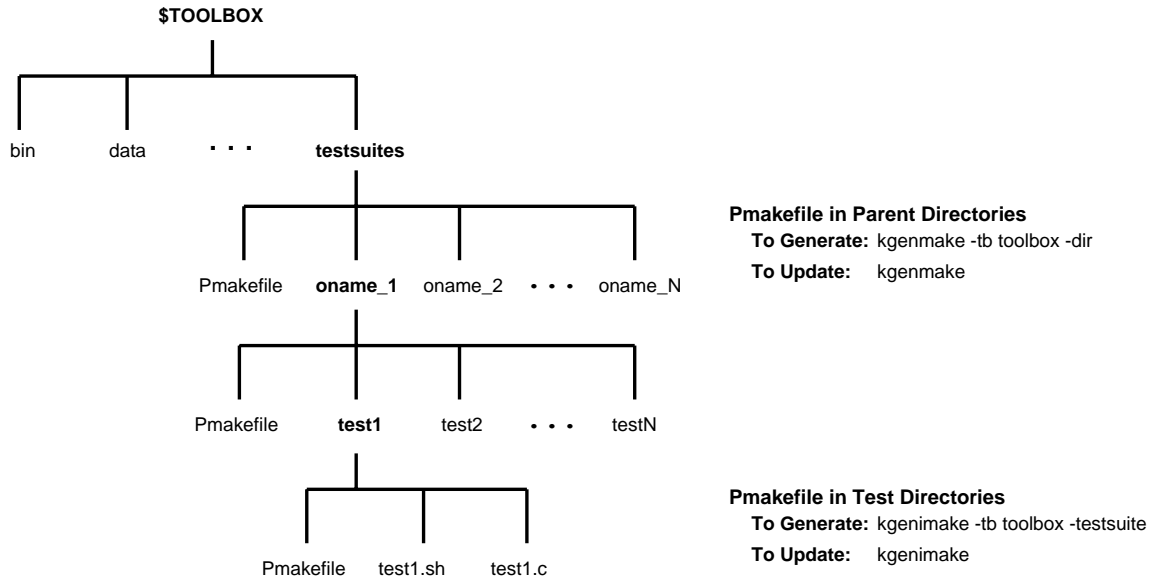
## B. The Testing Environment



**Figure 1:** Suggested directory structure for test suites that test program functionality via the command line. In the diagram, *o\_name* refers to the name of each program being tested.

A test can be either a script file that tests the program from its command line user interface (CLUI), or a compiled program that tests library function calls. Test suites are not supported as software objects, and therefore are not accessible *craftsman* or *composer* so you will need to create the directories and their associated

## Directory Structure for Library Test Suites



**Figure 2:** Suggested directory structure for test suites for library function call tests. In the diagram, *o\_name* refers to the name of each object, for example each library, being tested.

Pmakefiles as you go.

### B.1. Test Suite Directory Structure

Figures 1 and 2 show the recommended way to organize test directories. When you created your toolbox, a testsuite directory was created for you at the top level of the toolbox. Create a subdirectory for each program object (kroutine, pane, script, or library).

The only difference between the directory structure for library tests and CLUI tests is that each test binary for the library must be in a separate subdirectory (Figure 2). CLUI tests only contain scripts, and all of these scripts can be in the same directory (Figure 1). A library test directory may also contain several script files — the restriction is only placed on the number of binaries per directory. For examples of CLUI tests, see the test suites in the Datamanip toolbox. For examples of library tests, see the test suites in the Dataserv toolbox.

### B.2. Test Suite Pmakefiles

For each directory that contains a test (scripts or a compiled routine), create the Pmakefile by typing the following command.

```
% kgenmake -tb <toolbox> -testsuite
```

This command will make a reference to all script and source files (\*.sh, \*.pl, \*.c, etc.) in the Pmakefile. If new files are added to the test directory, regenerate the Pmakefile by typing

```
% kgenmake
```

To create a Pmakefile in a parent testsuite directory, for example, in the `$TOOLBOX/testsuite` directory, type

```
% kgenmake -tb <toolbox> -dir
```

This command will reference all subdirectories contained in the parent directory in the Pmakefile. If new subdirectories are added, update the Pmakefile by running:

```
% kgenmake
```

### B.3. Requirements

A few things are important to keep in mind when writing test suites. First of all, the tests should be non-interactive. They should provide pass/fail results without requiring a user to verify the output data. Second, whenever possible, avoid using tools with known architecture dependencies; for example, use `sh` over `csh`. Third, if verification data sets are saved, they need to be small to keep the size of the toolbox at a minimum.

### B.4. Test Data

Test Data sets can be found in the data directories of the Sampledata toolbox, and in the `data/testdata` directory of the Datamanip toolbox. Data sets can also be created on the fly by generating and importing ASCII files (see Example 1 in the CLUI example script given later in this chapter). If your test suites depend on data in another toolbox, document that dependency in a README file in the `$TOOLBOX/testsuite` directory.

### B.5. Suggestions for Writing Test Suites

When creating a test suite, run the test and verify the results manually using programs such as `kprdata`, `kstats`, `ksegcmp`, and `editimage`. Test combinations and variations of the input parameters to your program or function, including boundary conditions and unexpected values. If you wish to test the large data processing capability of your program, but want to work with small data sets, you can set the `VisiQuest_DATA_SIZE` environment variable. This variable indicates to Data Services a memory buffer size, and is typically initialized to 2Mb. Setting the variable to a much smaller value, 8 bytes for example, will force the data to be accessed as if it were a very large data set.

Integrating test suites into the software development cycle, as opposed to writing them as the last step, can help increase productivity. For example, if your program should process in the data type determined by the input object data type (the `karith2` program is an example of this), first prototype the operation in a single data type (double) and test it fully before replicating code. If you use a reference data object when writing the program, you can set the actual data type that will be written to the output data object, yet process in the "prototype" data type. Then, when code to support all types is added, the original test suite should run without errors.

When writing `sh` scripts, it is useful to append file names with `$$`.

```
karith1 -i image:ball -o tmpdir/result.$$
```

This will give each temporary file a unique name based on the process ID, and will guarantee that the temporary files do not get overwritten by another process. It is also recommended that you write the temporary files to the directory defined by the `TMPDIR` environment variable. This removes path dependencies defined by

your local environment, and allows the temporary directory to be changed easily when space runs short. The CLUI test template provided in the VisiQuest bootstrap toolbox sets a `tmpdir` variable in the script by checking the `TMPDIR` environment variable.

If you wish to debug a `sh` test script, you can add the `-x` flag to the script and all commands will be echoed to the `tty`.

```
#!/bin/sh -x
```

## B.6. Useful Routines for Testing

There are many routines in VisiQuest that can be used for building and verifying test suites, but the three following routines are especially useful.

- `ksegcmp` - Compare Two Data Objects
- `kstats` - Compute Statistics of Data Object
- `kprdata` - Print Data in Formatted ASCII

Brief explanations of these programs are given below. For more information, see the associated man pages.

### B.6.1. `ksegcmp` — Compare Two Data Objects

The `ksegcmp` program allows you to compare different aspects of two input data objects and receive the output in various forms. Data comparison options include the option to cast the input data to the same data type before comparing, and the option to allow a tolerance when comparing data values. Specifying a tolerance is useful when comparing floating point values, which may differ slightly on different platforms. The `ksegcmp` program allows you to compare segment level attributes, such as the size and type of the value data, and global attributes, such as the format storage type. All, or a specified subset of the data segments can be compared. `ksegcmp` provides several output options that are useful during the different phases of creating and using a test suite. In the earlier phases, the *print summary*, *print position, type and value of differing points*, and the *ASCII output file* options are quite useful. For the final, non-interactive test suite, the *silent* option, along with the *return exit status* option, are used.

### B.6.2. `kstats` - Compute Statistics of Data Object

The `kstats` program will compute the specified statistics of the input data object, and write out the results in either ASCII or binary format. The ASCII output is useful for verifying results when creating the test suite, but the binary output is more portable, so if a result is stored for later comparison, it should be stored in binary format. The binary output values can be printed using `kprdata -val -attr`, described below. `kstats` allows for an optional gating input, and allows the statistical processing unit to be defined as the whole data set, or as any

combination of *width*, *height*, *depth*, *time* and *elements*.

### B.6.3. kprdata - Print Data in Formatted ASCII

The `kprdata` program prints a binary file as ASCII. This program is useful for examining data object attributes and data values when creating test suites. The user can specify which segments of the data object to view. If a map exists, the value and map data can be printed separately, or the value data can be mapped through the map prior to printing.

## C. Running Test Suites

Tests can be run in the test directory by either directly executing the file

```
% test1.sh
```

or by typing "`kmake test`" in that directory. Since test suites are supported by `kmake`, they can be run in batch mode from a parent test directory. For example, running

```
% kmake test
```

from the `$TOOLBOX/testsuite` directory will cause all tests in all of the `testsuite` subdirectories to be run. The exit status of each test is echoed to the `tty`, and all other information, such as error messages and intermediate status information, is redirected to a `ktest.arch` file located in the associated test directory. The *arch* suffix is appended by the `kmake` system using the `$ARCH` variable.

Example produced by running `kmake test` from the `$DATAMANIP/testsuite/karith2` directory is shown below.

```
% kmake test
--- 'test' ---
karith2 (./test1.sh) ... passed
karith2 (./test2.sh) ... passed
karith2 (./test3.sh) ... passed
karith2 (./test4.sh) ... passed
```

The following is an excerpt of the output that would be redirected into the `ktest.arch` file.

```
% more ktest.solaris
running test1.sh for karith2 ... process (14145)
Passed: Test 1 --> add - 2 input: no masks, no maps
Passed: Test 2 --> add - 2 input: both have masks
Passed: Test 3 --> add - 2 input: both have masks
Passed: Test 4 --> add - 2 input: i1 has mask, i2 does not
Passed: Test 5 --> add - 2 input: i2 has mask, i1 does not
Passed: Test 6 --> add -1 input byte
Passed: Test 7 --> add - 1 input int
Passed: Test 8 --> add - 1 input float
Passed: Test 9 --> add - 1 input double with mask
Passed: Test 10 --> sub - 2 input: no masks, no maps
Passed: Test 11 --> mul - 2 input: no masks, no maps
Passed: Test 12 --> div - 2 input: no masks, no maps
Passed: Test 13 --> div - 2 input: no masks, no maps
Passed: Test 14 --> absdiff - 2 input: no masks, no maps
Passed: Test 15 --> hypot - 2 input: no masks, no maps
Passed: Test 16 --> hypot - 2 input: no masks, no maps
Passed: Test 17 --> min - 2 input: no masks, no maps
Passed: Test 18 --> max - 2 input: no masks, no maps
```

```
Passed: Test 19 --> ldexp - 2 input: no masks, no maps
Passed: Test 20 --> pow - 2 input: no masks, no maps
```

Tests that are expected to fail due to a pending bug, or those that should not pass, can be flagged by adding a \*.ign file to the test directory for the test(s) that should be ignored. For example, if test1.sh identifies a known bug, having a test1.sh.ign file in that same directory will produce the following output when running kmake test.

```
karith2 (test1.sh) ... failed (expected)
```

## D. CLUI Test Suites

CLUI test suites are used to test the functionality of a program via its Command Line User Interface and are written in a scripting language. The template and example given below are written in sh. For more examples, see the test suites provided with the Datamanip, Image and Matrix toolboxes.

### D.1. Test Script Template

The following template test script file is located in

```
$BOOTSTRAP/repos/templates/test.sh.
```

Copy the test script to your local testsuite directory, modify lines 7 & 8, and insert your tests at line 60 between the "Add tests below this" and "End of testsuite" comment lines.

The template script contains a function *report* which takes five arguments: the test number, the data object generated by the test, a data object containing the known result to compare against, a string describing the test, and the tolerance to use when comparing object data values. The report function performs the comparison, prints the result of the test, and sets the exit status variable.

```
#!/bin/sh

# Replace the oname in the following line with the name of the program
# you are testing.

object_name="oname"

# -----
# ADD TESTS BETWEEN the "Add tests below this" and "End of test
# suite" comment lines (line 59).
# -----

file_name=`echo $0 | sed -e 's/^.*/\`
echo "running "$file_name" for "$object_name" ... process ($$)"

# report utility
# -----
# Usage: report test_num test_obj cntrl_obj test_descr tol
#
# The report function uses ksegcmp to compare the test output
# data object ($2) against the correct result ($3), and reports
# success or failure of the test. If the result of ksegcmp is
# false, status is set to 1.
#
# Input Arguments: test_num - test number
```

```

#           test_obj - the object generated by the test
#           cntrl_obj - known result to compare against
#           test_descr - a string describing the test
#           tol - tolerance allowed when comparing object
#                   data values. Recommended tolerance 1e-5
# -----
report()
{
    test_num="$1"
    test_obj="$2"
    cntrl_obj="$3"
    test_descr="$4"
    tol="$5"

    ksegcmp -i1 "$test_obj" -i2 "$cntrl_obj" -tol "$tol" -sat 1 -s -rt 1 -all
    if [ $? = 0 ]
    then
        echo "Passed:  Test  $test_num --> "$test_descr""
    else
        echo "FAILED:  Test  $test_num --> "$test_descr""
        status=1
    fi
}

# Initialize status to be success (ie. 0). Upon failure, status
# is initialized to 1. Set the tmpdir variable. Script will
# use /tmp if $TMPDIR is not defined.
status=0
tmpdir=${TMPDIR:-/tmp}
BINARY_PATH=${BINARY_PATH:-`kecho -tb toolbox -echo binpath`}

# -----
# Add tests below this
# -----

# -----
# End of test suite
# -----

exit $status

```

## D.2. Test Script Example

The following is a commented example script (example.sh) that can be found in the `$/DATAMANIP/test-suite/example` directory. This example was written using the template script described in the previous section, and can be run by going into the `$/DATAMANIP/testsuite/example` directory and typing

```

% example.sh

or

% kmake test

```

There are three example tests in this example test script. *Example 1* demonstrates using the ASCII import routines to build test and answer data sets. *Example 2* demonstrates using the kecho program to determine the path to a toolbox containing test data. The karith2 program is tested using a file in the `$/DATAMANIP/data/testdata` directory, and the result is compared against a file containing the correct result. The



datamanip toolbox data/testdata directory contains a set of very small files that are easy to verify using kprdata. *Example 3* demonstrates the use of kstats to verify the results of a program. The mean of the test result is computed using kstats and is compared against the known mean value. Computing the statistics is useful when testing larger data sets, and when testing programs where the results from run to run may vary, for example, in noise generation routines.

```

#!/bin/sh

# Replace the oname in the following line with the name of the program
# you are testing.

object_name="karith1 & khisto"

# -----
# ADD TESTS BETWEEN the "Add tests below this" and "End of test
# suite" comment lines (line 59).
# -----

file_name=`echo $0 | sed -e 's/^.*/'`
echo "running "$file_name" for "$object_name" ... process ($$)"

# report utility
# -----
# Usage: report test_num test_obj cntrl_obj test_descr tol
#
# The report function uses ksegcmp to compare the test output
# data object ($2) against the correct result ($3), and reports
# success or failure of the test. If the result of ksegcmp is
# false, status is set to 1.
#
# Input Arguments: test_num - test number
#                   test_obj - the object generated by the test
#                   cntrl_obj - known result to compare against
#                   test_descr - a string describing the test
#                   tol - tolerance allowed when comparing object
#                          data values. Recommended tolerance 1e-5
# -----
report()
{
    test_num="$1"
    test_obj="$2"
    cntrl_obj="$3"
    test_descr="$4"
    tol="$5"

    ksegcmp -i1 "$test_obj" -i2 "$cntrl_obj" -tol "$tol" -sat 1 -s -rt 1 -all
    if [ $? = 0 ]
    then
        echo "Passed: Test $test_num --> "$test_descr""
    else
        echo "FAILED: Test $test_num --> "$test_descr""
        status=1
    fi
}

# Initialize status to be success (ie. 0). Upon failure, status
# is initialized to 1. Set the tmpdir variable. Script will
# use /tmp if $TMPDIR is not defined.

```

```

status=0
tmpdir=${TMPDIR:-/tmp}

# -----
# Add tests below this
# -----

# -----
# Example 1
# This example demonstrates using the ascii import routines to
# build test and answer data sets. The karith1 -abs option is
# being tested for float data. For examples of using the ascii
# import routines to build more complex data objects, see
# test1.sh for knormal in the datamanip toolbox. Appending $$
# (process number) to the temporary file names ensures that
# tests won't collide, and is also useful for finding
# testsuites that are leaving files behind.
# -----

# Set the test number and the description of the test
test_num=1
descr="Example 1: absolute value of float data"

# Create a temporary ascii file containing the test data values
cat <<EOF > $tmpdir/asc.$$
-1 1
57 -1e20
65536 -1.12345e-38
EOF

# Import the data as a 2x1x1x1x3 float data set using kasc2val
kasc2val -i1 $tmpdir/asc.$$ -wsize 2 -esize 3 -type float -o $tmpdir/data.$$

# Perform karith1 operation
karith1 -i $tmpdir/data.$$ -o $tmpdir/out.$$ -abs

# Create a temporary ascii file containing the result values
cat <<EOF > $tmpdir/asc.$$
1 1
57 1e20
65536 1.12345e-38
EOF

kasc2val -i1 $tmpdir/asc.$$ -wsize 2 -esize 3 -type float -o $tmpdir/result.$$

# Compare the results of the karith1 operation against the
# correct results using the report utility
report $test_num "$tmpdir/out.$$" "$tmpdir/result.$$" "$descr" "1e-5"

# Remove the temporary files created by this test
rm -f $tmpdir/out.$$ $tmpdir/result.$$ $tmpdir/asc.$$
rm -f $tmpdir/data.$$

# -----
# Example 2
# This example demonstrates using the kecho program to determine
# the path to the datamanip toolbox. The karith2 program is
# tested using a file in the $DATAMANIP/data/testdata directory,
# and the result is compared against a file containing the

```

```

# correct result. The datamanip toolbox data/testdata
# directory contains a set of very small files that are easy to
# verify using kprdata.
# -----

# Set the test number and the description of the test
test_num=2
descr="Example 2: data multiplied, then divided by 2"

# set path for test data
tbpath=`kecho -tb DATAMANIP -echo path`
dpath="$tbpath/data/testdata"
BINARY_PATH=${BINARY_PATH:-`kecho -tb DATAMANIP -echo binpath`}

# Perform test operations
karith2 -i1 $dpath/3x4.float.viff -o $tmpdir/out.$$ -mul -real 2
karith2 -i1 $tmpdir/out.$$ -o $tmpdir/out.$$ -div -real 2

# Compare data against the known result
report $test_num "$tmpdir/out.$$" "$dpath/3x4.float.viff" "$descr" "1e-5"

# Remove temporary files created by this test
rm -f $tmpdir/out.$$

# -----
# Example 3
# This example demonstrates using kstats to verify the results
# of a program. The mean of the result is computed using
# kstats and is compared against the known mean value. In this
# example, only the mean is computed. It is often easier to
# compute all of the statistics (kstats -all), and save the
# verified results in a binary file (-o) in the test suite
# subdirectory. This file can then be used directly when
# calling the report utility.
#
# Computing the statistics is useful when testing larger data
# sets, and when testing programs where the results from run to
# run may vary, for example, in the noise generation routines.
#
# For more test suite examples that use kstats, see the knoise,
# kgnoise, and kstats testsuites in the datamanip toolbox.
# -----

# Set the test number and the description of the test
test_num=3
descr="Example 3: ball.xv histogram equalized"

# Histogram equalize ball.xv
khistops -equalize -i image:ball -o $tmpdir/out.$$ -whole

# Calculate the mean of the result
kstats -i $tmpdir/out.$$ -o $tmpdir/mean.$$ -mean

# Create an ascii file containing the known mean value
cat <<EOF > $tmpdir/asc.$$
128.971
EOF

# Compare the test mean against the known mean

```

```

report $test_num "$tmpdir/mean.$$" "$tmpdir/asc.$$" "$descr" "1e-3"

# Remove the temporary files created by this test
rm -f $tmpdir/asc.$$ $tmpdir/out.$$ $tmpdir/mean.$$

# -----
# End of test suite
# -----

exit $status

```

## E. Library Test Suites

*Library test suites* consist of a complete functioning C program, one or more baseline data sets or output logs, and one or more shell scripts which will run the test. Typical C test programs consist of code which exercises a particular library call or set of library calls and prints to `stdout` some manner of output illustrating whether the library calls functioned correctly or not. When creating the test suite, the output produced from a successful run of the compiled C program should be captured into a file. The file should typically be named *baseline*, although any name may be used as long as the test script is modified accordingly. Avoid printing floating-point and double numbers as there may be precision problems between different machines.

When the test suite is run using `kmake test`, the C program will be compiled from scratch into an executable named *ktest*, and the shell script will be run. A typical shell script will run the *ktest* program while capturing `stdout` into a file. This file is then compared with the baseline file using the UNIX *cmp* command. If the files match, then the test is considered successful.

The template library test script file shown below is located in

```
$BOOTSTRAP/repos/templates/lib.sh
```

For examples of library test suites, see the test suites provided in the `dataserv` toolbox.

```

#!/bin/sh

# -----
# This script must be named test.sh
# -----

# run the compiled program and compare the output against a baseline
ktest > test.out
cmp -s test.out baseline

# Assign status according to the result of the comparison.
# On failure, status is set to 1.
if [ $? = 0 ]; then
    status=0
else
    status=1
fi

# -----
# Clean up any left-over files your .c file may have generated
# -----
rm -rf test.xv test.out

# -----

```

```
# End of test suite
# -----
exit $status
```

## F. The ktestutils Library

The bootstrap toolbox contains a small library called `ktestutils` which provides a number of utility routines helpful when writing testsuites for a library object. Testsuites objects are automatically linked against this library, but all other software objects are not.

To use the library, your testsuite program should include the following line:

```
#include <ktestutils/ktestutils.h>
```

This line should appear *after* the toolbox include file. So for a testsuite in toolbox `pandora`, the top of a test-suite program would be:

```
#include <pandora.h>
#include <ktestutils/ktestutils.h>
```

The body of your testsuite should be enclosed with calls to `ktest_begin` and `ktest_end`. The former function takes two parameters: an identifier for the testsuite, and a short string describing what is being tested. For example, the `kcms` library has a testsuite for the function `kcms_legal_identifier`:

```
ktest_begin("35.kcms", "kcms_legal_identifier");

... test cases ...

ktest_end();
```

A testsuite may contain a number of *test cases*. Each test case should be enclosed with calls to `ktest_case()` and `ktest_result()`. The former function takes a single parameter: a short string description of the test case, and the latter function takes a single parameter that specifies whether the test was successful. Legal values for this parameter are `KTEST_FAIL` and `KTEST_PASS`. For example:

```
ktest_case("toolbox name is empty string");

... test case ...

ktest_result( conditional ? KTEST_PASS : KTEST_FAIL );
```

### F.1. The ktestutils API

The functions provided are:

- `ktest_begin()` - start a testsuite
- `ktest_end()` - end a testsuite
- `ktest_case()` - introduce a test case
- `ktest_result()` - end test case and register the result

## F.1.1. `ktest_begin()` — *start a testsuite*

### Synopsis

```
void
ktest_begin(
    kstring  testsuite_id,
    kstring  testing)
```

### Input Arguments

`testsuite_id`

The identifier for the testsuite. This is typically the name of the directory which contains the testsuite, but it doesn't have to be.

`testing`

A short text string (60 characters at most) which describes what the testsuite is testing.

### Description

This function is used to introduce a testsuite.

For example, a **klibc** testsuite for the function `kstring_copy()` may start with the line:

```
ktest_begin("35.klibc", "kstring_copy()");
```

## F.1.2. `ktest_end()` — *end a testsuite*

### Synopsis

```
void
ktest_end(
    void)
```

### Description

This function is used to finish a testsuite. This marks the end of the testsuite log, and if any cases failed, then a line is printed to stdout giving the number of failures.

After printing out any information, `ktest_end` calls `kexit()` with the parameter `KEXIT_SUCCESS` if no cases failed, and `KEXIT_FAILURE` if one or more cases failed.

### F.1.3. `ktest_case()` — *introduce a test case*

#### Synopsis

```
void
ktest_case(
    kstring description)
```

#### Input Arguments

`description`  
A short description of the test case. The description should not exceed 60 characters.

#### Description

This function is used to introduce a test case within a testsuite. You should have called `ktest_begin()` before calling this function.

For example, when testing the function `kstring_copy()`, one case to test for is a NULL input string. This test case is introduced with the following line:

```
ktest_case("NULL input string");
```

### F.1.4. `ktest_result()` — *end test case and register the result*

#### Synopsis

```
void
ktest_result(
    int result)
```

#### Input Arguments

`result`  
The result of the test case. There are two legal values: `KTEST_PASS` if the test was successful, `KTEST_FAIL` otherwise.

#### Description

This function is used to mark the end of a test case, and register whether the test case passed or failed.

You should have called `ktest_case()` before calling this function to introduce the test case.

# Table of Contents

A. Introduction . . . . .	8-1
B. The Testing Environment . . . . .	8-1
B.1. Test Suite Directory Structure . . . . .	8-2
B.2. Test Suite Pmakefiles . . . . .	8-2
B.3. Requirements . . . . .	8-3
B.4. Test Data . . . . .	8-3
B.5. Suggestions for Writing Test Suites . . . . .	8-3
B.6. Useful Routines for Testing . . . . .	8-4
B.6.1. ksegcmp — Compare Two Data Objects . . . . .	8-4
B.6.2. kstats - Compute Statistics of Data Object . . . . .	8-4
B.6.3. kprdata - Print Data in Formatted ASCII . . . . .	8-5
C. Running Test Suites . . . . .	8-5
D. CLUI Test Suites . . . . .	8-6
D.1. Test Script Template . . . . .	8-6
D.2. Test Script Example . . . . .	8-7
E. Library Test Suites . . . . .	8-11
F. The ktestutils Library . . . . .	8-12
F.1. The ktestutils API . . . . .	8-12
F.1.1. ktest_begin() — <i>start a testsuite</i> . . . . .	8-13
F.1.2. ktest_end() — <i>end a testsuite</i> . . . . .	8-13
F.1.3. ktest_case() — <i>introduce a test case</i> . . . . .	8-14
F.1.4. ktest_result() — <i>end test case and register the result</i> . . . . .	8-14



**This page left intentionally blank**

## **Chapter 9**

# **CVS and VisiQuest**

## **Revision Control of Toolboxes and Software objects**



# Chapter 9 - CVS and VisiQuest

## A. Introduction

CVS (Concurrent Versions System) is an Open Source revision control system which provides functionality similar to the GNU RCS revision control system. CVS extends RCS from single files to hierarchical collections of directories and their contents. CVS provides a number of functions that are extremely useful in managing software throughout its life cycle. AccuSoft Corporation has adopted CVS for revision control of VisiQuest. Included with the VisiQuest release are VisiQuest-specific CVS tools; these tools are "wrappers" around the cvs program that allow VisiQuest software objects and toolbox objects to be maintained as entities within CVS. We strongly suggest that Concurrent Versions System (CVS) be used in conjunction with the VisiQuest software development environment. This chapter explains how to use CVS with the VisiQuest system.

It should be noted that it is important for you to become familiar with CVS before using it with VisiQuest, because there are nuances to using CVS that are affected when using the VisiQuest CASE tools. This chapter assumes a working knowledge of CVS, and only documents the nuances of using CVS with the VisiQuest CASE tools. There are many sources of information on CVS. The CVS manual and FAQ are good sources of information with respect to using and setting up CVS. The manual comes with the CVS source code in post script form, and all other information can be found on the web site <http://www.cyclic.com>. A pdf version is included as an appendix to the Toolbo Programming manual. CVS and configuration management in general can be found at <http://www.loria.fr/~molli/> and <http://www.cyclic.com>.

## B. The VisiQuest CVS tools

The CVS support system in VisiQuest consists of five programs: **kcvsadd**, **kcvscommit**, **kcvsrm**, **kcvsstatus**, and **kcvsupdate**. These programs are all Perl 5 scripts, that take the necessary steps to add, commit, remove, status, and update files in a VisiQuest toolbox, without explicitly having to know how the files in a toolbox interact. They help keep the database and informational files for a toolbox up to date as files and software objects change.

### B.1. **kcvsadd** — *cvs add a software object, file, or directory*

#### Object Information

Category:	<i>Software Management</i>	Subcategory:	<i>Configuration Management</i>
Operator:	<i>kcvsadd</i>	Object Type:	<i>Script</i>
In Cantata:	<i>No</i>		

#### Description

**kcvsadd** is a wrapper around the cvs add functionality. It allows users to add directories, files, and software objects to the CVS repository. In order for this program to work, the toolbox you tell it to

operate in, must also be under CVS. To put a toolbox under cvs, use the cvs import command. When using cvs import, you should avoid importing the bin, config, lib, mach, and include/<oname> directories by moving them out of the way before calling cvs import. These directories contain duplicate copies of other files, binaries, or site specific configuration files that should not be maintained under CVS. Once a toolbox is imported, move these directories back to their original places.

When adding a file or directory, just specify the following command to add it to the repository:

```
% kcvsadd -file /path/to/file_or_dir
```

It defaults to "-file ." if no parameters are specified on the command line. When a new software object is created, you should specify the following command to add it to the repository:

```
% kcvsadd -tb toolbox -oname oname
```

This will do cvs add for all the directories and files in the software object, plus to adds for the extra toolbox maintenance files that can be created when a new software object is run.

## Command Line Arguments

```
=====
Usage for kcvsadd:
% kcvsadd

[-tb]      (string) name of toolbox
           (default = {none})
[-oname]   (string) object name
           (default = {none})
[-file]    (string) file or directory
           (default = {none})
[-ofile]   (string) File to which to write output from cvs
           (default = {none})
=====
```

## **B.2. kcvsrm** — *Remove a file, software object, or toolbox object and updates its CVS repository*

### Object Information

Category:	<i>Software Management</i>	Subcategory:	<i>Configuration Management</i>
Operator:	<i>kcvsrm</i>	Object Type:	<i>Script</i>
In Cantata:	<i>No</i>		

### Description

This perl script mirrors the functionality of krm with respect to removing files, toolboxes, or software object. Additionally, kcvsrm will update an associated CVS repository if the item being deleted is under CVS control.

This program can be used to remove:

- (1) a toolbox
- (2) a program or library object from a toolbox
- (3) a file from a program or library object

When a toolbox is removed, the directory structure of the toolbox and all the contents therein are deleted from the disk. Then, the reference to the deleted toolbox is removed from the Toolboxes file.

When a program or library object is removed, all the source, documentation, miscellaneous, and binary files associated with the software object and the directory structure in which they are contained must be removed from the disk. Then, removes the entry referencing the deleted software object is deleted from the toolbox configuration file.

When a file is removed, the process is simple: the file is removed.

This routine will always prompt whether or not you really wanted to remove the item unless you specify the *[-force]* flag.

Once the removal is accomplished, the script will use cvs commands as appropriate to ensure that the CVS repository correctly reflects that something has been deleted. This includes updating toolbox configuration database files, object cms database files, and Imakefiles in parent directories.

Kcvstrm uses krm with -force to handle the actual removal, so it should be noted that removing routines with an associated library routine will always assume that the associated lkroutine.c file should be removed along with the kroutine being deleted. Krm makes this assumption when force mode is specified, and kcvstrm has to call krm in force mode to avoid krm's prompting.

Finally, kcvstrm will not attempt to modify the CVS repository unless it can figure out if a file is under CVS control. Thus, it will correctly remove files and objects that are not under CVS control. If told to remove a file, software, or toolbox object not under CVS control, kcvstrm will just call krm to remove the specified object and exit.

## Examples

The following example removes a file "README".

```
% kcvstrm -i README
```

```
Operation: delete a file
```

```
File:  README
```

```
Please confirm.
```

```
Do you want to continue with the removal? (Yes/No) [No]: y  
Removing 'README'
```

Updating repository to reflect removal of 'README'

The next two examples removes a kroutine and a kroutine with associated library file from a toolbox.

```
% kcvstrm -tb worktb -oname kcp
```

Operation: delete a software object from toolbox

Toolbox: WORKTB

Object: kcp

Please confirm.

Do you want to continue with the removal? (Yes/No) [No]: y

Removing object 'kcp'

Updating CVS repository '/path/to/cvs/repository'

Checking out a temporary copy of 'worktb/objects/kroutine/kcp'

Performing CVS remove and committing changes to repository

Updating repos/db directory of 'WORKTB'

Updating repos/config directory of 'WORKTB'

Updating Imakefile in parent directory of 'kcp'

```
% Operation: delete a software object from toolbox
```

Toolbox: WORKTB

Object: karith2

Please confirm - the object has an associated l\*.c file in 'kdatamanip'.

This file will also be deleted if you selected Yes.

Do you want to continue with the removal? (Yes/No) [No]: y

Removing object 'karith2'

Updating CVS repository '/research/timelord/TARDIS'

Checking out a temporary copy of 'worktb/objects/kroutine'

Performing CVS remove and committing changes to repository

Updating repos/db directory of 'WORKTB'

Updating repos/config directory of 'WORKTB'

Updating Imakefile in parent directory of 'karith2'

Updating associated library 'kdatamanip' in 'WORKTB'

This final example removes the WORKTB toolbox.

```
% kcvsrm -tb bogus
```

Operation: delete a toolbox

Toolbox: BOGUS

Please confirm - the toolbox is not empty!

Do you want to continue with the removal? (Yes/No) [No]: y

Removing toolbox 'BOGUS'

Updating CVS repository '/research/timelord/TARDIS'

Checking out a temporary copy of 'bogus'

Performing CVS remove and committing changes to repository

## See Also

krm(1), cvs(1)

## Restrictions

When removing a file that also belongs to a software or toolbox object, kcvsrm cannot always detect that it belongs to an object, so the user should use composer to ensure that the reference is removed from the cms database.

## References

Version Management with CVS

## Command Line Arguments

```
=====
Usage for kcvsrm:
% kcvsrm

[-tb]      (string) toolbox to be removed (or toolbox from which to remove object
           (default = {none})
[-oname]   (string) software object to be removed
           (default = {none})
[-i]       (infile) file to be removed
           (default = {none})
[-cvsout]  (infile) file for cvs output (default /dev/null)
           (default = {none})
[-force]   (flag) force removal?
=====
```

### **B.3. kcvscommit** — *commit an object, list of objects, or toolbox*



## Object Information

Category: *Software Management*                      Subcategory: *Configuration Management*  
Operator: *kcvscommit*                                      Object Type: *Script*  
In Cantata: *No*

## Description

The **kcvscommit** script is a wrapper around the cvs commit functionality. It helps users commit files in their toolboxes. The program has one required parameter '-m', which corresponds to the cvs commit -m parameter. It should be a short description of the change which is being committed. kcvscommit can operate in three modes: toolbox mode, software object mode, and file or directory mode.

In toolbox mode, one or more toolboxes can be specified to be committed as follows:

```
% kcvscommit -tb tb1,tb2,tb3,tb4 -m "Doing a multi-toolbox commit"
or for all toolboxes:
```

```
% kcvscommit -alltb -m "Doing a big commit"
```

where all toolboxes refers to the list specified in your tools.ini file. If no tools.ini file exists, it gets the complete list of toolboxes in your Toolboxes files.

In software object mode, you specify a single software object to be committed as follows:

```
% kcvscommit -tb toolbox -oname object_name -m "Commit of TOOLBOX::object-name"
```

In file or directory mode, you specify the file or directory to commit with the following command:

```
% kcvscommit -file /path/to/dir_or_file -m "Commit of dir_or_file"
```

If none of -tb, -alltb, or -file is specified, it assumes "-file .".

## Command Line Arguments

```
=====
Usage for kcvscommit:
% kcvscommit
  -m          (string) logging message

  [-ofile]   (string) name of output file
              (default = {none})
  [-file]    (string) name of file to commit
              (default = {none})
  [-tb]      (string) toolbox
              (default = {none})
  [-r]       (string) revision of module
              (default = {none})
  [-oname]   (string) name of software object
```

(default = {none})

## B.4. **kcvsupdate** — *make clean/update repository*

### Object Information

Category:	<i>Software Management</i>	Subcategory:	<i>Configuration Management</i>
Operator:	<i>kcvsupdate</i>	Object Type:	<i>Script</i>
In Cantata:	<i>No</i>		

### Description

The **kcvsupdate** script is a wrapper for the cvs update functionality. It helps users update files in their toolbox from the repository. This program operates in three modes: toolbox mode, software object mode, and file or directory mode.

In toolbox mode, one or more toolboxes can be specified to be updated as follows:

```
% kcvsupdate -tb tb1,tb2,tb3,tb4  
or for all toolboxes:
```

```
% kcvsupdate -alltb  
where all toolboxes refers to the list specified in your tools.ini file. If no tools.ini file exists, it gets the complete list of toolboxes in your Toolboxes files.
```

In software object mode, you specify a single software object to be updated as follows:

```
% kcvsupdate -tb toolbox -oname object_name
```

In file or directory mode, you specify the file or directory to update with the following command:

```
% kcvsupdate -file /path/to/dir_or_file
```

If none of -tb, -alltb, or -file is specified, it assumes "-file .".

### Command Line Arguments

```
=====
```

Usage for kcvsupdate:

```
% kcvsupdate  
  
[-tb] (string) toolbox
```

```

        (default = {none})
[-oname] (string) software object
        (default = {none})
[-r]     (string) revision of module
        (default = {none})
[-outdir] (string) if specified, kcvupdate creates output file $outdir/$tb.kcvupdate.$dat
        (default = {none})
=====

```

## B.5. **kcvstatus** — *check cvs status of a software object or toolbox*

### Object Information

Category:	<i>Software Management</i>	Subcategory:	<i>Configuration Management</i>
Operator:	<i>kcvstatus</i>	Object Type:	<i>Script</i>
In Cantata:	<i>No</i>		

### Description

The **kcvstatus** script is a wrapper around the cvs status functionality. It helps users check the status of files in their toolbox. This program operates in three modes: toolbox mode, software object mode, and file or directory mode.

In toolbox mode, one or more toolboxes can be specified to check the status as follows:

```
% kcvstatus -tb tb1,tb2,tb3,tb4
or for all toolboxes:
```

```
% kcvstatus -alltb
where all toolboxes refers to the list specified in your tools.ini file. If no tools.ini file exists, it gets the complete list of toolboxes in your Toolboxes files.
```

In software object mode, you specify a single software object to be checked as follows:

```
% kcvstatus -tb toolbox -oname object_name
```

In file or directory mode, you specify the file or directory to be checked with the following command:

```
% kcvupdate -file /path/to/dir_or_file
If none of -tb, -alltb, or -file is specified, it assumes "-file .".
```

## Command Line Arguments

```
=====
Usage for kcvstatus:
% kcvstatus

[-tb]      (string) toolbox
           (default = {none})
[-oname]   (string) software object name
           (default = {none})
[-file]    (string) name of a file
           (default = {none})
[-ofile]   (string) File to which to print output
           (default = {none})
=====
```

## C. Using CVS and the VisiQuest CVS Tools: A Tutorial

This section is designed to familiarize you with CVS, its purpose and use, and how to use the VisiQuest kcvs tools to assist you with CVS revision control. This section is for UNIX users only. <sup>1</sup>

CVS is a version control system. Using it, you can record a history of file updates. It provides a way to manage changes to a collection of software files when many people are working on them all at once. CVS also allows branching off a version of the software a group may be working on, i.e. create a whole new directory tree which starts with a copy of the current source tree, and then begins to keep the branches' own versions in a separate area.

CVS allows you to retrieve old versions of the files as well, allowing you to back out of changes if necessary, or even just to compare new versions with older ones. It stores all files in a centralized repository, containing directories and files, in an arbitrary tree.

File versions are each stored with their version numbers - one version and version number for each committed change.

The remainder of this section covers the following issues:

1. Create a CVS repository
2. Commit changes to an object or file
3. Show and resolve conflicting changes
4. Update a toolbox, object or file
5. Checkout a toolbox, object, file or files
6. Add new objects or files to the repository
7. See how multiple changes are reflected in a subsequent update

By convention in this section, anything in square brackets, [], is to be replaced by the corresponding name. For instance, if the directory for CVSROOT is supposed to be /usr/local/cvsroot, and the directions say:

Type: `setenv CVSROOT [dirname]`

Then you would enter:

```
setenv CVSROOT /usr/local/cvsroot
```

---

<sup>1</sup> Portions of this text are taken from the "Version Management with CVS" document for CVS 1.10 copyright 1992, 1993]

Before following the instructions in this tutorial, you should have created your own toolbox, and created at least one object, preferably two.

## C.1. Creating a CVS repository

1. To create a CVS repository, you must first tell cvs what directory to put the repository in by setting the CVSROOT environment variable.

If you are using csh and tcsh, type:

```
% setenv CVSROOT /[your full home directory path]/repository
```

If you are using sh or bash, type:

```
% CVSROOT /[your home directory name]repository  
% export CVSROOT
```

2. Make a repository directory in your own home directory; from your home directory, type:

```
% mkdir repository
```

3. Make the CVSROOT repository directory:

```
% mkdir repository/CVSROOT
```

4. Now make the CVSOUT directory in your home directory. This is what is listed in the tools.ini file in the \$HOME/.kri/{PRODUCT\_DIR} directory.

5. Now create the repository in the CVSROOT directory, like so:

```
% cvs init
```

You should see an output similar to this one:

```
Creating the /research/guest/repository//CVSROOT directory  
Creating a simple /research/guest/repository//CVSROOT/modules file  
Creating a simple /research/guest/repository//CVSROOT/logininfo file  
Creating a simple /research/guest/repository//CVSROOT/commitinfo file  
Creating a simple /research/guest/repository//CVSROOT/rcsinfo file  
Creating a simple /research/guest/repositoary//CVSROOT/editinfo-file  
Creating a simple /research/guest/respository//CVSROOT/rcs template file  
Creating a simple /research/guest/repository//CVSROOT/checkout-list file  
Copying the new version of 'commit_prep' to /research/guest/repository//CVSROOT for you  
Copying the new version of 'log_accum' to /research/guest/repository//CVSROOT for you  
Copying the new version of 'cln_hist' to /research/guest/repository//CVSROOT for you  
Enabling CVS history logging  
All done! Running 'mkmodules' as my final step
```

Note: cvs init is careful never to overwrite any exiting files in the repository, so no harm is done if you run cvs init on an already set-up

directory.

6. Add the entire directory structure of your work toolbox to the repository. First, import the directory structure to the repository:

```
% cd [toolbox dir]
% cvs import -m "Import of my toolbox" [repository
% dirname]
% [vendor tag] [release tag]
```

For vendor tag, type "VisiQuest", and for release tag, type KP2001 (without the quotes). In this case, the repository dirname is the name of your toolbox. This is the name under which all the directories under the toolbox will be stored inside the repository. Change directories to the repository directory, type `ls *` and you will see how this is done. You should see an output similar to this one:

```
cvs import: Importing /research/guest/junk/bin
cvs import: Importing /research/guest/junk/include
cvs import: Importing /research/guest/junk/include/junk.h
cvs import: Importing /research/guest/junk/lib
cvs import: Importing /research/guest/junk/repos
cvs import: Importing /research/guest/junk/repos/license
cvs import: Importing research/guest/junk/repos/config

No conflicts created by this import
```

## C.2. Checking out a copy of something in the repository

Try moving your original directory, and then using `cvs checkout` to get it back:

```
% cd . .
% mv [toolbox dir] [toolbox dir].orig
% cvs checkout [repository dirname]
% ls -R [toolbox dir]
```

CVS should have restored all the original files in your toolbox directory. Again, in this case, repository dirname and toolbox dir are the same thing. You will also notice that in the checked-out version, there are CVS directories in the base & subdirectories of your toolbox. These are necessary for committing changes, and will not appear until you checkout the stored directory structure, as you just did.

If you had wanted to store more than one copy of your toolbox in the repository, you could have imported the toolbox into one subdirectory under the repository directory, and then imported the other copy into a different subdirectory, as in this example:

```
% cvs import -m "This is the 1st copy of the junk tb" copy1/junk VisiQuest training
```

Then:

```
% cvs import -m "This is the 2nd copy of the junk tb" copy2/junk VisiQuest training
```

WARNING: make sure you are in the directory you wish to have imported, because CVS will import everything in./ and below; don't make the mistake of importing for example, your entire home directory and its contents (like the author did).

### C.3. Committing changes to an object to the CVS repository (kcvscscommit)

**kcvscscommit** is the VisiQuest tool that should be used for committing changes to files in toolbox objects and software objects.

1. Make a change to an existing object inside your toolbox - in the source code, write:

```
kprintf("This is my change.");
```

2. Enter:

```
% kcvscscommit -tb [toolbox name] -oname [object name] -m "Made a change"
```

OR Enter

```
% kcvscscommit -file [file name you just changed] -m "your comment here"
```

You should see a message like the following:

```
Checking in ktest.c;  
/research/guest/repository/training/objects/kroutine/ktest/src/  
ktest.c,v < ktest.c  
new revision: 1.3; previous revision: 1.2  
done
```

In this case, both commands commit the same change. If you were to make changes to more than one file in an object, you need to use the second command. If you were to make changes to more than one object in a toolbox, omit the "-oname" argument, and kcvscscommit will serve the entire toolbox for changes.

### C.4. Checking the status of an object: kcvstatus

**kcvstatus** is a VisiQuest tool for checking the CVS status of files, toolbox objects and software objects.

Kcvstatus prints a line for each directory it is going to check, indicating the directory pathname. After the directory has been checked, it prints a line repeating the directory name and saying that it is done. No other output indicates that the directory contents are up to date. If there are locally modified files, unknown files, or files with a conflict, that will be indicated. For example, here is the kcvstatus output of a library object:

```
% kcvstatus -tb imagine -oname xvlan  
checking status in /research/vision/oasis/imagine/include  
status in /research/vision/oasis/imagine/include done  
checking status in /research/vision/oasis/imagine/repos/config/mk  
status in /research/vision/oasis/imagine/repos/config/mk done  
checking status in /research/vision/oasis/imagine/repos/db  
status in /research/vision/oasis/imagine/repos/db done  
checking status in /research/vision/oasis/imagine/repos/config
```



```

status in /research/vision/oasis/imagine/repos/config done
checking status in /research/vision/oasis/imagine/objects/library/xvlang/..
status in /research/vision/oasis/imagine/objects/library/xvlang/.. done
checking status in /research/vision/oasis/imagine/objects
status in /research/vision/oasis/imagine/objects done
checking status in /research/vision/oasis/imagine/objects/library/xvlang
cvs server: Examining .
Unknown file: src/junk
cvs server: Examining src
File: ToolboxList.c      Status: Locally Modified
File: ToolboxMenu.c     Status: Locally Modified
File: Loop.c            Status: Needs Patch
status in /research/vision/oasis/imagine/objects/library/xvlang done

```

In this case, `kcvsstatus` reminds us that we need to decide whether or not to commit the changes to "ToolboxList.c" and "ToolboxMenu.c" (perhaps the modifications represent a bug fix). It tells us that someone else has checked in a newer version of "Loop.c", and that we need to run "kcvsupdate" to get the updated "Loop.c" file. It also reminds us to remove the unwanted "junk" file.

## C.5. Resolving conflicting versions of a file

Suppose two people have checked out copies of the same toolbox and each made different changes to the same file. In this case, CVS will "merge" the changes. For example, if one developer has changed lines 5 - 10, and the second developer has changed lines 10 - 20, CVS will automatically incorporate both changes, and indicate that it has merged the differences; it will give no warnings or mention of a conflict.

However, suppose the two developers have both changed the the same line(s) of the same file. When it is time to commit those changes, there will be a *CVS conflict*; CVS recognizes conflicts in these two versions, and will alert you to this problem. It is your responsibility to resolve conflicts.

1. Simulate this problem by checking out a copy of the toolbox into a subdirectory called newcopy. From your home directory type:

```

% mkdir newcopy
% cd newcopy
% cvs checkout [toolbox name]

```

2. Make a change in the original toolbox to the source code in one of the objects; it can just be another simple `kprintf`, or change the original print message from part 5. Commit this change. Now make a change to an object in the checked-out copy of the same file. Then type:

```

% cvs update

```

Note that you cannot use `kcvsupdate` here because the checked-out copy has not been made a toolbox. You will get a message similar to this one:

```

cvs update: Updating RCS file: /research/guest/repository/training/objects/kroutine/1

```

```
retrieving revision 1.2
retrieving revision 1.3
Merging differences between 1.2 and 1.3 into ktest.c
rcsmerge: warning: conflicts during merge cvs update:
conflicts found in ktest.c
```

If you are working from inside an actual toolbox, you can run `kcvsstatus` on the object, and you will get an output indicating in which file the conflict lies. You can also view the file itself and see what CVS has done to point out the conflict.

3. Resolve the conflict by deciding how to integrate the first change with the second one. Commit the change:

```
% kcvscommit -file [filename you just changed] -m "Resolved conflict"
```

4. Then go to the toolbox directory and enter:

```
% kcvsupdate -tb [toolbox name] -oname [object name]
```

You should get this type of message:

```
Updating in /research/guest/junk/objects/kroutine/ktest
```

View the file to see that the changes have been correctly updated.

## C.6. Adding a new object or file to the repository: `kcvsadd`

New toolbox or software objects, as well as new files, are added to the CVS repository using `kcvsadd`.

1. Create a new kroutine using Craftsman, and use `kcvsadd` to add the new object to the repository, like so:

```
% kcvsadd -tb [toolbox name] -oname [new kroutine name]
```

You should see output like the following:

```
cvs add: scheduling file 'internals.h' for addition
cvs add: use 'cvs commit' to add this file permanently
cvs add: scheduling file 'Pmakefile' for addition
cvs add: use 'cvs commit' to add this file permanently
cvs add: scheduling file 'ltest.todo' for addition
```

```
Directory /research/guest/repository/objects/library/ltest
added to the repositoryDirectory /research/guest/repository
/training/objects/library/ltest/src added to the repositoryDirecory
/research/guest/repository/objects/library/ltest/info added to the
repositoryDirectory /research/guest/repository/objects/library/ltest/man added
to the repositoryDirectory
/research/guest/repository/objects/library/ltest/html
added to the repositoryDirectory
/research/guest/repository/objects/library/ltest/db
```

```
added to the repositoryDirectory
/research/guest/repository/include/ltest added to the repository.
```

2. Then commit the object:

```
% kcvscscommit -tb [toolbox name] -oname [new kroutine name]
-m "This is my new kroutine"
```

3. Now go to the second directory (the copy of the toolbox) and add the object like so:

```
% cvs update -Pd
```

The "P" option prunes empty directories, and the "d" option will create any directories that exist if they're missing from the directory.

## C.7. Deleting software objects with CVS: `kcvstrm`

The `kcvstrm` tool is used to delete directories that are currently under CVS. From the original toolbox, remove the kroutine from the CVS repository:

```
% kcvstrm -tb [toolbox name] -oname [kroutine name]
```

Respond to the prompt with "Yes" to continue with the removal. (Note that you must still commit the removal before the kroutine will be deleted from the repository).

## C.8. Committing and Updating Multiple Changes

When only the toolbox name is specified, `kcvscscommit` and `kcvscsupdate` will commit and update all files in a toolbox. When a toolbox name and an object name are specified, `cvscommit` and `kcvscsupdate` will commit and update all files in the software object. `Kcvscscommit` has a `[-file]` option that can be used to commit a single file.

- Use `composer` to add two files to one of the remaining software objects. Use your editor to edit a third file. Then use `kcvcsadd` and `kcvscscommit` to add and commit this file to the repository - from the `src/directory` of that kroutine, type:

```
% kcvcsadd -file [file1]
% kcvcsadd -file [file2]
% kcvscscommit -tb [toolbox name] -oname [object name] -m "Changed multiple things in tool"
```

Note that all of the changes to the software object were committed.

## Table of Contents

A. Introduction . . . . .	9-1
B. The VisiQuest CVS tools . . . . .	9-1
B.1. <code>kcvsadd</code> — <i>cvs add a software object, file, or directory</i> . . . . .	9-1
B.2. <code>kcvsrm</code> — <i>Remove a file, software object, or toolbox object and updates its CVS repository</i> . . . . .	9-2
B.3. <code>kcvscommit</code> — <i>commit an object, list of objects, or toolbox</i> . . . . .	9-5
B.4. <code>kcvsupdate</code> — <i>make clean/update repository</i> . . . . .	9-7
B.5. <code>kcvsstatus</code> — <i>check cvs status of a software object or toolbox</i> . . . . .	9-8
C. Using CVS and the VisiQuest CVS Tools: A Tutorial . . . . .	9-10
C.1. Creating a CVS repository . . . . .	9-11
C.2. Checking out a copy of something in the repository . . . . .	9-12
C.3. Committing changes to an object to the CVS repository ( <code>kcvscommit</code> ) . . . . .	9-13
C.4. Checking the status of an object: <code>kcvsstatus</code> . . . . .	9-13
C.5. Resolving conflicting versions of a file . . . . .	9-14
C.6. Adding a new object or file to the repository: <code>kcvsadd</code> . . . . .	9-15
C.7. Deleting software objects with CVS: <code>kcvsrm</code> . . . . .	9-16
C.8. Committing and Updating Multiple Changes . . . . .	9-16

**This page left intentionally blank**

**Chapter 10**

**User Interface Specification**

**Syntax and Details  
of the UIS file**



# Chapter 10 - User Interface Specification

## A. Introduction To The User Interface Specification (UIS) File

Kroutine, xvroutine, pane, and some script software objects have at least one UIS file. The UIS file is comprised of lines made up of fields in a strictly defined syntax. Each UIS line describes either a program argument on the CLUI, an item on the GUI, or both. The UIS was designed at a high level of abstraction; each type of line contains the information necessary to completely describe that type of item in the GUI, as well as in the CLUI. Thus, from a single UIS file for a particular application, the code for both the CLUI and the GUI may be generated directly. The UIS file is used for three distinct purposes in VisiQuest:

1. It defines the CLUI of all VisiQuest programs.
2. It defines the GUI of the program that is used when the program is run using the [-gui] option; this same GUI is used to integrate the program into the VisiQuest visual language.
3. For interactive X-Windows based applications (xvroutines), it defines the interactive GUI of the application.

For *kroutines*, a single UIS file will suffice for all purposes. *Xvroutines* often use two UIS files: one to define the CLUI of the program and to integrate it into VisiQuest, and another to define the more sophisticated independent GUI of the program. For *pane objects*, the UIS file provides the alternate interface to the base program of the pane object. For *script objects*, the UIS file is used to integrate the script into VisiQuest.

### A.1. UIS File Creation / Modification / Editing

The `guise` GUI design tool helps you create the desired UIS file(s) for your software objects and prototype the specified GUI(s). `guise` displays the GUI as specified in the UIS file in an active but non-functional state. `guise` is discussed in detail in Chapter 4 of the *Toolbox Programming Manual*.

`guise` allows you to perform the iterative process involved in prototyping a GUI:

- (1) adding new items to or deleting outdated items from the UIS file,
- (2) modifying the attributes of existing items interactively, and
- (3) manually editing the UIS file using a visual editor

For simple Graphical User Interfaces (GUI) using input files, output files, integers, floats, strings, logical, etc., the UIS file is easily created and modified using `guise` alone. For more complex GUI's (ie, those with mutu-

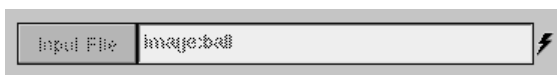


ally inclusive/exclusive/loose groups, etc), some manual editing may be required.

## B. Most Common UIS Line Fields

Each line in the UIS is made of *fields*, where each field in the UIS line corresponds to an *attribute* of the *GUI selection* or an *argument* for the *CLUI*. Each field of each line in the UIS must be specified properly in order for the UIS to be correctly interpreted. Many of these fields occur repeatedly in most of the different types of lines in the UIS. This section discusses in detail the meanings of the fields that occur in many of the UIS lines. Specific syntax of each UIS line is given in the next section.

### B.1. Activate



---

**Figure 1:** A de-activated GUI selection is stippled and will not accept input.

---

The *activate field* appears on almost every line of the UIS; it is applicable only to the GUI, having no effect on the CLUI. The allowed values for the activate field are TRUE (1) and FALSE (0). It indicates whether a selection is active (*activate* = TRUE) or inactive (*activate* = FALSE).

If an item is inactive, it will still be displayed on the GUI; however, it will usually appear stippled, and will not accept any input. This field allows various parts of the GUI to be disabled. For example, use of the *active* field allows application programs to be used by persons other than the developer while the software itself is still under development; the items corresponding to the unfinished software may be deactivated until segments of the controlling software are debugged, when the appropriate selections may be activated for use. Note that panes may be disabled by setting the activate field to 0 on the UIS line describing the controlling guide button, and that entire subforms may be disabled by setting the activate field to 0 on the UIS line describing the controlling subform button.

### B.2. Selected

The *selected field* is mostly for internal bookkeeping by the *xvforms* library; it indicates whether an item has recently had its value changed by the user. It is applicable only to the GUI, having no effect on the CLUI.

The allowed values for the selected field are TRUE (1) and FALSE (0). As far as creation of a UIS is concerned, this field should always be 0 for all lines *with the following exceptions*:

- If a guide pane is used on a subform, *only one* of the guide button (-g) lines in the guide pane [-G to -E] definition may have its *selected* field 1. In this way, the designer of the GUI may specify which pane will appear by default when the subform is initially mapped.
- In rare cases when a master form is used, it is desirable to have one of the subforms mapped at the start of the application along with the master form. If this is the case, one of the subform button (-d) lines in the master form [-S to -E] definition may have its *selected* field set to 1. If the subforms are NOT mutually exclusive,

more than one of the subform button (-d) lines may have its selected field set to 1.

### B.3. Optional



---

**Figure 2:** An optional input file selection has the optional box to the left. A checkmark is displayed when the option is selected.

---

The *optional field* indicates whether a selection is optional (*optional* = 1) or required (*optional* = 0). It is used by both the CLUI and the GUI.

In the CLUI, an argument described in the UIS by a selection with its optional field set to 1 is considered optional on the command line. The application will be informed if the user provided the argument corresponding to the selection, and will be expected to act accordingly. If the argument described in the UIS has its optional field set to 0, it will be interpreted as required on the command line; when the user fails to specify the argument on the command line, an error message will be displayed. If the user is running the program in interactive mode, they will be repeatedly prompted for a value for the required argument until they provide a valid response; if they are not running the program in interactive mode, the program will exit immediately after printing the error message.

In the GUI, a selection that is optional will have a button to the left of the title of the selection. If the box appears pressed, it indicates that this item is currently selected; if the button is not highlighted, the indicated parameter will not be utilized. The optional selection can be used in one of two ways:

1. When the selection has a default value, the user may either provide a new value or use the default value of the selection (by clicking the button).
2. When the selection has no default value, the user may choose between using the parameter (by clicking on the button and entering a value) or not using the parameter (by de-selecting the button).

Whether or not a default value is provided for the selection determines in which of the above two ways the optional field is used.

Mutually exclusive groups, mutually inclusive groups, and toggles must have all optional members. In other words, ALL selections within the [-C to -E], [-B to -E], and [-T to -E] groups must have the *optional* field = 1.

### B.4. Opt\_Sel (Option Selected)

The *opt\_sel* field indicates whether an optional selection is to be used by default (*opt\_sel* = 1) or not (*opt\_sel* = 0). It is used only by the GUI, as the CLUI *always* uses the default of an optional selection unless it is

otherwise specified on the command line.

If a selection is not optional, this field **MUST** be TRUE (option selected = 1). However, when a selection is optional, this field tells whether or not the optional box on the GUI (described above) is initially highlighted. If a command line argument is optional, then the corresponding selection on the GUI will also be optional.

In some cases, the box on the optional selection of the GUI may be intrusive. If you wish to hide the optional box, the `opt_sel` field may be set to 2 ("Selected, but not Shown").

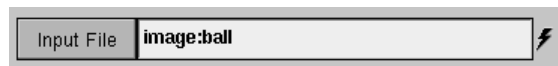
When a Toggle [-T to -E] set is used, **ONE and ONLY ONE** of the selections within the toggle should have the option selected field set to 1. Which selection has this field set determines the default selection of the toggle. Note that the selection in the toggle group that has its option selected field set to 1 must be consistent with the value of the *default* field on the (-T) line.

When a Mutually Exclusive [-C to -E] group is used, **ONE and ONLY ONE** of the selections within the set should have the optional selected field set to 1. Which one has this field set will determine the default selection of the mutually exclusive group.

When a Mutually Inclusive [-B to -E] group is used, **NONE or ALL** of the selections within the set should have the optional selected field set

When a Loose [-K to -E] group is used, **AT LEAST ONE** of the selections within the set should have the optional selected field set to 1.

## B.5. Live



---

**Figure 3:** A "live" selection has the lightning bolt symbol to the right.

---

The "live" field is used only by the GUI of a software object. When a selection is "live" (*live* = TRUE), software control is immediately diverted when the user initiates a change of value in the selection. A change of value can be initiated in the following ways:

1. When the user presses <Return> in the parameter box of a "live" InputFile (-I), OutputFile (-O), Integer(-i), Float (-f), Double (-h), String (-s), or StringList (-y) selection
2. When the user moves a scroll bar associated with a "live" Integer (-i), Float (-f), or Double (-h) selection
3. when the user changes the value of a "live" Toggle (-T), Logical (-l), Flag (-t), List (-x) or Cycle (-c) selection

When used in a \*.pane file, setting the "live" field in a UIS line means that when a change of value is initiated on the GUI in VisiQuest, the network involving the glyph associated with that kroutine will be re-run downstream from that glyph.

When used in a \*.form file, setting the "live" field means that when a change of value is initiated on the xvroutine's GUI, software control will be passed immediately back to the xvroutine. The xvroutine may then take appropriate action according to the value of the selection.

When a selection is not "live," flow control will not be returned to the application until the user clicks on an action button, or until the user changes the value of another selection that happens to be "live." If more than one piece of information is needed before action can be initiated, however, individual selections should NOT be "live."

## B.6. Geometry String

The *geometry string* appears as a field in practically every line in the UIS. Needless to say, it is ignored by the CLUI. The geometry string gives the overall geometry of an item in CHARACTERS of the font size currently being used in the following form, where the xoffset and yoffset are measured from the upper left hand corner of the parent widget. Geometry strings are float values, although they may be specified in the UIS lines as simple integers as well.

[width x height + xoffset + yoffset]

On selections, the geometry must allow enough room for an optional box (if the selection is optional), the title, and a parameter box for text input. If the selection is of type float (-f) or integer (-i), and a scroll bar is desired, at least 2 character widths should be provided for the scroll bar. Note that the size of the scroll bar determines its accuracy. If a scroll bar is not desired on a float or int selection, the width specified in the geometry string should be reduced until the scroll bar no longer appears.

## B.7. Title Offset

The *title offset* string appears as a field in many lines in the UIS that have a geometry string. Like the *geometry* field, it is always ignored by the CLUI. Like the geometry string, the title offset is a float value, although it may be specified in the UIS line as a pair of simple integers. It must be of the form:

[+x\_position +y\_position]

This field gives the x,y positioning of the LABEL of the item, which will appear x characters to the right and y characters down from the upper left hand corner of the backplane of the item.

## B.8. Default

This field gives the *default* value of the selection, whatever that selection may be. Used by both the GUI and the CLUI, the default will always be of the same type as the selection. For instance, the selection is specified by a Float (-f) line, the default field on that line MUST be a floating-point number. Integer selections must have integer defaults, and so on. The default field is very important to both the CLUI and the GUI. In the case of the CLUI, optional arguments that are not specified by the user on the command line will take on the default

value specified by the default field. In the GUI, the default value will appear as specified on the forms, and will remain so until they are changed by the user.

## B.9. Title

This field specifies the title that will appear on the GUI for the item; it is ignored by the CLUI. The title can contain spaces, but should remain fairly short (1-20 characters) to be effective. Since the title is delineated with single tic marks ('), if an apostrophe (') is to be used as part of the title, it must be "escaped" with a backslash, as in 'Don't Forget to Escape Apostrophes.' If no title is desired, specify '' as the title.

## B.10. Description

The *description field* gives a brief description of the purpose of the item. It is only used by the CLUI; comments in the generated code are taken directly from the description fields in the UIS. Descriptions should be clear and concise. Tic marks (') may NOT be used as part of the description.

## B.11. Variable

The *variable field* is used by both the CLUI and the GUI. In automatically generated code for the CLUI, the variable field dictates the name of the program argument. In code generated for the GUI of xv routines, the variable field is translated into the naming conventions for the C structure that provides the link between the application program and the *xvforms* library. Therefore, the string specified as the variable must be a syntactically correct variable name in the C programming language, or the code generated for the program will not compile.

# C. UIS Line Syntax

## C.1. UIS Lines That Structure The GUI and Required UIS Lines

This section details those UIS lines that are required in every UIS file, as well as those UIS lines that are used to define the structure of a GUI.

### C.1.1. StartForm (-F) Line

#### UIS Line Explanation:

The (-F) line is associated with the GUI as a whole. Every UIS file *must* begin with this line. The (-F) StartForm line must be matched with an End (-E) line. Collectively, the [-F to -E] form description will make up the entire UIS.

#### Use by the GUI:

When a master form is used, the StartForm (-F) line specifies attributes of the master form. When used with a single subform, many of the fields specified on the (-F) line will be overridden by

those on the StartSubform (-M) line; however, it must always appear.

**Use by the CLUI:**

The StartForm line does not play a part in the CLUI of a VisiQuest program.

**UIS File Placement:**

The (-F) line must be the first UIS line to appear in a UIS file; however, it may be preceded by comments having a '#' in the first column.

**syntax:**

-F Vers Act Sel GS TOffset 'Title' Var

**Vers:** The special *version field* is specific to the (-F) line only. The current version of the UIS is 4.3. The version number provides a way for the xvforms library to check if the UIS file will have a syntax proper to the current implementation. This allows us to keep track of outdated UIS files and to inform the user that they are outdated.

**Act:** The *activate field*, set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the entire GUI is to be de-activated, an extremely rare instance.

**Sel:** The *selected field* is always set to 1 on the (-F) line.

**GS:** For the *geometry string field*, [wxh+x+y] specifies the minimum size of the master form (if a [-S to -E] master form description is present). It is unused if a master form is not used. Geometry strings may be specified as float or integer values. Note that the master form will always be created large enough to accommodate its contents; the geometry string may be used to force it to be larger than that.

**TOffset:** In the *title offset field*, +x+y dictates positioning of the title on the master form if a master form is used, the positioning of the title on the backplane of the subform if a master form is not used.

**Title:** The *title string* must be delineated by tic marks. It will appear as the title on the master form if a master form is used; if a master form is not used, it will appear on the backplane of the subform if the subform geometry does not obscure it. For no title, use ' '.

**Var:**

1. **GUI:** The *variable name* is used to give the form a unique name if a master form is used, for internal book-keeping and reference in app-defaults files.
2. **GUI Code Generation:** The *variable name* is also used by the xvroutine code generator to name the GUI Info structure. Suppose the program name is '*program*' and the variable name on the (-F) line is '*tutorial*.' In this case,
  - *program\_tutorial* will be the name of the FormInfo structure defined in "form\_info.h," and
  - *gui\_info* will be a pointer to the *program\_tutorial* structure defined in the run\_*program* main GUI driver.

- -F 4.3 1 1 70x90+0+0 +5+1 'Our Form' tutorial

In the above example, the StartForm line shows that the UIS is of the current version, 4.3; the *activate field* is set as required, and the title "Our Form" is correctly surrounded by tic marks. The title will appear five character widths over and one character width down from the upper left hand corner of the backplane of the form.

### C.1.2. StartSubform (-M) Line

- The (-M) line is associated with a subform of the GUI. Collectively, The [-M to -E] subform definition specifies the appearance and functionality of a subform. The (-M) line must be matched with an End (-E) line, which ends the subform definition. At least one (-M) StartSubform line is required in a UIS file.
- Every GUI has at least one subform. For simple GUI's, a single subform is all that is necessary. The subform provides a backplane for one or more panes. If the subform has more than one pane, it will also have a guide pane with which to change the pane that is currently displayed.
- The StartSubform line is not used by the CLUI directly.
- A single StartSubform (-M) line may appear directly after the (-F) StartForm line (as in a CLUI or a GUI with a single subform), or a set of [-M to -E] subform definitions may appear after the [-S to -E] StartMaster definition (as in a GUI with multiple subforms).
- -M Act Sel GS TOffset 'Title' Var

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set to 0 if the subform described by the [-M to -E] set beginning with this (-M) line is to be deactivated.

**Sel:** The *selected field* is always set to 0 except in the case when, in the master form [-S to -E] description, the (-d) SubformButton line that is paired with this subform has its selected field set to 1, indicating that this subform should be mapped immediately, along with the master form.

**GS:** The *geometry field*, wxh+x+y. The geometry field describes, in characters, the minimum size of the subform and its position on the backplane. A subform will always be created large enough to accomodate its contents; the geometry field may be used to make it larger than that. Geometry strings may be set to float or integer values.

**TOffset:** The *title offset field*, +x+y, specifies in characters the distance from the upper-left corner of the backplane of the subform that the subform title should appear.

**Title:** The *title string* must be delineated by tic marks. It will appear on the backplane of the subform at the position specified by the *title offset*, unless it is obscured by one of the panes on the subform. For no title, use ' '.

**Var:**

1. **GUI:** The *variable name* is used to give the subform a unique name for internal bookkeeping and reference in app-defaults files. It is also used to match the (-M) line describing the subform with the (-d) line of the subform button that will bring it up.
2. **GUI Code Generation:** The *variable name* is used by the xvroutine code generator to name the SubformInfo structure. Suppose the variable on the (-F) line is 'master' and the variable name on the (-M) line is 'subform1.' In this case,
  - \* *master\_subform1* will be the name of the the SubformInfo structure corresponding to this subform, defined in form\_info.h.
  - \* *run\_subform1.c* will be the name of the file containing the subform driver
  - \* *run\_subform1* will be the name of the subform driver
  - \* *subform1\_info* will be a pointer to the *master\_subform1* structure defined in the *run\_subform1* driver

□ The title on the (-M) line is used by the code generator as the *short program description*. However, this short program description as given in the title of the (-M) line will be over-ridden by the short program description as specified in the man page if they differ.

□ -M 1 0 50x70+6+8 +5+1 'Subform Number One' subform1

In this example, we have a subform labeled "Subform Number One." This title will appear 5 characters to the right and 1 character down from the upper left hand corner of the backplane of the subform.

### C.1.3. StartPane (-P) Line

- The (-P) line is associated with a pane of the GUI. Collectively, the [-P to -E] pane definition specifies the appearance and functionality of a pane. The (-P) line must be matched with an End (-E) line, which ends the pane definition. At least one (-P) StartPane line is required in a UIS file.
- Every GUI has at least one pane. For very simple GUI's, a single pane on a single subform is all that is necessary. The pane provides a backplane for the desired selections.
- The StartPane line is not used by the CLUI directly.
- A single StartPane (-P) line may appear directly after the (-M) StartSubform line (as in a CLUI or a very simple GUI), or a set of [-P to -E] subform definitions may appear after the [-G to -E] GuidePane definition (as in a GUI with multiple panes).
- -P Act Sel GS TOffset 'Title' Var

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the pane described by the [-M to -E] set beginning with this (-M) line is to be deactivated.



**Sel:** The *selected field*; always set to 0.

**GS:** The *geometry field*,  $wxh+x+y$ . The geometry field describes, in characters, the size of the pane and its position on the subform. A pane will always be created large enough to accommodate its contents; the geometry string may be used to make it larger than that. Geometry strings may be set to float or integer values.

**TOffset:** The *title offset field*,  $+x+y$ , specifies in characters the distance from the upper-left corner of the backplane of the pane that the pane title should appear.

**Title:** The *title string* must be delineated by tic marks. It will appear on the backplane of the pane at the position specified by the *title offset*, unless it is obscured by one of the selections on the pane. If no title is desired, specify ' '.

**Var:**

1. **GUI:** The *variable name* is used to give the pane a unique name for internal book-keeping and reference in app-defaults files.  
It is also used to match the (-P) line describing the pane with the (-g) line of the guide button that will bring it up.
2. **GUI Code Generation:** The *variable name* is used by the xvroutine code generator to name the PaneInfo structure. Suppose the variable on the (-M) line is '*subform1*' and the variable name on the (-P) line is '*pane1*.' In this case,
  - \* *subform1\_pane1* will be the name of the the PaneInfo structure corresponding to this pane, defined in *form\_info.h*.
  - \* *run\_subform1.c* will be the name of the file containing the pane driver for this pane (and all other pane drivers for panes on this subform).
  - \* *run\_pane1* will be the name of the pane driver
  - \* *pane1\_info* will be a pointer to the *subform1\_pane1* structure defined in the *run\_pane1* pane driver.

□ -P 1 0 30x70+20+2 +10+2 'Pane Number One' pane1

In this example, we have a pane labeled "Pane Number One." This title will appear 5 characters to the right and 2 characters down from the upper left hand corner of the backplane of the pane. Note that the geometry specifies that this pane is to be created 20 characters to the right of the upper-left corner of the subform that it resides in, most likely to make room for a guide pane to the left. If a GuidePane [-G to -E] definition was used on this subform, it would have to include a GuideButton (-g) line with a variable named "pane1" in order to bring up this pane.

#### C.1.4. StartMaster (-S) Line

- The (-S) line is associated with the master form of a GUI. Collectively, the [-S to -E] pane definition specifies the appearance and functionality of a master form. The (-S) line must be matched with an End (-E) line, which ends the master form definition.

- When more than one subform is needed on the GUI, a *master form* is necessary. A master form always contains one or more *subform buttons* that allow the user to control which subform is currently displayed; while the currently displayed subforms can be mapped and unmapped, the master form remains displayed throughout the run of the program. A master form may also include master action buttons and selections, but these should be limited to only those that would otherwise be repeated on each of the subforms in the GUI.
- The StartMaster line does not play a part in the CLUI of a VisiQuest program.
- The StartMaster (-S) Line may *only* appear directly after the StartForm (-F) line. Any GUI that uses a master form will have *one and only one* occurrence of this line in its UIS file.
- 

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the entire master form is to be deactivated, an extremely rare instance.

**Mutually\_Exclusive:** This is a special field specific to the (-S) line only. It tells whether the subform buttons specified by the (-d) lines within the master form definition [-S to -E] are mutually exclusive. Note that this field has absolutely nothing to do with the (-C) line, which designates a mutually exclusive group of selections. If the *Mutually\_Exclusive* field is set to 0, the *xyforms* library will allow any or all of the subforms associated with the subform buttons to map when a subform button is clicked by the user. If the *Mutually\_Exclusive* field is set to 1, only one subform will be allowed to map at any time. If the user clicks on a subform button while a subform is already mapped, the previous subform will go away, and the new subform will replace it.

- -S 1 1

In the example, the presence of the StartMaster line indicates that we will be using a master form; subform buttons on the master will be mutually exclusive, so that the user will not be able to map more than one subform at any one time.

### C.1.5. StartGuide (-G) Line

- The (-G) line is associated with the guidepane of a subform. Collectively, the [-G to -E] pane definition specifies the appearance and functionality of a guidepane. The (-G) line must be matched with an End (-E) line, which ends the guidepane definition.
- When more than one pane is needed on a subform, a *guidepane* is necessary. A guidepane always contains two or more *guide buttons* that allow the user to control which pane is currently displayed on the subform. A guidepane may also include subform action buttons and selections, but these should be limited to only those which would otherwise be repeated on each of the panes of the subform.
- The StartGuide line does not play a part in the CLUI of a VisiQuest program.
- The StartGuide (-G) Line may *only* appear directly after the StartSubform (-M) line. Any subform that uses a guidepane will have *one and only one* occurrence of this line in its [-M to -E] subform definition.

□ -G Act GS TOffset 'Title'

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the guide pane described by the [-G to -E] set beginning with this (-G) line is to be deactivated; a very rare situation.

**GS:** The *geometry field*, wxh+x+y, describes, in characters, the size of the guide pane and its position on the subform. The geometry should be large enough to accommodate all the buttons on the guide pane. Geometry strings may be set to float or integer values.

**TOffset:** The *title offset field*, +x+y, specifies in characters the distance from the upper left hand corner of the backplane of the guide pane that the guide pane title should appear.

**Title:** The *title string* must be delineated by tic marks. It will appear on the backplane of the guide pane at the position specified by the *title offset*, unless it is obscured by one of the buttons on the guide pane. If no title is desired, specify ' '.

□ -G 1 20x70+0+0 +0+0 ' '

This example shows a guide pane that will appear along the left side of the subform on which it appears. The title has been deliberately left out.

### C.1.6. End (-E) Line

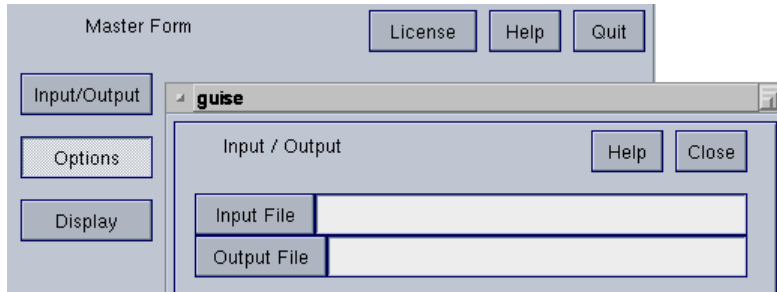
- The End (-E) line is used to end a UIS definition.
- The (-E) line is used by the GUI to find the end of a definition for a form, a subform, a pane, a guide pane, a master form, a toggle, a submenu, a mutually inclusive group, and a mutually inclusive group.
- It is not directly used by the CLUI.
- Placement of -E lines is dictated by context with other lines in the UIS.
- -E

The End (-E) line has no fields following it.

## C.2. UIS Lines That Are Positionally Restricted

Some lines in the UIS are restricted to a particular portion of the UIS. A summary of those lines follows.

## C.2.1. SubformButton (-d) Line



**Figure 4:** A subform button is used on a master form to map the subform with which it is associated.

- A SubformButton (-d) line is used to specify a subform button on a master form; it is matched with a [-M to -E] subform definition which will appear later in the UIS file. The variable specified on the (-d) line must match that of its corresponding (-M) line in order to associate the subform button with the definition of the subform that it will bring up.
- The user clicks on a subform button in order to map the subform with which it is associated. Mapping of the subform is done automatically. Whether or not two subforms may be mapped at the same time is dictated by the StartMaster line.
- The (-d) line is not used by the CLUI.
- The SubformButton (-d) line may appear only within a [-S to -E] master form definition. At least two subform buttons are required on a master form.
- -d Act Sel GS 'Title' Var

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the subform associated with the subform button described by this (-d) line is to be deactivated.

**Sel:** The *selected field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 1 if the subform associated with the subform button described by this (-d) line is to be automatically mapped along with the master form, not a very common usage.

**GS:** The *geometry string field*, wxh+x+y, in characters, describes the size of the subform button and its position on the master in characters. Note that the width field must be large enough to accommodate the title, or the title will be truncated. Geometry strings may be specified as float or integer values.

**Title:** The *title string* must be delineated by tic marks. It will appear as the title on the subform button. For no title, use ' '.

**Var:**

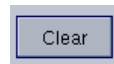
1. **GUI:** The *variable* is used to give the subform button a unique name for internal book-keeping and reference in app-defaults files.

The *variable* on a SubformButton Line is also used to match the subform button with the subform that it will be associated with. Thus, there must be a StartSubform (-M) line appearing later in the UIS file with an identical variable name. The subform defined by that [-M to -E] definition will be the one that is brought up when the user clicks on the subform button defined by this Subform-Button line.

□ -d 1 0 10x2+2+2 'Subform 1' subform1

In the example, we have a subform button 10 characters wide by 2 characters tall, located 2 characters to the right and 2 characters down from the upper-left corner of the master, labeled "Subform 1". The subform button produced by this line will control the mapping of the subform that is described by the [-M to -E] subform definition where the -M line has a variable also named "subform1".

### C.2.2. MasterAction (-n) Line



---

**Figure 5:** A master action lets the user request a particular action; in this case, the action will be to "Clear".

---

- The MasterAction (-n) Line specifies an action button on a master form.

#### Item Description

The purpose of an action button is to return software control to the application program; it is used exclusively in the \*.form file by *xvroutines*. In general, when a pane contains two or more non-live selections, an action button should be placed at the bottom of the pane so that the user can tell the application, "I've finished setting values of selections now, go ahead and perform (some action)." Alternatively, an application may be able to perform a particular operation without any additional information provided by the user besides the fact that the user now wants the operation performed; this is the other case in which an action button is appropriate. A mouse click on the action button causes software control to be diverted from the graphical user interface to the application's subroutine that is associated with the action button, where the name of the subroutine in question is determined by the variable associated with the action button, as well as the variable associated with the pane, guide pane, or master form on which the action button appears.

#### Use by the GUI

The user clicks on an action button to request a particular action from the application. Control is immediately returned from the GUI to the application so that the action may be performed.

## Use by the CLUI

The line in the UIS file representing the action button is ignored by the CLUI.

- The MasterAction (-n) line may appear only within a [-S to -E] master form definition. If desired, it may appear within a submenu [-D to -E] definition.
- -n Act Sel GS 'Title' 'Desc' Var

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the action button described by this (-n) line is to be deactivated.

**Sel:** The *selected field*, always set to 0.

**GS:** The *geometry field*, wxh+x+y, describes in characters the size of the action button and its position on the master. Note that the width field must be large enough to accommodate the title, or the title will be truncated. Geometry strings may be specified as float or integer values.

**Title:** The *title string* must be delineated by tic marks. It will appear as the title on the action button.

**Desc:** The *description string* must be delineated by tic marks; it should very briefly explain the purpose of the action button.

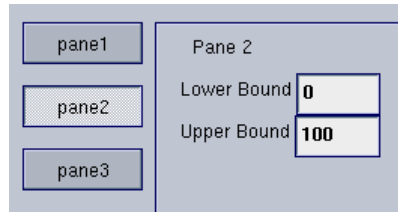
### Var:

1. **GUI:** The *variable* name is used to give the action button a unique name so that the button will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create two variables associated with the action button. These variables will appear in the PaneInfo structure as defined in "form\_info.h".  
Suppose that the action button was defined on a pane defined with the variable '*pane1*', and that the variable for the action button was '*do\_it*'. In this example,
  - \* *pane1\_info->do\_it\_struct* can be passed to *xvf\_set\_attribute()*
  - \* *pane1\_info->do\_it* will be TRUE if the user clicked on the action button.

- -n 1 0 7x2+2+5 'Reset' 'Resets display' reset

In the example, we have a master action button 7 characters wide by 2 characters tall, located 2 characters to the right and 5 characters down from the upper-left corner of the master, labeled "Reset."

### C.2.3. GuideButton (-g) Line



**Figure 6:** A guide button lets the user control which pane in a multi-pane subform is currently mapped. In this excerpt from a sample GUI, the three guide buttons in the guidepane to the left are used to switch among the three panes on the right. Currently, the guide button "Pane2" is used to display the second pane, named "Pane 2".

- A GuideButton (-g) line is used to specify a guide button on a guidepane; it is matched with a [-P to -E] pane definition which will appear later in the UIS file. The variable specified on the (-g) line must match that of its corresponding (-P) line in order to associate the guide button with the definition of the pane that it will bring up.

#### Item Description

When a subform contains more than one pane, it is necessary to have a mechanism with which to change the pane that is currently displayed. The guide button, situated on the guide pane, provides this mechanism. Each pane has its own guide button; when the user clicks on the guide button, that pane is displayed. Only one pane may be displayed in a subform at any given time. Each guide button is associated with its pane with the use of the *variable*, which must be identical for the guide button and the pane with which it is associated. Guide buttons are used exclusively in the \*.form file for *xvroutines*.

#### Use by the GUI

The user clicks on a guide button in order to map the pane with which it is associated. Mapping of the pane is done automatically. Only one pane may be mapped at any one time.

#### Use by the CLUI

The line in the UIS file representing the guide button is ignored by the CLUI.

- The GuideButton (-g) line may appear only within a [-G to -E] guidepane definition. At least two guide buttons are required on a guidepane. If desired, it may appear within a submenu [-D to -E] definition.
- -g Act Sel GS 'Title' Var

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the pane described by the [-P to -E] set associated with this (-g) line is to be deactivated; this is a common way to de-activate a pane on a subform.

**Sel:** The *selected field* is set to 1 (TRUE) or 0 (FALSE). Set it to 1 if the pane associated with the guide button described by this (-g) line is the one to be displayed immediately when this subform is mapped. Set it to 0 otherwise. *One and only one* guide button on a subform may have its selected field set to 1.

**GS:** The *geometry field*, wxh+x+y, describes, in characters, the size of the guide button and its position on the guide pane. The geometry should be large enough to accommodate the title or it will be truncated. Geometry strings may be set to float or integer values.

**Title:** The *title string* must be delineated by tic marks. It will appear on the backplane of the guide pane at the position specified by the *title offset*, unless it is obscured by one of the buttons on the guide pane.

**Var:**

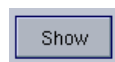
**GUI:** The *variable* is used to give the guide button a unique name for internal book-keeping and reference in app-defaults files.

The variable is also used to match the guide button to the pane with which it will be associated. Thus, there must be a StartPane (-P) line appearing later in the UIS file with an identical variable name. The pane defined by that [-P to -E] definition will be the one that is brought up when the user clicks on the guide button defined by this GuideButton line.

□ -g 1 0 5x2+6+1 'Input' input

In this example, we are creating a guide button labeled "Input," implying that the pane associated with will deal with input. It is 5 characters wide and 2 characters high, located 6 characters to the right and 1 character down from the upper-left corner of its guide pane. The guide button produced by this (-g) line will control the mapping of the pane described by the [-P to -E] pane definition, where the -P line also has a variable name of "Input." If we had wanted that pane to appear on this subform as the default, we would have set the selected field on this (-g) line to 1.

### C.2.4. SubformAction (-m) Line



---

**Figure 7:** A subform action lets the user request a particular action; in this case, the action will be to "show".

---

□ A SubformAction (-m) line is used to specify an action button on a guidepane.

#### Item Description

The purpose of an action button is to return software control to the application program; it is used exclusively in the \*.form file by *xvroutines*. In general, when a pane contains two or more non-live selections, an action button should be placed at the bottom of the pane so that the user can tell the



application, "I've finished setting values of selections now, go ahead and perform (some action)." Alternatively, an application may be able to perform a particular operation without any additional information provided by the user besides the fact that the user now wants the operation performed; this is the other case in which an action button is appropriate. A mouse click on the action button causes software control to be diverted from the graphical user interface to the application's subroutine that is associated with the action button, where the name of the subroutine in question is determined by the variable associated with the action button, as well as the variable associated with the pane, guide pane, or master form on which the action button appears.

### Use by the GUI

The user clicks on an action button to request a particular action from the application. Control is immediately returned from the GUI to the application so that the action may be performed.

### Use by the CLUI

The line in the UIS file representing the action button is ignored by the CLUI.

- The SubformAction (-m) line may appear only within a [-G to -E] master form definition. If desired, it may appear within a submenu [-D to -E] definition.
- m Act Sel GS 'Title' 'Desc' Var
- The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the action button described by this (-m) line is to be deactivated.
- The *selected field*, always set to 0.
- The *geometry field*, wxh+x+y, describes, in characters, the size of the action button and its position on the master form. Note that the width field must be large enough to accommodate the title or else the title will be truncated. Geometry strings may be set to float or integer values.
- The *title string* must be delineated by tic marks. It will appear as the title on the action button.
- The *description string* must be delineated by tic marks. Give a brief description of the purpose of the action button.

### Var:

1. **GUI:** The *variable* name is used to give the action button a unique name so that the button will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create two variables associated with the action button. These variables will appear in the PaneInfo structure as defined in "form\_info.h".  
Suppose that the action button was defined on a pane defined with the variable '*pane1*', and that the variable for the action button was '*do\_it*'. In this example,  
\* *pane1\_info->do\_it\_struct* can be passed to *xvf\_set\_attribute()*  
\* *pane1\_info->do\_it* will be TRUE if the user clicked on the action button.

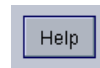
- -m 1 0 7x2+2+5 'Plot' 'Plot the data' plot

In the example, we have a subform action button 7 characters wide by 2 characters tall, located 2 characters to the right and 5 characters down from the upper left hand corner of the guide pane, and labeled "Plot".

### C.3. UIS Lines Representing General Purpose Items

The general purpose items listed below may appear within a [-S to -E] master form definition, [-G to -E] guidepane definition, or [-P to -E] Pane definition, depending on whether they are desired in a master form, guidepane or pane.

#### C.3.1. Help (-H) Line



---

**Figure 8:** Every master form, subform, and pane should have one help button that allows the user to access an online help page that covers that part of the GUI.

---

- A Help (-H) line specifies a help button.

#### Item Description

The help button is a standard device on the GUI; all programs in the VisiQuest system are required to provide online help that can be accessed directly via the user interface. The help button provides access to a help file involving no additional work by the application program; all that is needed is the correct path to the help file. VisiQuest standards specify that help buttons always provide help pages specific to their location. Thus, on a GUI with all three levels of hierarchy, the help button located on the master form will give information about the program as a whole, the help button located on a subform will give information about the general options offered by that subform, and the help button located on a pane will give detailed information about the I/O located on that pane. One help button should be used for each pane, each subform, and the master form (if any) in the \*.form file for *xvroutines*. Every \*.pane file should include one help button to access the man page for the program.

#### Use by the GUI

The help button provides access to online help from the GUI.

#### Use by the CLUI

The UIS line representing the help button is ignored by the CLUI.

- The -H Help line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may appear within a submenu [-D to -E] definition if desired.

□ -H Act GS 'Title' 'Desc' Help\_Path Var

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the help button described by this (-H) line is to be deactivated.

**GS:** The *geometry field*, wxh+x+y, in characters, describes the size of the help button and its position on the master form in characters. Note that the width field must be large enough to accommodate the title, or the title will be truncated. Geometry strings may be specified as float or integer values.

**Title:** The *title string* must be delineated by tic marks. It will appear as the title on the help button.

**Desc:** The *description string* must be delineated by tic marks; it should mention what the help file(s) talk about. **Help\_Path:** This path may be the path to a specific help file to be displayed when the user clicks on the help button, or it may be the path to a directory in which two or more help files are contained. If the help path accesses a particular file, the help object will come up with that file displayed in it. If it accesses a directory, the help object that comes up will have an "Options" button which produces a pulldown menu at the upper left hand corner. There will be one entry in the pulldown menu for each file in the directory. When the user selects an item from the menu, the file that it names will be displayed. NOTE: if a directory is specified as the help path, it is important to make sure that no extraneous files exist in that directory, as the help object will create an item in the pulldown menu with the name of each file in the directory, and allow the user to access every file in the directory. NOTE: if the help path specified is invalid, the user will receive an error message when they click on the help button.

□ -H 1 7x2+2+7 'Help' \$ENVISION/objects/xvroutine/xprism/help/plot help

The *Help\_Path* used here specifies a directory (note that the use of the \$ENVISION or \$TOOLBOX environment variables are accepted, as is the use of a tilde, "~"). This particular directory happens to contain 5 help files that discuss methods of output for the **xprism2** and **xprism3** plotting packages; all of them are properly formatted for online display. When the help button is clicked, this executes khel in its own window. The file named "Overview" found in this directory will initially be displayed in the text window; the others may be displayed by selecting from the list shown.

**Var:**

1. **GUI:** The *variable* name is used to give the help button a unique name so that the button will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create a variables associated with the help button. This variable will appear in the PaneInfo structure as defined in "form\_info.h".  
Suppose that the help button was defined on a pane defined with the variable '*pane1*', and that the variable for the help button was '*help*'. In this example,  
\* *pane1\_info->help\_struct* can be passed to *xvf\_set\_attribute()*

### C.3.2. Quit (-Q) Line



---

**Figure 9:** A quit button allows the user to quit the program. Quit buttons are also used to close panes and subforms.

---

- A Quit (-Q) line is used to allow the user to quit the program or close a subform.

#### Item Description

The quit button allows the program to exit. In addition, when a GUI has a master form with several subforms, quit buttons on the subforms allow the user to unmap them and thus reduce clutter on the screen. When used to exit the program, the quit button is by convention labeled "Exit;" when used to close a subform, the quit button is by convention labeled "Close."

#### Use by the GUI

Allows the user to close a subform or exit the program.

#### Use by the CLUI

The UIS line specifying the Quit button is ignored by the CLUI.

- The -Q Help line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may appear within a submenu [-D to -E] definition if desired.
- -Q Act Sel GS 'Title'

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the quit button described by this (-Q) line is to be deactivated.

**Sel:** The *selected field*, always set to 0 (FALSE).

**GS:** The *geometry field*,  $wxh+x+y$ , describes, in characters, the size of the quit button and its position on the master form. Note that the width field must be large enough to accommodate the title, or the title will be truncated. Geometry strings may be set to float or integer values.

**Title:** The *title string* must be delineated by tic marks. It will appear as the title on the help button. If the (-Q) line describing the quit button appears in the description of a subform that is subordinate to a master form, this title can be specified as 'Close,' which causes the subform to unmap (or closes the window in question), rather than causing the program to end.

- -Q 1 0 5x2+2+9 'Quit'

In the example, we have a quit button, 5 characters wide by 2 characters tall, located 2 characters to the right and 9 characters down from the upper-left corner of the master form, guide pane, or pane on which it appears, and labeled "Quit."

### C.3.3. InputFile (-I) Line



---

**Figure 10:** An Input File selection; the title is on a button that brings up the VisiQuest file/aliases browser.

---

- The InputFile (-I) line allows the user to specify an input file from either the GUI or the CLUI.

#### Item Description

This allows the user to specify an input file; the file will be checked for existence and read permission. An input file may be part of a toggle, or a member of a mutually inclusive or mutually exclusive group. It may be optional or required. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### Use by the GUI

When used by the GUI, the *input file* selection allows the user to enter an input file directly or to use the file browser to select a file. The input file selection consists of an optional box (if the selection is optional), a title, and a text box in which the user may enter a filename. The title of the input file selection is actually a button that can be used to bring up the file browser.

#### Use by the CLUI

When used by the CLUI, the input file specifies an *input file argument*, as in "-i1 my\_input\_image.viff". If an invalid input file argument is entered, the user will be re-prompted. When an optional input file argument is absent from the command line, it takes on the default value specified (the default value may be NULL).

- The -I InputFile line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may be part of a [-T to -E] toggle, a [-C to -E] mutually exclusive group, or a [-B to -E] mutually inclusive group.
- -I Act Sel Opt OptSel Live FileCheck GS 'Def' 'Title' 'Desc' Var

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the input selection described by the (-I) line is to be deactivated.

**Sel:** The *selected field*, always set to 0.

**Opt:** The *optional field*, 1 if selection is optional, 0 if it is required.

**OptSel:** The *option selected field*; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 1 if the selection is optional and the default action is to use the default value of the selection. Use 0 if the selection is optional and the default action is not to utilize its value.

**Live:** The *live field*, 1 if the selection is to return immediately when the user presses <Return>, 0 otherwise.

**FileCheck:** The *filecheck field* determines whether or not the input file specified should be checked for existence before being allowed to go to the program. When filecheck is set to TRUE (1), the filename will be checked for existence and readability. If the file does not exist or is not readable, an error message will be printed. When set to FALSE (0), the filename is not checked.

**GS:** The *geometry field*, wxh+x+y. The geometry describes, in characters, the size of the input file selection and its position. The geometry should be large enough to accommodate all components of the input file selection: the optional box, if the selection is optional, the title, and a reasonable amount of space for the parameter box in which the user will type the filename of the inputfile. Geometry strings may be set to float or integer values.

**Def:** The *default field* may contain a value for the default input file, or it may contain a space, indicating that no default is applicable. The default field must be surrounded by tic marks.

**Title:** The *title string* must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the input file. If no title is desired, specify ' '.

**Desc:** The *description field* should contain a short description of the input file. The code generator will use this string as a comment when generating code for the command line user interface. The description must be surrounded by tic marks.

**Var:**

1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the input file selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection appears on a master form, guidepane, or pane, respectively. The first will contain the value of the filename when control is returned to the application, the second has a specialized purpose, and the third is generated only if the selection is optional or live. Suppose that the selection was defined on a pane defined with the variable '*panel*', and that the variable for the input file was '*il*'. In this example,
  - \* *panel\_info->il* will hold the current filename
  - \* *panel\_info->il\_struct* can be passed to *xvf\_set\_attribute()*
  - \* *panel\_info->il\_selected* will be generated only if the selection is live; it will be set to TRUE when the user changes the filename.
  - \* *panel\_info->il\_optsel* will be generated only if the selection is optional; it will be TRUE if the

user has the optional box highlighted.

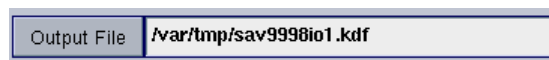
Variable references are constructed similarly for input file selections on the master form and guidepane.

3. **CLUI:** The *variable* name is used to determine the name of the flag that will mark this input file argument to the program. Supposing that the variable for the input file is 'i1', the command line syntax will be [-i1 {input file}].
4. **CLUI Code Generation:** The code generated will include two fields corresponding to the input file argument, indicating whether or not the input file was specified on the command line, and the filename specified. In our example,
  - \* *program->i1\_flag* will be TRUE if [-i1] appeared on the command line
  - \* *program->i1\_file* will hold the filename

□ -I 1 0 0 1 1 1 40x1+1+4 ' ' 'Input Image' 'the input image' i1

In this example, we have a required input file selection that is "live," so that when used on the GUI, control will be returned immediately to the application when the user presses <Return> in the parameter box. The *file type* is 1, indicating that the input file will be an image file formatted for VisiQuest. The selection has been given plenty of room - 40 characters - so that the parameter box will be nice and long; the title of this input file selection is "Input Image." The name of the field in the Form Information structure generated by the *xvroutine* code generator will be identifiable by the letters, "i1," and the flag provided for the CLUI will be "-i1 file-name."

### C.3.4. OutputFile (-O) Line



**Figure 11:** The title of an Output File selection is a button that brings up the VisiQuest file/aliases browser.

- The OutputFile (-O) line allows the user to specify an output file from either the GUI or the CLUI.

#### Item Description

The OutputFile allows the user to specify an output file that will be checked for write permission. OutputFile lines may be part of a toggle, or members of a mutually inclusive or mutually exclusive group. They may be optional or required. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### Use by the GUI

When used by the GUI, the *output file selection* allows the user to enter an output file directly or to use the file browser to select an existing output file. The output file selection consists of an optional box (if the selection is optional), a title, and a text box in which the user may enter a filename. The

title of the output file selection is actually a button that can be used to bring up the file browser.

### Use by the CLUI

When used by the CLUI, the output file specifies an *output file argument*, as in "-o1 my\_output\_file". The user will be re-prompted for invalid output file arguments. When an optional output file argument is absent from the command line, it takes on the default value specified (the default value may be NULL).

The -O OutputFile line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may be part of a [-T to -E] toggle, a [-C to -E] mutually exclusive group, or a [-B to -E] mutually inclusive group.

□ -O Act Sel Opt OptSel Live FileCheck GS 'Def' 'Title' 'Desc' Var

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the output selection described by the (-O) line is to be deactivated.

**Sel:** The *optional field*, 1 if selection is optional, 0 if it is required.

**Opt:** The *option selected field* is set to 1 if selection is required, 1 if it is optional and the default action is to use the default value provided, or 0 if it is optional and the default action is to ignore its value.

**OptSel:** The *option selected field*; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 1 if the selection is optional and the default action is to use the default value of the selection. Use 0 if the selection is optional and the default action is not to utilize its value.

**Live:** The *live field* is set to 1 if the selection is to return immediately when the user presses <Return>, 0 otherwise.

**FileCheck:** The *filecheck field* determines whether or not the output file specified should be checked for existence before being allowed to go to the program. When filecheck is set to TRUE (1), the filename will be checked for existence and readability; if the file does not exist or is not readable, an error message will be printed. When set to FALSE (0), the filename is not checked.

**GS:** The *geometry field*, wxh+x+y. The geometry describes, in characters, the size of the output file selection and its position. The geometry should be large enough to accommodate all components of the output file selection: the optional box, if the selection is optional, the title, and a reasonable amount of space for the parameter box in which the user will type the filename of the output file. Geometry strings may be set to float or integer values.

**Def:** The *default field* may contain a value for the default output file, or it may contain a space, indicating that no default is applicable. The default field must be surrounded by tic marks.

**Title:** The *title string* must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea of the purpose of the output file. If no title is desired, specify ' '.

**Desc:** The *description field* should contain a short description of the output file. The code generator will use this string as a comment when generating code for the command line user interface. The description must be



surrounded by tic marks.

**Var:**

1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the output file selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection appears on a master form, guidepane, or pane, respectively. The first will contain the value of the filename when control is returned to the application, the second has a specialized purpose, and the third is generated only if the selection is optional or live.  
Suppose that the selection was defined on a pane defined with the variable '*pane1*', and that the variable for the output file was '*o2*'. In this example,
  - \* *pane1\_info->o2* will hold the current filename
  - \* *pane1\_info->o2\_struct* can be passed to *xvf\_set\_attribute()*
  - \* *pane1\_info->o2\_selected* will be generated only if the selection is live; it will be set to TRUE when the user changes the filename.
  - \* *pane1\_info->o2\_optsel* will be generated only if the selection is optional; it will be TRUE if the user has the optional box highlighted.Variable references are constructed similarly for output file selections on the master form and guidepane.
3. **CLUI:** The *variable* name is used to determine the name of the flag that will mark this output file argument to the program. Supposing that the variable for the output file is '*o2*', the command line syntax will be [-o2 {output file}].
4. **CLUI Code Generation:** The code generated will include two fields corresponding to the output file argument, indicating whether or not the output file was specified on the command line, and the filename specified. In our example,
  - \* *program->o2\_flag* will be TRUE if [-o2] appeared on the command line
  - \* *program->o2\_file* will hold the filename

□ -O 1 0 1 0 0 1 40x1+1+4 ' ' 'Blurred Output Image' 'output blurred image' o2

In this example, we have an optional output file selection; the default action will be to ignore this selection, as the option selected field is set to false, and no default value has been provided. The *file type* is 1, indicating that if this selection is used, the output file will be an image file formatted for the VisiQuest System. The selection has been given plenty of room--40 characters--so that the parameter box will be nice and long; the title of this output file selection is "Blurred Output Image". The name of the field in the Form Information structure generated by the xvroutine code generator will be identifiable by the letters "o2", and the flag provided for the CLUI will be "-o2 filename."

### C.3.5. Integer (-i) Line



**Figure 12:** An required Integer selection using a scrollbar.

- The Integer (-i) line allows the user to specify an integer from either the GUI or the CLUI.

#### Item Description

Integers may be part of a toggle, or members of a mutually inclusive or mutually exclusive group. They may be optional or required. Bounded or unbounded values may be specified. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### Use by the GUI

When used by the GUI, the *integer selection* consists of an optional box (if the integer is optional) a title, and a text box in which the integer may be entered. Bounded integers may use an optional scrollbar. Live integers will have the stylized carriage return symbol appended to the right side.

#### Use by the CLUI

When used by the CLUI, the integer specifies an *integer argument*, as in "-int\_variable 21". Values entered for bounded integer arguments will be checked to be sure they are within bounds; the user will be re-prompted for invalid entries. When an optional integer argument is absent from the command line, it takes on the default value specified.

- The -i Integer line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may be part of a [-T to -E] toggle, a [-C to -E] mutually exclusive group, or a [-B to -E] mutually inclusive group.
- -i Act Sel Opt OptSel Live GS LB UB Def Mech Incr 'Title' 'Desc' Var

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the integer selection described by the (-i) line is to be deactivated.

**Sel:** The *selected field*, always set to 0.

**Opt:** The *optional field*, 1 if selection is optional, 0 if it is required.

**OptSel:** The *option selected field*; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 1 if the selection is optional and the default action is to use the default value of the selection. Use 0 if the selection is optional and the default action is not to utilize its value.

**Live:** The *live field* is set to 1 if the selection is to return immediately when the user presses <Return>, 0 otherwise.

**GS:** The *geometry field*,  $w \times h + x + y$ . The geometry describes, in characters, the size of the integer selection and its position. The geometry should be large enough to accommodate all components of the integer file selection: the optional box, if the selection is optional, the title, and a reasonable amount of space for the parameter box in which the user will type in a value for the integer. Extra space must be allocated for a scroll bar, if one is desired. Geometry strings may be set to float or integer values.

**LB and UB:** The *lower bound* and *upper bound* fields of the integer selection are very important. Here are the possible values for the lower and upper bounds of an integer:

LB = UB = -2	{ legal values < 0 }
LB = UB = -1	{ legal values <= 0 }
LB = UB = 1	{ legal values >= 0 }
LB = UB = 2	{ legal values > 0 }
LB = UB = X	{ no range checking }
LB = X; UB = Y	{ X <= legal value <= Y }

**Def:** The *default* for the integer must be a legal value, set within the bounds specified by the *lower bound* and the *upper bound*.

**Mech:** The *mechanism field* is used to specify whether or not a scrollbar will be used with integers that are strictly bound (ie, only those that fit the last case for the *lower bound* and *upper bound* given above). When integers are strictly bound and the width is sufficient, a scroll bar will appear to the right of the integer parameter box, allowing the user to set the integer value with the scroll bar. To turn off the scrollbar, provide a value of 0; to enable the scrollbar, provide a value of 1. In the future, specifying a 2 may indicate the preference for an alternate mechanism to be used for this purpose, such as a dial.

**Incr:** This field is slated for use in the future; it does not work as of yet. The *increment* tells how scrollbars and dials should increment when there is a scrollbar or a dial used with the int selection. The *increment* field may be set to any integer that is greater than 0 and less than (*lower bound* - *upper bound*); it can have an equal or lesser number of decimal places than what is specified by *precision*. It is used ONLY when the int selection uses a scrollbar or a dial, i.e., when *mech* is set to 1 or 2, the int is strictly bound, and the width is sufficient to allow creation of the scrollbar or dial.

**Title:** The *title string* must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the integer's use. If no title is desired, specify ' '.

**Desc:** The *description field* should contain a short description of the purpose of the integer. The code generator will use this string as a comment when generating code for CLUI. The description must be surrounded by tic marks.

**Var:**

1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.  
**GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the integer selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection

appears on a master form, guidepane, or pane, respectively. The first will contain the value of the integer when control is returned to the application, the second has a specialized purpose, and the third is generated only if the selection is optional or live.

Suppose that the selection was defined on a pane defined with the variable *'panel'*, and that the variable for the integer was *'thres'*. In this example,

\* *panel\_info->thres* will hold the current integer value

\* *panel\_info->thres\_struct* can be passed to *xvf\_set\_attribute()*

\* *panel\_info->thres\_selected* will be generated only if the selection is live; it will be set to TRUE when the user changes the integer value.

\* *panel\_info->thres\_optsel* will be generated only if the selection is optional; it will be TRUE if the user has the optional box highlighted.

Variable references are constructed similarly for integer selections on the master form and guidepane.

2. **CLUI:** The *variable* name is used to determine the name of the flag that will mark this integer argument to the program. Supposing that the variable for the integer is *'thres'*, the command line syntax will be [-thres {integer value}].

3. **CLUI Code Generation:** The code generated will include two fields corresponding to the integer argument, indicating whether or not the integer was specified on the command line, and the integer specified. In our example,

\* *program->thres\_flag* will be TRUE if [-thres] appeared on the command line

\* *program->thres\_int* will hold the integer

□ -i 1 0 0 1 0 50x1+1+4 0 255 40 1 1 'Threshold' 'threshold value' thres

In this example, we have an required integer selection. The selection has been given plenty of room--50 characters--so that the bounded integer will be given a scroll bar next to the parameter box; because there is lots of room, the scroll bar will be long, and therefore precise. The title of this integer selection is "Threshold". Legal values for the Threshold integer will be from 0 to 255, inclusive, and the default value will be 40. The name of the field in the Form Information structure generated by the xvroutine code generator will be identifiable by the letters, "thres," and the flag provided for the command line user interface might be "-thres 10."

### C.3.6. Float (-f) Line



**Figure 13:** An optional Float selection using a scrollbar.

The Float (-f) line allows the user to specify a float from either the GUI or the CLUI.

#### Item Description

A float is used when a floating point number is needed. A float may be part of a toggle, or a

member of a mutually inclusive or mutually exclusive group. It may be optional or required. Bounded or unbounded values may be specified. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

### Use by the GUI

When used by the GUI, the float specifies a *float selection*. The float selection consists of an optional box (if the float is optional), a title, and a text box in which the float may be entered. A bounded float may use an optional scroll bar. A live float will have the stylized carriage return symbol appended to the right side.

### Use by the CLUI

When used by the CLUI, the float specifies a *float argument*, as in "-float\_variable 0.123". Values entered for bounded float arguments will be checked to be sure they are within bounds; the user will be re-prompted for invalid and bogus entries. When an optional float argument is absent from the command line, it takes on the default value specified.

- The -f Float line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may be part of a [-T to -E] toggle, a [-C to -E] mutually exclusive group, or a [-B to -E] mutually inclusive group.
  
- -f Act Sel Opt OptSel Live GS LB UB Def Prec Mech Incr 'Title' 'Desc' Var

**Act:** The *activation* field is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the float selection described by the (-f) line is to be deactivated.

**Sel:** The *selected* field, always set to 0.

**Opt:** The *optional* field is set to 1 if selection is optional, 0 if it is required.

**OptSel:** The *option selected field*; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 1 if the selection is optional and the default action is to use the default value of the selection. Use 0 if the selection is optional and the default action is not to utilize its value.

**Live:** The *live field* is set to 1 if the selection is to return immediately when the user presses <Return>, 0 otherwise.

**GS:** The *geometry field*, wxh+x+y. The geometry describes, in characters, the size of the float selection and its position. The geometry should be large enough to accommodate all components of the float selection: the optional box, if the selection is optional, the title, and a reasonable amount of space for the parameter box in which the user will type in the value of the float. Extra space must be allocated for a scroll bar, if one is desired. Geometry strings may be set to float or integer values.

**LB and UB:** The *lower bound* and *upper bound* fields of the float selection are very important. Here are the possible values for the lower and upper bounds of a float:

LB = UB = -2	{ legal values < 0.0 }
LB = UB = -1	{ legal values <= 0.0 }
LB = UB = 1	{ legal values >= 0.0 }
LB = UB = 2	{ legal values > 0.0 }

LB = UB = X	{ no range checking }
LB = X; UB = Y	{ X <= legal value <= Y }

**Def:** The *default* for the float must be a legal value, set within the bounds specified by the *lower bound* and the *upper bound*.

**Prec:** This integer field specifies the *precision*, or the number of decimal places that are to be displayed on the GUI and the CLUI for the float value. For example, setting *prec* to 3 says that three digits should appear to the right of the decimal point. Use of a -1 indicates that all the available decimal places should be used. Use of a 0 indicates that trailing zeroes should be removed from the value; a decimal point will appear only if it is followed by a digit.

**Mech:** The *mechanism field* is used to specify whether or not a scrollbar will be used with floats that are strictly bound (i.e., only those that fit the last case for the *lower bound* and *upper bound* given above). When floats are strictly bound and the width is sufficient, a scroll bar will appear to the right of the float parameter box, allowing the user to set the float value with the scroll bar. To turn off the scrollbar, provide a value of 0; to enable the scrollbar, provide a value of 1. In the future, specifying a 2 may indicate the preference for an alternate mechanism to be used for this purpose, such as a dial.

**Incr:** This field is slated for use in the future; it does not work as of yet. The *increment* tells how scrollbars and dials should increment when there is a scrollbar or a dial used with the float selection. The *increment* field may be set to any floating-point number that is greater than 0 and less than (*lower bound* - *upper bound*); it may have an equal or lesser number of decimal places than what is specified by *precision*. It is used ONLY when the float selection uses a scrollbar or a dial, i.e., when *mech* is set to 1 or 2, the float is strictly bound, and the width is sufficient to allow creation of the scrollbar or dial.

**Title:** The *title string* must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the float's use. If no title is desired, specify ' '.

**Desc:** The *description field* should contain a short description of the purpose of the integer. The code generator will use this string as a comment when generating code for the command line user interface. The description must be surrounded by tic marks.

**Var:**

1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the float selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection appears on a master form, guidepane, or pane, respectively. The first will contain the value of the float when control is returned to the application, the second has a specialized purpose, and the third is generated only if the selection is optional or live. Suppose that the selection was defined on a pane defined with the variable '*pane1*', and that the variable for the float was '*dspmt*'. In this example,
  - \* *pane1\_info->dspmt* will hold the current float value
  - \* *pane1\_info->dspmt\_struct* can be passed to *xvf\_set\_attribute()*
  - \* *pane1\_info->dspmt\_selected* will be generated only if the selection is live; it will be set to

TRUE when the user changes the float value.

\* *panel\_info->dspmt\_optsel* will be generated only if the selection is optional; it will be TRUE if the user has the optional box highlighted.

Variable references are constructed similarly for float selections on the master form and guide-pane.

3. **CLUI:** The *variable* name is used to determine the name of the flag that will mark this float argument to the program. Supposing that the variable for the float is '*dspmt*', the command line syntax will be [-dspmt {float value}].
4. **CLUI Code Generation:** The code generated will include two fields corresponding to the float argument, indicating whether or not the float was specified on the command line, and the filename specified. In our example,
  - \* *program->dspmt\_flag* will be TRUE if [-dspmt] appeared on the command line
  - \* *program->dspmt\_float* will hold the float value

□ -f 1 0 1 1 0 25x1+1+4 -2.0 -2.0 -3.0 3 1 1 'Neg Displacement' 'neg displace value' dspmt

In this example, we have an optional float selection; since the option selected field is set to 1, the default action will be to use the default value of -3.0. The selection has been given just enough room - 25 characters. Float values will be displayed to 3 decimal places, as specified by the *precision* field. It cannot have a scroll bar, since the value is unbounded, so both the *mechanism* and the *increment* fields are ignored. Legal values are any floats strictly less than 0.0. The title of this float selection is "Neg Displacement". The name of the field in the Form Information structure generated by the *xvroutine* code generator will be identifiable by the letters "dspmt", and the flag provided for the command line user interface might be "-dspmt -10.0."

### C.3.7. Double (-h) Line



---

**Figure 14:** An optional Double selection.

---

The Double (-h) line allows the user to specify a float from either the GUI or the CLUI.

#### Item Description

A double is used when double precision accuracy is needed. It may be part of a toggle, or a member of a mutually inclusive or mutually exclusive group. It may be optional or required. Bounded or unbounded values may be specified. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### Use by the GUI

When used by the GUI, the double specifies a *double selection*. The double selection consists of

an optional box (if the double is optional), a title, and a text box in which the double may be entered. A bounded double may use an optional scroll bar. A live double will have the stylized carriage return symbol appended to the right side.

### Use by the CLUI

When used by the CLUI, the double specifies a *double argument*, as in "-dbl\_variable 0.987654321". Values entered for a bounded double argument will be checked to be sure they are within bounds; the user will be re-prompted for out-of-bounds and bogus entries. When an optional double selection is absent from the command line, it takes on the default value specified.

- The -h Double line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may be part of a [-T to -E] toggle, a [-C to -E] mutually exclusive group, or a [-B to -E] mutually inclusive group.
- -h Act Sel Opt OptSel Live GS LB UB Def Prec Mech Incr 'Title' 'Desc' Var

**Act:** The *activation field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the double selection described by the (-h) line is to be deactivated.

**Sel:** The *selected field*, always set to 0.

**Opt:** The *optional field* is set to 1 if selection is optional, 0 if it is required.

**OptSel:** The *option selected field*; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 1 if the selection is optional and the default action is to use the default value of the selection. Use 0 if the selection is optional and the default action is not to utilize its value.

**Live:** The *live field*, 1 if the selection is to return immediately when the user presses <Return>, 0 otherwise.

**GS:** The *geometry field*, wxh+x+y. The geometry describes, in characters, the size of the double selection and its position. The geometry should be large enough to accommodate all components of the double selection: the optional box, if the selection is optional, the title, and a reasonable amount of space for the parameter box in which the user will type in the value of the double. Extra space must be allocated for a scroll bar, if one is desired. Geometry strings may be set to float or integer values.

**LB and UB:** The *lower bound* and *upper bound* fields of the double selection are very important. Here are the possible values for the lower and upper bounds of a double:

LB = UB = -2	{ legal values < 0.0 }
LB = UB = -1	{ legal values <= 0.0 }
LB = UB = 1	{ legal values >= 0.0 }
LB = UB = 2	{ legal values > 0.0 }
LB = UB = X	{ no range checking }
LB = X; UB = Y	{ X <= legal value <= Y }

**Def:** The *default* for the double must be a legal double, within the bounds specified by the *lower bound* and the *upper bound*.



**Prec:** This is the number of decimal places that should be used when displaying and obtaining the double in the GUI. It must be a positive integer no greater than 14.

**Mech:** The *mechanism field* is used to specify whether or not a scrollbar will be used with doubles that are strictly bound (i.e., only those that fit the last case for the *lower bound* and *upper bound* given above). When doubles are strictly bound and the width is sufficient, a scroll bar will appear to the right of the doubles parameter box, allowing the user to set the doubles value with the scroll bar. To turn off the scrollbar, provide a value of 0; to enable the scrollbar, provide a value of 1. In the future, specifying a 2 may indicate the preference for an alternate mechanism to be used for this purpose, such as a dial.

**Incr:** This field is slated for use in the future; it does not work as of yet. The *increment* tells how scrollbars and dials should increment when there is a scrollbar or a dial used with the double selection. The *increment* field may be set to any double precision number that is greater than 0 and less than (*lower bound* - *upper bound*). It may have an equal or lesser number of decimal places than what is specified by *precision*. It is used ONLY when the double selection uses a scrollbar or a dial, i.e., when *mech* is set to 1 or 2, the double is strictly bound, and the width is sufficient to allow creation of the scrollbar or dial.

**Title:** The *title string* must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the double's use. If no title is desired, specify ' '.

**Desc:** The *description field* should contain a short description of the purpose of the double. The code generator will use this string as a comment when generating code for the command line user interface. The description must be surrounded by tic marks.

**Var:**

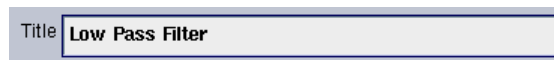
1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the double selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection appears on a master form, guidepane, or pane, respectively. The first will contain the value of the double when control is returned to the application, the second has a specialized purpose, and the third is generated only if the selection is optional or live. Suppose that the selection was defined on a pane defined with the variable '*pane1*', and that the variable for the double was '*dspmt*'. In this example,
  - \* *pane1\_info->dspmt* will hold the current double value
  - \* *pane1\_info->dspmt\_struct* can be passed to *xvf\_set\_attribute()*
  - \* *pane1\_info->dspmt\_selected* will be generated only if the selection is live; it will be set to TRUE when the user changes the double value.
  - \* *pane1\_info->dspmt\_optsel* will be generated only if the selection is optional; it will be TRUE if the user has the optional box highlighted.Variable references are constructed similarly for double selections on the master form and guidepane.
3. **CLUI:** The *variable* name is used to determine the name of the flag that will mark this double argument to the program. Supposing that the variable for the double is '*dspmt*', the command line syntax will be [-dspmt {double value}].

4. **CLUI Code Generation:** The code generated will include two fields corresponding to the double argument, indicating whether or not the double was specified on the command line, and the file-name specified. In our example,
  - \* *program->dspmt\_flag* will be TRUE if [-dspmt] appeared on the command line
  - \* *program->dspmt\_double* will hold the double value

□ -h 1 0 1 1 0 25x1+1+4 -2.0 -2.0 -3.0 5 0 0 'Neg Displacement' 'neg displace value' dspmt

In this example, we have an optional double selection; since the *option selected* field is set to 1, the default action will be to use the default value of -3.0. The selection has been given just enough room--25 characters--and we know that it cannot have a scroll bar since the value is unbounded. Legal values are any doubles strictly less than 0.0. The title of this double selection is "Neg Displacement". The name of the field in the Form Information structure generated by the *xvroutine* code generator will be identifiable by the letters "dspmt," and the flag provided for the CLUI might be "-dspmt -10.0."

### C.3.8. String (-s) Line




---

**Figure 15:** A string selection that will be used to enter a title.

---

- The String (-s) line allows the user to specify a string from either the GUI or the CLUI.

#### Item Description

Strings may be part of a toggle, or members of a mutually inclusive or mutually exclusive group. They may be optional or required. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### Use by the GUI

When used by the GUI, the *string selection* consists of an optional box (if the string is optional), a title, and a text box in which the string may be entered. Live strings will have the stylized carriage return symbol appended to the right side.

#### Use by the CLUI

When used by the CLUI, the String specifies a *string argument*, as in "-string\_variable 'my special string'". When an optional string selection is absent from the command line, it takes on the default value specified (the default value may be NULL). Any printable string is considered to be valid input.

- The -s String line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may be part of a [-T to -E] toggle, a [-C to -E]

mutually exclusive group, or a [-B to -E] mutually inclusive group.

□ -s Act Sel Opt OptSel Live GS 'Def' 'Title' 'Desc' Var

**Act:** The *activation field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the string selection described by the (-s) line is to be deactivated.

**Sel:** The *selected field*, always set to 0.

**Opt:** The *optional field* is set to 1 if selection is optional, 0 if it is required.

**OptSel:** The *option selected field*; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 1 if the selection is optional and the default action is to use the default value of the selection. Use 0 if the selection is optional and the default action is not to utilize its value.

**Live:** The *live field* is set to 1 if the selection is to return immediately when the user presses <Return>, 0 otherwise.

**GS:** The *geometry field*, wxh+x+y. The geometry describes, in characters, the size of the string selection and its position. The geometry should be large enough to accommodate all components of the string selection: the optional box, if the selection is optional, the title, and a reasonable amount of space for the parameter box in which the user will type in the value of the string. Geometry strings may be set to float or integer values.

**Def:** The *default field* may contain a value for the string, or it may contain a space, indicating that no default is applicable. The default field must be surrounded by tic marks.

**Title:** The *title string* must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the string. If no title is desired, specify ' '.

**Desc:** The *description field* should contain a short description of the purpose of the string selection. The code generator will use this description as a comment when generating code for the command line user interface. The description must be surrounded by tic marks.

**Var:**

1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the string selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection appears on a master form, guidepane, or pane, respectively. The first will contain the value of the string when control is returned to the application, the second has a specialized purpose, and the third is generated only if the selection is optional or live.  
Suppose that the selection was defined on a pane defined with the variable '*pane1*', and that the variable for the string was '*func*'. In this example,  
\* *pane1\_info->func* will hold the current string  
\* *pane1\_info->func\_struct* can be passed to *xvf\_set\_attribute()*

\* *panel\_info->func\_selected* will be generated only if the selection is live; it will be set to TRUE when the user changes the string.

\* *panel\_info->func\_optsel* will be generated only if the selection is optional; it will be TRUE if the user has the optional box highlighted.

Variable references are constructed similarly for string selections on the master form and guide-pane.

3. **CLUI:** The *variable* name is used to determine the name of the flag that will mark this string argument to the program. Supposing that the variable for the string is '*func*', the command line syntax will be [-func {string}].
4. **CLUI Code Generation:** The code generated will include two fields corresponding to the string argument, indicating whether or not the string was specified on the command line, and the string specified. In our example,
  - \* *program->func\_flag* will be TRUE if [-func] appeared on the command line
  - \* *program->func\_string* will hold the string

□ -s 1 0 0 1 0 40x2+1+5 'sin(x)\*cos(y)' 'Function' '3D function' func

In this example, we have a required string selection. The selection has been given an excessive amount of room--40 characters long and 2 characters high--so that the parameter box will have lots of room for the function, and if that isn't enough, the cursor will wrap around to the second line. The title of this string selection is "Function," and the default value of this function is "sin(x)\*cos(y)." The name of the field in the Form Information structure generated by the *xvroutine* code generator will be identifiable by the letters "function," and the flag provided for the command line user interface will be "-function xxxxxx."

### C.3.9. Flag (-t) Line



---

**Figure 16:** A flag selection allows the user to indicate whether or not to use a particular option, in this case, "Invert".

---

- The Flag (-t) line allows the user to specify a flag from either the GUI or the CLUI.

#### Item Description

A flag allows the input of an implied boolean value. A flag may be part of a toggle, or a member of a mutually inclusive or mutually exclusive group. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### Use by the GUI

When used by the GUI, the flag specifies a *flag selection*. The flag selection consists of an optional

box and a title. When the optional box is highlighted, the value of the flag is considered to be TRUE (1). When the optional box is un-highlighted, the value of the flag is considered to be FALSE (0). A live flag selection has the stylized carriage return symbol appended to the right side.

### Use by the CLUI

When used by the CLUI, the flag specifies a *flag argument*. If the flag is provided on the command line, as in "-flag\_variable," the argument is considered to be TRUE (1). If the flag is absent from the command line, the argument is considered to be FALSE (0).

- The -t Flag line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may be part of a [-T to -E] toggle, a [-C to -E] mutually exclusive group, or a [-B to -E] mutually inclusive group.
- -t Act Sel Opt OptSel Live GS 'Title' 'Desc' Var

**Act:** The *activation field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the flag selection described by the (-t) line is to be deactivated.

**Sel:** The *selected field*, always set to 0.

**Opt:** The *optional field* must be set to 1 for the Flag.

**OptSel:** The *option selected field* determines the default value of the flag; it must be off (0) by default.

**Live:** The *live field* is set to 1 if the selection is to return immediately when the user changes the value of the logical, 0 otherwise.

**GS:** The *geometry field*, wxh+x+y. The geometry describes, in characters, the size of the logical selection and its position. The geometry should be large enough to accommodate all components of the logical selection: the optional box, if the logical is optional, the title, and the button that will display the current boolean value of the logical. Geometry strings may be set to float or integer values.

**Title:** The *title string* must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the logical selection. If no title is desired, specify ' '.

**Desc:** The *description field* should contain a short description of the string. The code generator will use this description as a comment when generating code for the CLUI. The description must be surrounded by tic marks.

### Var:

1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the flag selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection

appears on a master form, guidepane, or pane, respectively. The first will contain the value of the filename when control is returned to the application, the second has a specialized purpose, and the third is generated only if the flag is live.

Suppose that the selection was defined on a pane defined with the variable *'panel'*, and that the variable for the flag was *'clobber'*. In this example,

\* *panel\_info->clobber* will hold the current boolean value

\* *panel\_info->clobber\_struct* can be passed to *xvf\_set\_attribute()*

\* *panel\_info->clobber\_selected* will be generated only if the selection is live; it will be set to TRUE when the user changes the flag value.

\* *panel\_info->clobber\_optsel* will be generated only if the selection is optional; it will be TRUE if the user has the optional box highlighted.

Variable references are constructed similarly for flag selections on the master form and guidepane.

3. **CLUI:** The *variable* name is used to determine the name of the flag that will mark this flag argument to the program. Supposing that the variable on for the flag is *'clobber'*, the command line syntax will be [-clobber].
4. **CLUI Code Generation:** The code generated will include a field corresponding to the clobber argument, indicating whether or not the flag was specified on the command line.
  - \* *program->clobber\_flag* will be TRUE if [-clobber] appeared on the command line

□ -t 1 0 1 0 0 20x1+1+5 'clobber' 'clobber old file?' clobber

In this example, we have an optional flag selection in its most common configuration: optional and NOT selected. On the GUI, the optional box will not be initially highlighted. Only if the user deliberately turns on the flag will it be used. On the command line, the flag will only be set to TRUE if the user specifies "-clobber."

### C.3.10. Logical (-l) Line



---

**Figure 17:** A logical selection that lets the user specify whether or not a software object should be installed in Cantata.

---

- The Logical (-l) line allows the user to specify a logical value explicitly from either the GUI or the CLUI.

#### Item Description

A logical allows the input of an explicit boolean value. Logicals may be part of a toggle, or members of a mutually inclusive or mutually exclusive group. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

### Use by the GUI

When used by the GUI, the *logical selection* consists of an optional box (if the selection is optional), a title, and a button which may be switched back and forth between the two possible values. Live logical selections have the stylized carriage return symbol appended to the right side.

### Use by the CLUI

When used by the CLUI, the Logical specifies a *logical argument*. The logical value is provided on the command line explicitly, as in "-logical 0" for a value of FALSE, or "-logical 1" for a value of TRUE. When an optional logical argument is absent from the command line, it takes on the default value specified. The user will be re-prompted for invalid entries.

- The -l Logical line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may be part of a [-T to -E] toggle, a [-C to -E] mutually exclusive group, or a [-B to -E] mutually inclusive group.
- -l Act Sel Opt OptSel Live GS Def 'Title' 'L0' 'L1' 'Desc' Var

**Act:** The *activation field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the logical selection described by the (-l) line is to be deactivated.

**Sel:** The *selected field*, always set to 0.

**Opt:** The *optional field* is set to 1 if the selection is optional, 0 if it is required.

**OptSel:** The *option selectedP field*; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 1 if the selection is optional and the default action is to use the default value of the selection. Use 0 if the selection is optional and the default action is not to utilize its value.

**Live:** The *live field* is set to 1 if the selection is to return immediately when the user changes the value of the logical, 0 otherwise.

**GS:** The *geometry field*, wxh+x+y. The geometry describes, in characters, the size of the logical selection and its position. The geometry should be large enough to accommodate all components of the logical selection: the optional box, if the logical is optional, the title, and the button that will display the current boolean value of the logical. Geometry strings may be set to float or integer values.

**Def:** The *default field* specifies the default value of the logical. It must be set to 1 (TRUE) or 0 (FALSE).

**Title:** The *title string* must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the logical selection. If no title is desired, specify ' '.

**L0:** The *label0 string* must be delineated by tic marks. This is the label that will appear on the button of the logical selection when the value of the logical is equal to 0. The most common *label0* string is "False."

**L1:** The *label1 string* must be delineated by tic marks. This is the label that will appear on the button of the logical selection when the value of the logical is equal to 1. The most common *label1* string is "True."

**Desc:** The *description field* should contain a short description of the string. The code generator will use this description as a comment when generating code for the CLUI. The description must be surrounded by tic marks.

**Var:**

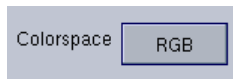
1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the logical selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection appears on a master form, guidepane, or pane, respectively. The first will contain the value of the logical when control is returned to the application, the second has a specialized purpose, and the third is generated only if the selection is optional or live.  
Suppose that the selection was defined on a pane defined with the variable '*pane1*', and that the variable for the logical was '*grid*'. In this example,
  - \* *pane1\_info->grid* will hold the current value of the logical (0 or 1)
  - \* *pane1\_info->grid\_struct* can be passed to *xvf\_set\_attribute()*
  - \* *pane1\_info->grid\_selected* will be generated only if the selection is live; it will be set to TRUE when the user changes the logical value.
  - \* *pane1\_info->grid\_optsel* will be generated only if the selection is optional; it will be TRUE if the user has the optional box highlighted.Variable references are constructed similarly for logical selections on the master form and guidepane.
3. **CLUI:**The *variable* name is used to determine the name of the flag that will mark this logical argument to the program. Supposing that the variable for the logical is '*grid*', the command line syntax will be [-grid {boolean value}].
4. **CLUI Code Generation:** The code generated will include two fields corresponding to the logical argument, indicating whether or not the logical was specified on the command line, and the boolean value specified. In our example,
  - \* *program->grid\_flag* will be TRUE if [-grid] appeared on the command line
  - \* *program->grid\_logic* will hold the boolean value

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

In this example, we have an optional logical selection; the selection will appear with the 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?"

### C.3.11. Cycle (-c) Line






---

**Figure 18:** A Cycle selection allowing the user to set the colorspace model, with three settings, "RGB", "CMY" and "HLS"; the current value is "RGB".

---

- The *Cycle (-c) line* allows the user to specify one of a set of predefined cyclic values from either the GUI or the CLUI.

### Item Description

A cycle allows the input of one of a number of predefined choices. Each choice is represented by both a number and a label, where the numbers are incremental integers beginning with an integer that you can specify. For example, you might have a cycle that moved through the sequence, "2 (dot), 3 (dash), 4 (dot-dash)". A Cycle may NOT be part of a toggle, but it may be a member of a mutually inclusive or mutually exclusive group. It may be optional or required. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

### Use by the GUI

When used by the GUI, the Cycle specifies a *cycle selection*. The cycle selection consists of an optional box (if the selection is optional), a title, and a button which may be cycled through the set of possible values. Live cycle selections have the stylized carriage return symbol appended to the right side. Because of their presentation, cycles are only recommended in situations where there are a small number (3 - 6) choices.

### Use by the CLUI

When used by the CLUI, the Cycle specifies a *cycle argument*. The cycle value is provided on the command line explicitly, where either the integer or the label of a choice may be specified. In the above example, if the user wanted "dot", they might specify either "-cycle\_variable 2" or "-cycle\_variable dot". When an optional cycle argument is absent from the command line, it takes on the default value specified. The user will be re-prompted for invalid entries. Both the integer value and the label of the cycle value specified will be made available to the program.

- The -c Cycle line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may be part of a [-C to -E] mutually exclusive group or a [-B to -E] mutually inclusive group.
- -c Act Sel Opt OptSel Live GS Total Start Def 'Title' 'Desc' Var 'Choice\_start' ..... 'Choice\_end'

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the cycle selection described by the (-c) line is to be deactivated.

**Sel:** The *selected field*, always set to 0.

**Opt:** The *optional field* is set to 1 if selection is optional, 0 if it is required.

**OptSel:** The *option selected field*; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 1 if the selection is optional and the default action is to use the default value of the selection. Use 0 if the selection is optional and the default action is not to utilize its value.

**Live:** The *live field* is set to 1 if the selection is to return immediately when the user changes the value of the logical, 0 otherwise.

**GS:** The *geometry field*,  $wxh+x+y$ . The geometry describes, in characters, the size of the logical selection and its position. The geometry should be large enough to accommodate all components of the logical selection: the optional box, if the logical is optional, the title, and the button that will display the current boolean value of the logical. Geometry strings may be set to float or integer values.

**Total:** The *total number* of choices that will be offered.

**Start:** The *start* of the values to be returned. Values are always incremental integers, but the start field allows you to specify these values to start at any integer value. For example, if start is set to 1 and *total* is set to 5, this implies that you will have 5 total choices for the cycle; values will range from 1 to 5, inclusive. However, you may want to specify start as something other than one--for instance, -5, 0, or 21. In these cases, assuming total is still 5, ranges will go from -5 to -1, 0 to 4, or 21 to 25, respectively.

**Def:** The *default field* specifies the default value of the cycle. It must be set to an integer within the range of start to total + start - 1.

**Title:** The *title string*, it must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the cycle selection. If no title is desired, specify ' '.

**Desc:** The *description field* should contain a short description of the string. The code generator will use this description as a comment when generating code for the command line user interface. The description must be surrounded by tic marks.

**Var:**

1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the cycle selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection appears on a master form, guidepane, or pane, respectively. The first will contain the integer value of the cycle when control is returned to the application, the second will contain the string associated with the integer value, the third has a specialized purpose, and the fourth is generated only if the selection is optional or live.  
Suppose that the selection was defined on a pane defined with the variable '*pane1*', and that the variable for the cycle was '*fruit\_type*'. In this example,  
\* *pane1\_info->fruit\_type* will hold the current integer value of the cycle

- \* *panel\_info->fruit\_type\_label* will hold the string associated with the current integer value
- \* *panel\_info->fruit\_type\_struct* can be passed to *xvf\_set\_attribute()*
- \* *panel\_info->fruit\_type\_selected* will be generated only if the selection is live; it will be set to TRUE when the user changes the cycle value.
- \* *panel\_info->fruit\_type\_optsel* will be generated only if the selection is optional; it will be TRUE if the user has the optional box highlighted.

Variable references are constructed similarly for cycle selections on the master form and guide-pane.

3. **CLUI:** The *variable* name is used to determine the name of the flag that will mark this cycle argument to the program. Supposing that the variable for the cycle is '*fruit\_type*', the command line syntax will be [-fruit\_type {integer value}].
4. **CLUI Code Generation:** The code generated will include three fields corresponding to the cycle argument, indicating whether or not the cycle was specified on the command line, the integer value specified, and the string associated with that integer. In our example,

- \* *program->fruit\_type\_flag* will be TRUE if [-fruit\_type] appeared on the command line

- \* *program->fruit\_type\_cycle* will hold the integer value

- \* *program->fruit\_type\_label* will hold the string associated with the integer value

- Last in the (-c) line is a list of strings describing the meanings of the cycle selection values. Each choice string must be delineated by tic marks; make sure that there are total strings specified, one for each integer value that the cycle selection may take on. According to the choice made by the user, one of these strings will be returned to the application, along with the integer value representing that choice.

- -c 1 0 1 0 1 15x1+1+3 4 1 2 'Fruit' 'type of fruit' fruit\_type 'apple' 'orange' 'pear' 'lemon'

In this example, we have an optional cycle selection. The cycle may take on one of four values represented by the strings "apple," "orange," "pear," and "lemon." *Total* is set to 4, since there are four choices. A *start* of 1 implies that this cycle will represent incremental integer values starting at 1: 1, 2, 3, and 4, respectively. The *default value* will be 2, interpreted in this case as a choice of "orange" from the four predefined types of available "Fruit".

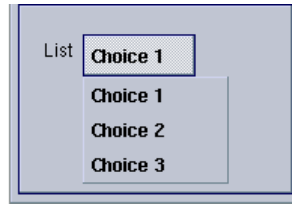
- -c 1 0 1 0 1 15x1+1+3 5 -5 -1 'Vegetable' 'type of veg' veg\_type 'corn' 'broccoli' 'peas' 'carrots' 'lima beans'

In this example, we have another optional cycle selection. The cycle may take one of five values represented by the strings "corn," "broccoli," "peas," "carrots," and "lima beans." *Total* is set to 5, since there are five choices. A *start* of -5 implies that this cycle will represent incremental integer values starting at -5: -5, -4, -3, -2, and -1, respectively. The *default value* will be -1, interpreted somewhat disappointingly as a choice of "lima beans."

- -c 1 0 1 0 1 15x1+1+3 3 25 26 'Mammal' 'type of mammal' mammal\_type 'hedgehog' 'wombat' 'poodle'

In this example, we have a required cycle selection. The cycle may take one of three values represented by the strings "hedgehog", "wombat", and "poodle". *Total* is set to 3, since there are three choices. A *start* of 25 implies that this cycle will represent incremental integer values starting at 25: 25, 26, and 27, respectively. The *default value* will be 26, interpreted as a choice of "wombat".

### C.3.12. List (-x) Line



---

**Figure 19:** A list selection is used to select from three choices. A list selection differs from a displayed list selection in that it has a button with which to access the pulldown menu, rather than displaying the list all the time.

---

- The List (-x) line allows the user to specify a value from a predefined list from either the GUI or the CLUI.

#### Item Description

A list allows the input of one of a number of predefined choices. Each choice is represented by both a number and a label, where the numbers are incremental integers beginning with an integer that may be specified. For example, you might have a list that offered the choices, "red (1), orange (2), yellow (3), green (4), blue(5) indigo (6)". Lists may be NOT be part of a toggle, but they may be members of a mutually inclusive or mutually exclusive group. They may be optional or required. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### Use by the GUI

When used by the GUI, the *list selection* consists of an optional box (if the selection is optional), a title, and a button which displays a pulldown menu containing the possible values of the list. Live list selections have the stylized carriage return symbol appended to the right side. Because of their presentation, lists are recommended in situations where there are a large number (more than 5) choices. The *list selection* differs from the *displayed list selection* in that the *list selection* features a pulldown menu that is accessible via a button, while the *displayed list selection* features a list the contents of which are always displayed.

#### Use by the CLUI

When used by the CLUI, the list specifies a *list argument*. The list value is provided on the command line explicitly, where either the integer or the label of a choice may be specified. In the above example, if the user wanted "red", they might specify either "-list\_variable 1" or "-list\_variable red". When an optional list argument is absent from the command line, it takes on the default value specified. The user will be re-prompted for invalid entries. Both the integer value and the label of the list value

specified will be made available to the program.

- The `-x` List line may be specified within a `[-S to -E]` master form definition, a `[-G to -E]` guide pane definition, or a `[-P to -E]` pane definition. In any of these, it may be part of a `[-C to -E]` mutually exclusive group or a `[-B to -E]` mutually inclusive group.
- `-x Act Sel Opt OptSel Live GS Total Start Def 'Title' 'Desc' Var 'Choice_start' ..... 'Choice_end'`

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the list selection described by the `(-c)` line is to be deactivated.

**Sel:** The *selected field*, always set to 0.

**Opt:** The *optional field* is set to 1 if the selection is optional, 0 if it is required.

**OptSel:** The *option selected field*; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 1 if the selection is optional and the default action is to use the default value of the selection. Use 0 if the selection is optional and the default action is not to utilize its value.

**Live:** The *live field* is set to 1 if the selection is to return immediately when the user changes the value of the logical, 0 otherwise.

**GS:** The *geometry field*, `wxh+x+y`. The geometry describes, in characters, the size of the logical selection and its position. The geometry should be large enough to accommodate all components of the logical selection: the optional box if the logical is optional, the title, and the button that will display the current boolean value of the logical. Geometry strings may be set to float or integer values.

**Total:** The *total* number of choices that will be offered.

**Start:** The *start* of the values to be returned. Values are always incremental integers, but the *start field* allows you to specify these values to start at any integer value. For example, if *start* is set to 1 and *total* is set to 5, this implies that you will have 5 total choices for the list; values will range from 1 to 5, inclusive. However, you may want to specify *start* as something other than one--for instance, -5, 0, or 21. In these cases, assuming *total* is still 5, ranges will go from -5 to -1, 0 to 4, or 21 to 25, respectively.

**Def:** The *default field* specifies the default value of the list. It must be set to an integer within the range of **start to total + start - 1**.

**Title:** The *title string* must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the list selection. If no title is desired, specify `' '`.

**Desc:** The *description field* should contain a short description of the string. The code generator will use this description as a comment when generating code for the CLUI. The description must be surrounded by tic marks.

## Var:

1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the list selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection appears on a master form, guidepane, or pane, respectively. The first will contain the integer value of the list when control is returned to the application, the second will contain the string associated with the integer value, the third has a specialized purpose, and the fourth is generated only if the selection is optional or live.  
Suppose that the selection was defined on a pane defined with the variable '*panel*', and that the variable for the list was '*day*'. In this example,
  - \* *panel\_info->day* will hold the current integer value of the list
  - \* *panel\_info->day\_label* will hold the string associated with the current integer value
  - \* *panel\_info->day\_struct* can be passed to *xvf\_set\_attribute()*
  - \* *panel\_info->day\_selected* will be generated only if the selection is live; it will be set to TRUE when the user selects a new item from the list.
  - \* *panel\_info->day\_optsel* will be generated only if the selection is optional; it will be TRUE if the user has the optional box highlighted.Variable references are constructed similarly for list selections on the master form and guidepane.
3. **CLUI:** The *variable* name is used to determine the name of the flag that will mark this list argument to the program. Supposing that the variable for the list is '*day*', the command line syntax will be [-day {integer value}].
4. **CLUI Code Generation:** The code generated will include three fields corresponding to the list argument, indicating whether or not the list was specified on the command line, the integer value specified, and the string associated with that integer. In our example,
  - \* *program->day\_flag* will be TRUE if [-day] appeared on the command line
  - \* *program->day\_list* will hold the integer value
  - \* *program->day\_label* will hold the string associated with the integer value

" **Choice\_start - Choice\_end** " Last in the (-x) line is a list of strings describing the meanings of the list selection values. Each choice string must be delineated by tic marks; make sure that there are *total* strings specified, one for each integer value that the list selection may take. According to the choice made by the user, one of these strings will be returned to the application, along with the integer value representing that choice.

```
□ -x 1 0 1 0 1 4x1+1+3 7 0 6 'Days' 'day of week' day 'Mon' 'Tues' 'Wed' 'Thurs' 'Fri' 'Sat' 'Sun'
```

In this example, we have an optional list selection. The list may take on one of seven values represented by abbreviations of days of the week. *Total* is set to 7, since there are seven days in the week. A *start* of 0 implies that this list will represent incremental integer values starting at 0: 0, 1, 2, 3, 4, 5, and 6, respectively. The *default value* will be 6, interpreted in this case as a choice of "Sun" from the list of days.

□ -x 1 0 0 1 1 9x1+1+3 3 1 1 'Face Card' 'face card' face\_card 'king' 'queen' 'jack'

In this example, we have a required list selection. The list may take on one of three values, represented by the strings "king", "queen", and "jack". *Total* is set to 3, since there are three face cards. A *start* of 1 implies that this list will represent incremental integer values starting at 1: 1, 2, and 3, respectively. The *default* value will be 1, interpreted that the default face card is the "king."

□ -x 1 0 1 0 1 15x1+1+3 3 20 20 'Weather' 'type of weather' weather\_type 'sunny' 'cloudy' 'raining'

In this example, we have an optional list selection. The list may take on one of three values, represented by the strings "sunny", "cloudy", and "raining." *Total* is set to 3, since there are three choices. A *start* of 21 implies that this list will represent incremental integer values starting at 20: 20, 21, and 22, respectively. The *default value* will be 20, interpreted as a choice of "sunny."

### C.3.13. DisplayList (-z) Line



---

**Figure 20:** A displayed list selection has its list displayed all the time, in contrast to a list selection, which has a button with which to access the pulldown menu.

---

- The DisplayList (-z) line allows the user to specify a value from a predefined list from either the GUI or the CLUI.

#### Item Description

A list allows the input of one of a number of predefined choices. Each choice is represented by both a number and a label, where the numbers are incremental integers beginning with an integer that you can specify. For example, you might have a list that offered the choices, "red (1), orange (2), yellow (3), green (4), blue(5) indigo (6)". Lists may be NOT be part of a toggle, but they may be members of a mutually inclusive or mutually exclusive group. They may be optional or required. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

#### Use by the GUI

When used by the GUI, the *displayed list selection* consists of an optional box (if the selection is optional), a title, and a list of items from which the user may select. The list may have a scrollbar depending on the geometry of the selection. Live displayed list selections have the stylized carriage return symbol appended to the right of the label. Because of their presentation, displayed lists are recommended in situations where there is a large number (more than 5) choices. Depending on context, the actual size of the displayed list selection might be limited and the scrollbar used rather than allowing the displayed list to grow large enough to display all entries. The *displayed list selection* features a list the contents of which are always displayed, while the *list selection* features a pulldown menu which is accessible via a button.

#### Use by the CLUI

When used by the CLUI, the displayed list specifies a *list argument*. The list value is provided on the command line explicitly, where either the integer or the label of a choice may be specified. In the above example, if the user wanted "red", they might specify either "-list\_variable 1" or "-list\_variable red". When an optional list argument is absent from the command line, it takes on the default value specified. The user will be re-prompted for invalid entries. Both the integer value and the label of the list value specified will be made available to the program.



- The `-z DisplayList` line may be specified within a `[-S to -E]` master form definition, a `[-G to -E]` guide pane definition, or a `[-P to -E]` pane definition. In any of these, it may be part of a `[-C to -E]` mutually exclusive group, a `[-B to -E]` mutually inclusive group, or a `[-K to -E]` loose group.
- `-z Act Sel Opt OptSel Live GS Total Start Def 'Title' 'Desc' Var 'Choice_start' .... 'Choice_end'`

**Act:** The *activation* field is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the list selection described by the `(-c)` line is to be deactivated.

**Sel:** The *selected* field, always set to 0.

**Opt:** The *optional* field is set to 1 if selection is optional, 0 if it is required.

**OptSel:** The *option selected* field; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 1 if the selection is optional and the default action is to use the default value of the selection. Use 0 if the selection is optional and the default action is not to utilize its value.

**Live:** The *live field* is set to 1 if the selection is to return immediately when the user changes the value of the logical, 0 otherwise.

**GS:** The *geometry field*, `wxh+x+y`. The geometry describes, in characters, the size of the logical selection and its position. The geometry should be large enough to accommodate all components of the logical selection: the optional box if the logical is optional, the title, and the button that will display the current boolean value of the logical. Geometry strings may be set to float or integer values.

**Total:** The *total* number of choices that will be offered.

**Start:** The *start* of the values to be returned. Values are always incremental integers, but the *start* field allows you to specify these values to start at any integer value. For example, if *start* is set to 1 and *total* is set to 15, this implies that you will have 15 total choices for the list; values will range from 1 to 15, inclusive. However, you may want to specify *start* as something other than one--for instance, -15, 0, or 22. In these cases, assuming *total* is still 15, ranges will go from -15 to -1, 0 to 14, or 22 to 37, respectively.

**Def:** The *default field* specifies the default value of the list. It must be set to an integer within the range of **start to total + start - 1**.

**Title:** The *title* string must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the list selection. If no title is desired, specify `' '`.

**Desc:** The *description field* should contain a short description of the string. The code generator will use this description as a comment when generating code for the command line user interface. The description must be surrounded by tic marks.

**Var:**

1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.

2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the list selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection appears on a master form, guidepane, or pane, respectively. The first will contain the integer value of the list when control is returned to the application, the second will contain the string associated with the integer value, the third has a specialized purpose, and the fourth is generated only if the selection is optional or live.

Suppose that the selection was defined on a pane defined with the variable *'pane1'*, and that the variable for the list was *'day'*. In this example,

\* *pane1\_info->day* will hold the current integer value of the list

\* *pane1\_info->day\_label* will hold the string associated with the current integer value

\* *pane1\_info->day\_struct* can be passed to *xvf\_set\_attribute()*

\* *pane1\_info->day\_selected* will be generated only if the selection is live; it will be set to TRUE when the user selects a new item from the list.

\* *pane1\_info->day\_optsel* will be generated only if the selection is optional; it will be TRUE if the user has the optional box highlighted.

Variable references are constructed similarly for list selections on the master form and guidepane.

3. **CLUI:** The *variable* name is used to determine the name of the flag that will mark this list argument to the program. Supposing that the variable for the list is *'day'*, the command line syntax will be [-day {integer value}].

4. **CLUI Code Generation:** The code generated will include three fields corresponding to the list argument, indicating whether or not the list was specified on the command line, the integer value specified, and the string associated with that integer. In our example,

\* *program->day\_flag* will be TRUE if [-day] appeared on the command line

\* *program->day\_list* will hold the integer value

\* *program->day\_label* will hold the string associated with the integer value

" **Choice\_start - Choice\_end** " Last in the (-z) line is a list of strings describing the meanings of the displaylist selection values. Each choice string must be delineated by tic marks; make sure that there are *total* strings specified, one for each integer value that the list selection may take. According to the choice made by the user, one of these strings will be returned to the application, along with the integer value representing that choice.

```
□ -z 1 0 1 0 1 4x1+1+3 12 1 1 'Month' 'month of year' month 'Jan' 'Feb' 'Mar' 'Apr' 'May' 'Jun'
    'Jul' 'Aug' 'Sep' 'Oct' 'Nov' 'Dec'
```

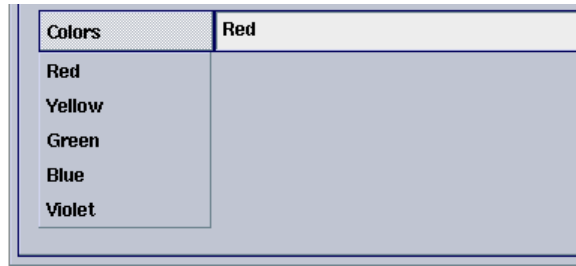
In this example, we have an optional list selection. The list may take one of seven values represented by abbreviations of days of the week. *Total* is set to 12, since there are twelve months in the year. A *start* of 1 implies that this list will represent incremental integer values starting at 1: 1, 2, 3, .... 12, respectively. The *default value* will be 1, interpreted in this case as a choice of "Jan" from the list of days.

```
□ -z 1 0 0 1 1 9x1+1+3 8 10 12 'Data Type' 'data type' data_type 'byte' 'short' 'int' 'long' 'float'
    'double' 'complex' 'double complex'
```

In this example, we have a required list selection. The list may take on one of eight values representing various data types. *Total* is set to 8, since there are eight data types listed. A *start* of 10 implies that this list will represent incremental integer values starting at 10: 10, 11, 12, ... 17, respectively. The *default value* will be 12,

interpreted that the default data type is "int".

### C.3.14. StringList (-y) Line



---

**Figure 21:** A stringlist selection offering the user one of a set of predefined colors, or the option to enter their own string.

---

The StringList (-y) line allows the user to specify a string value from either the GUI or the CLUI; when the GUI is used, the selection process is augmented by the presentation of a predefined list of strings.

#### Item Description

A stringlist allows the selection of predefined strings, or typing of a new string if the desired string is not in the predefined list. For example, you might have a "color" parameter where you know that the basic ROYGBIV colors are used most often, but you don't want to rule out the option of a less frequently used color. Stringlists may be NOT be part of a toggle, but they may be members of a mutually inclusive or mutually exclusive group. They may be optional or required. StringLists are used in the \*.form file for *xvroutines*, and in the \*.pane file for any VisiQuest program.

#### Use by the GUI

When used by the GUI, the *stringlist selection* consists of an optional box (if the selection is optional), a title, and a text box in which a string may be entered. The title is a button that will display a pulldown menu with the predefined values of the list. Live stringlist selections have the stylized carriage return symbol on the right side.

#### Use by the CLUI

When used by the CLUI, the StringList specifies a *stringlist argument*. The stringlist argument is treated identically to the string argument; the benefit of using a stringlist only applies to GUIs.

- The -y StringList line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may be part of a [-C to -E] mutually exclusive group or a [-B to -E] mutually inclusive group.
- -y Act Sel Opt OptSel Live GS Total Def 'Title' 'Desc' Var 'Choice\_start' ..... 'Choice\_end'

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the stringlist selection described by the (-y) line is to be deactivated.

**Sel:** The *selected field*, always set to 0.

**Opt:** The *optional field* is set to 1 if selection is optional, 0 if it is required.

**OptSel:** The *option selected field*; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 1 if the selection is optional and the default action is to use the default value of the selection. Use 0 if the selection is optional and the default action is not to utilize its value.

**Live:** The *live field* is set to 1 if the selection is to return immediately when the user changes the value of the logical, 0 otherwise.

**GS:** The *geometry field*,  $wxh+x+y$ . The geometry describes, in characters, the size of the logical selection and its position. The geometry should be large enough to accommodate all components of the logical selection: the optional box, if the logical is optional, the title, and the button that will display the current boolean value of the logical. Geometry strings may be set to float or integer values.

**Total:** The *total* number of predefined choices that will be offered.

**Def:** The *default field* specifies the default value of the stringlist. It must be set to an integer within the range of 1 to **total**.

**Title:** The *title string* must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the stringlist selection. If no title is desired, specify ' '.

**Desc:** The *description field* should contain a short description of the selection. The code generator will use this description as a comment when generating code for the CLUI. The description must be surrounded by tic marks.

#### **Var:**

1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the stringlist selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection appears on a master form, guidepane, or pane, respectively. The first will contain the value of the string when control is returned to the application, the second has a specialized purpose, and the third is generated only if the selection is optional or live.  
Suppose that the selection was defined on a pane defined with the variable '*panel*', and that the variable on the (-y) line was '*color*'. In this example,
  - \* *panel\_info->color* will hold the current string
  - \* *panel\_info->color\_struct* can be passed to *xvf\_set\_attribute()*
  - \* *panel\_info->color\_selected* will be generated only if the selection is live; it will be set to TRUE when the user sets a new value for the string.
  - \* *panel\_info->color\_optsel* will be generated only if the selection is optional; it will be TRUE if

the user has the optional box highlighted.

Variable references are constructed similarly for stringlist selections on the master form and guidepane.

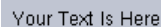
3. **CLUI:** The *variable* name is used to determine the name of the flag that will mark this string argument to the program. Supposing that the variable for the stringlist is 'color', the command line syntax will be [-color {string}].
  4. **CLUI Code Generation:** The code generated will include two fields corresponding to the stringlist argument, indicating whether or not the string was specified on the command line, and the string specified. In our example,
    - \* *program->color\_flag* will be TRUE if [-color] appeared on the command line
    - \* *program->color\_string* will hold the string
- Last in the (-y) line is a list of strings describing the meanings of the predefined stringlist values. Each string must be delineated by tic marks; make sure that there are *total* strings specified. If the predefined list is used to make a choice, the chosen string will be automatically updated in the parameter box and returned to the application.
- -y 1 0 1 0 1 15x1+1+3 6 1 'Colors' 'colors' color 'red' 'orange' 'yellow' 'green' 'blue' 'purple'

In this example, we have an optional stringlist selection. The user may choose one of the predefined color strings from the list: "red," "orange," "yellow," "green," "blue," or "purple." *Total* is set to 6, since there are six predefined choices. The default string will be "red," as indicated by the *default field* being set to 1, indicating the first choice. If the user wants to specify a color other than the ones given, they may type that choice in the GUI parameter box.

- -y 1 0 0 1 1 15x1+1+3 5 2 'Fonts' 'font type' font\_type 'fixed' 'variable' 'schoolbook' 'helvetica' 'italic'

In this example, we have a required stringlist selection. The user may choose one of the predefined color strings from the list: "fixed," "variable," "schoolbook," "helvetica," or "italic." *Total* is set to 5, since there are five predefined choices. The default string will be "variable," as indicated by the *default field* being set to 2, the second choice. If the user wants to specify a font other than the ones given, he may type in that choice in the GUI parameter box.

### C.3.15. Blank (-b) Line



Your Text Is Here

---

**Figure 22:** A blank selection is used to display text on the GUI.

---

The Blank (-b) line allows the user to specify a piece of text on the GUI. The -b Blank line may be specified

within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. In any of these, it may be a member of a [-D to -E] submenu if desired.

□ -b +x+y 'Title' Var

**+x+y:** This is the absolute position, in character widths, of the text specified in the *title field* from the upper-left corner of the pane in which it appears.

**Title:** The *Title* is the text that is to appear on the blank selection. Remember that the Blank selection is most often used for adding text to the graphical user interface--any text can be added to any pane in the graphical user interface by using Blank selections. If no title is desired, there is no need for the use of the (-b) line.

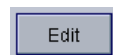
**Var:**

1. **GUI:** The *variable* name is used to give the selection a unique name so that the selection will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create a variable associated with the blank selection. This variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the selection appears on a master form, guidepane, or pane, respectively.  
Suppose that the selection was defined on a pane defined with the variable '*panel*', and that the variable for the blank was '*blk*'. In this example,  
\* *panel\_info->blk\_struct* can be passed to *xvf\_set\_attribute()* in order to change the text displayed by the blank selection  
Variable references are constructed similarly for blank selections on the master form and guidepane.

□ -b +5+6 'This is a test of the blank selection' blk

In this example, the text specified, "this is a test of the blank selection", will appear on the fifth line down and 6 character widths over from the upper-left corner of the pane in which it appears.

### C.3.16. PaneAction (-a) Line



---

**Figure 23:** A pane action lets the user request a particular action; in this case, the action will be to "edit".

---

□ The PaneAction (-a) Line specifies an action button on a pane.

## Item Description

The purpose of an action button is to return software control to the application program; it is used exclusively in the \*.form file by *xvroutines*. In general, when a pane contains two or more non-live selections, an action button should be placed at the bottom of the pane so that the user can tell the application, "I've finished setting values of selections now, go ahead and perform (some action)." Alternatively, an application may be able to perform a particular operation without any additional information provided by the user besides the fact that the user now wants the operation performed; this is the other case in which an action button is appropriate. A mouse click on the action button causes software control to be diverted from the graphical user interface to the application's subroutine that is associated with the action button, where the name of the subroutine in question is determined by the variable associated with the action button, as well as the variable associated with the pane, guide pane, or master form on which the action button appears.

## Use by the GUI

The user clicks on an action button to request a particular action from the application. Control is immediately returned from the GUI to the application so that the action may be performed.

## Use by the CLUI

The line in the UIS file representing the action button is ignored by the CLUI.

- The -a PaneAction Line may only occur in a [-P to -E] pane definition. Within the pane definition, it may be a member of a [-D to -E] submenu if desired.
- -a Act Sel GS 'Title' 'Desc' Var

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the action button described by this (-n) line is to be deactivated.

**Sel:** The *selected field* is always set to 0.

**GS:** The *geometry field*, wxh+x+y, describes, in characters, the size of the action button and its position. Note that the width field must be large enough to accommodate the title, or the title will be truncated. Geometry strings may be set to float or integer values. **Title:** The *title string* must be delineated by tic marks; it will appear as the title on the action button.

**Desc:** The *description field* should contain a short description of the purpose of the action button.

## Var:

1. **GUI:** The *variable* name is used to give the action button a unique name so that the button will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create two variables associated with the action button. These variables will appear in the PaneInfo structure as defined in "form\_info.h".  
Suppose that the action button was defined on a pane defined with the variable '*panel*', and that the variable for the action button was '*do\_it*'. In this example,

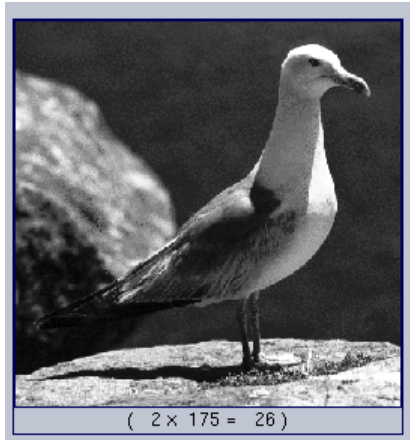
- \* *panel\_info->do\_it\_struct* can be passed to *xvf\_set\_attribute()*
- \* *panel\_info->do\_it* will be TRUE if the user clicked on the action button.

□ -a 1 0 15x2+12+15 'Do It' 'do something' do\_it

In the example, we have a pane action button, 15 characters wide by 2 characters tall, located 12 characters to the right and 15 characters down from the upper-left corner of the master form, labeled "Do It," indicating that when the user clicks on this button, the application will "do it," whatever "it" may be.



### C.3.17. Workspace (-w) Line



---

**Figure 24:** A workspace selection provides a general-purpose area on the GUI for whatever purpose may be appropriate for the application. Here, a workspace selection is used by `editimage` to hold the displayed image.

---

The Workspace (-w) line allows the developer to create a general purpose area for image, graphics, or special-purpose GUI display.

#### Item Description

The Workspace provides a general purpose manager widget on the GUI which may be used as a backplane for display of images, graphics, or special-purpose GUI elements that are not provided directly. The workspace is used exclusively in the \*.form files of *xvroutines*.

#### Use by the GUI

The application may use the workspace for whatever purpose may be applicable. It has absolutely no functionality on its own; the application is provided with the address of the requested widget, and is then responsible for the use and maintenance of this widget. Once the workspace widget is displayed, the GUI does not manage or interfere with it further.

#### Use by the CLUI

The line in the UIS file representing the workspace is ignored by the CLUI.

- The -w Workspace line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition.
- -w GS 'Desc' Var

**GS:** The *geometry field*, `wxh+x+y`, describes, in characters, the size of the workspace widget and its position on the master form, guide pane, or pane in *pixels* (note that it is the only line in the UIS to have a geometry

string in terms of pixels, and not characters). The geometry string for a (-w) line must have integer values.

**Desc:** The *description field* should contain a short description of the purpose of the workspace.

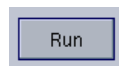
**Var:**

1. **GUI:** The *variable* name is used to give the workspace a unique name so that the workspace will work properly with concert and journal playback.
2. **GUI Code Generation:** The *variable* name is also used by the xvroutine code generator to create variables associated with the workspace. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h", depending on whether the workspace appears on a master form, guidepane, or pane, respectively. The first will contain the widget address of the workspace and the second has a specialized purpose. Suppose that the workspace was defined on a pane defined with the variable '*panel*', and that the variable for the workspace was '*canvas*'. In this example,
  - \* *panel\_info->canvas* will hold the widget address
  - \* *panel\_info->canvas\_struct* can be passed to *xvf\_set\_attribute()*Variable references are constructed similarly for workspaces on the master form and guidepane.

□ -w 1 0 500x500+2+5 workspace

In the example, we have a workspace widget that is 500 pixels square in size. It can be used for anything desired by the application programmer. It is located 2 characters down and 5 characters across from the upper-left corner of the master form, guide pane, or pane, depending on whether the line appears in the [-S to -E] master definition, [-G to -E] guide pane definition, or in the [-P to -E] pane definition. It is labeled with the title, "Workspace," 10 characters to the right and one character down from the upper-left corner of the widget itself.

### C.3.18. Routine (-R) Line



---

**Figure 25:** The routine button is used in \*.pane files as part of the *cantata* GUI for a VisiQuest program. Commonly labelled, "Run", or "Execute", the routine button executes the program in question, using the values of the other selections on the pane to determine the arguments to pass the program.

---

□ The (-R) line specifies a Routine Button.

#### Item Description

The Routine Button is used in the \*.pane UIS file for all VisiQuest programs so that they may be

integrated into the VisiQuest visual language. Routine buttons are used exclusively in the \*.pane files of VisiQuest programs.

### Use by the GUI

When the user clicks on this button, the specified program is immediately executed with the arguments specified by the values of all other selections on the pane.

### Use by the CLUI

The line in the UIS file representing the routine button is ignored by the CLUI.

- The **-R** Routine line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition.
- **-R** Act Sel Exec\_Type GS 'Title' 'Desc' Routine

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the logical selection described by the (-R) line is to be deactivated.

**Sel:** The *selected field*, always set to 0.

**Exec\_Type:** The *exec\_type* attribute indicates to VisiQuest whether or not its output should be piped to the expression parser so that those values can be used as variables in a VisiQuest workspace. When the user specifies that the value is to be written to a variable, a the value of the variable is written to stdout. The **kprval** kroutine is a good example of this model. At this time, only two values are supported for the *exec\_type*:

- 1: VisiQuest will not pipe output to expression parser
- 4: VisiQuest will pipe output to expression parser

For those who may not be sure which *execution type* their routine should have, a value of 1 is recommended.

**GS:** 6 The *geometry field*, wxh+x+y. The geometry describes, in characters, the size of the logical selection and its position. The geometry should be large enough to accommodate all components of the logical selection: the optional box, if the logical is optional, the title, and the button that will display the current boolean value of the logical. Geometry strings may be set to float or integer values.

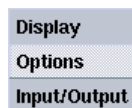
**Title:** The *title string* must be delineated by tic marks; in the case of the (-R) line, the title is usually "Execute", indicating to the user that when they click on the *Routine* button, the routine in question will be executed.

**Desc:** The *description field* is unused in the (-R) line. This is the program that will be executed when the user clicks on the Routine button. It should be specified using the name of the toolbox bin in which the program binary is located and the name of the binary itself, using the syntax \${TOOLBOXNAME}BIN/{binary name}. For example, the karith1 program of the \$DATAMANIP toolbox has its Routine attribute specified as \$DATAMANIPBIN/karith1.

- **-R** 1 0 1 8x2+1+6 'Execute' 'run kconvert' \$DATAMANIPBIN/kconvert

In this example, we are creating a *Routine* button that will execute the **vconvert** routine when clicked on by the user. The title of the routine button will be "Execute", so as to match practically every other routine button currently existing in VisiQuest. We have set the *exec\_type* field to 1 so that only one of these processes may be running at any particular time.

### C.3.19. StartSubMenu (-D) Line



---

**Figure 26:** A submenu is created from button selections and blank selections.

---

- Collectively, the [-D to -E] pane SubMenu definition specifies a pulldown menu, where a single button specified by the -D line represents the menu until it is activated. The (-D) line *must* be matched by an End (-E) line to end the group of buttons making up the menu. Only buttons and labels are allowed within a SubMenu definition.

#### Item Description

When many items are accessible from a particular master form, guidepane, or pane of the GUI, one can reduce clutter by grouping buttons and labels together onto a pulldown menu. Any group of selections that are made up of a *single* button or label may be collected into a submenu; thus, candidates for menu contents include: subform buttons (for submenus on master forms only) and guide buttons (for submenus on guide panes only) as well as action buttons, quit buttons, help buttons, and blank selections on any part of the GUI. It is most often used in the \*.form file for an *xvroutine*. It may also be used in the \*.pane file for any VisiQuest program, although by convention it is used sparingly in this context.

#### Use by the GUI

The submenu button presents a single button on the GUI; clicking on the button will display a pulldown menu from which any of the other buttons may be selected. Other buttons that are grouped into a submenu will retain their original function.

#### Use by the CLUI

The line in the UIS file representing the submenu is ignored by the CLUI.

- The -D StartSubMenu line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition.
- -D Act Sel GS 'Title' Var

**Act:** The *activation* field is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the submenu button described by this (-D) line is to be deactivated.

**Sel:** The *selected field*, always set to 0.

**GS:** The *geometry field*,  $wxh+x+y$ , describes, in characters, the size of the submenu button and its position on the master. Note that the width field must be large enough to accommodate the title, or the title will be truncated. Geometry strings may be set to float or integer values.

**Title:** The *title string* must be delineated by tic marks; it will appear as the title on the submenu button.

**Var: GUI:** The *variable* is used internally by the *xvforms* library for book-keeping.

**GUI Code Generation:** The *variable* name is also used by the *xvroutine* code generator to create a variable associated with the submenu. This variable will appear in the *FormInfo*, *SubformInfo*, or *PaneInfo* structure as defined in "form\_info.h," depending on whether the submenu appears on a master form, guidepane, or pane, respectively.

Suppose that the submenu was defined on a pane defined with the variable '*panel*,' and that the variable for the submenu was '*pulldown*.' In this example,

\* *panel\_info->pulldown\_struct* can be passed to *xvf\_change\_gui()* in order to change the attributes of the submenu.

Variable references are constructed similarly for submenu definitions on the master form and guidepane.

**CLUI:** The variable of the submenu is ignored by the CLUI.

**CLUI Code generation:** The UIS line corresponding to the submenu is not used by the code generator.

□ -D 1 0 10x5+4+3 'Output' outpt

In the example, we are creating a submenu button that is 10 characters wide and 5 characters high, located 4 characters to the right and 3 characters down from the upper-left corner of the master on which it appears. It will be labeled "Output," implying that the subforms accessible via this pull-down submenu will perhaps allow the user to perform different types of output.

### C.3.20. IncludeSubform (-k) Line

□ -k Filename

□ The full path and filename of the file that contains the subform [-M to -E] definition to be included at the place where the (-k) line occurs. Note that use of the tilde (~) and the \$ENVISION or \$TOOLBOX environment variables are accepted.

□ -k \$DESIGN/objects/xvroutine/VisiQuest/uis/preferences.subform

In this example, we substitute the single (-k) line for the entire [-M to -E] subform definition that otherwise would have appeared in the UIS at this point. In the file found in \$DESIGN/repos/VisiQuest/subforms/clas-

sify/console.subform, the subform definition must appear.

### C.3.21. IncludePane (-p) Line

- -p Filename
- The full path and filename of the file that contains the pane [-P to -E] definition to be included at the place where the (-p) line occurs. Note that use of the tilde (~) and the \$DESIGN or \$TOOLBOX environment variables are accepted.
- -p \$DESIGN/objects/xvroutine/craftsman/uis/script.pane

In this example, we substitute the single (-p) line for the entire [-P to -E] pane definition that otherwise would have appeared in the UIS at this point. In the file found in \$DESIGN/objects/xvroutine/craftsman/uis/script.pane, the entire pane definition must appear.

### C.3.22. Stdin (-e) Line



---

**Figure 27:** A stdin selection is used only on pane objects that are being used to integrate non-VisiQuest programs (that are dependent on stdin for input) into the VisiQuest system.

---

- The Stdin (-e) line allows the user to specify that input should be taken from standard input.

#### Item Description

VisiQuest can be used as an *integration system*. This is appropriate when there exists a number of non-VisiQuest programs upon which one wishes to enforce standardized documentation, a consistent user interface, and accessibility from the VisiQuest visual language, VisiQuest. The procedure that is followed when one is doing such an integration is to create a *pane object* for each program that is to be integrated; a pane object provides a VisiQuest GUI for each non-VisiQuest program. The stdin and stdout selections provide a mechanism whereby non-VisiQuest programs depending on stdin and stdout may be integrated into VisiQuest.

It is important to understand that programs written under the VisiQuest development system do not depend on stdin or stdout; instead, they have a formalized command line user interface where input and output files are specified using input file and output file arguments, as in "-i image:ball" or "-o my\_image.viff".

However, if the programs to be integrated into VisiQuest depend on stdin and stdout for input and output, then some mechanism must be provided from within VisiQuest to accommodate them. The stdin

and stdout selections were created for this reason.

### Use by the GUI

The stdin and stdout selections are only used in the \*.pane files for *pane objects* that are created in order to integrate non-VisiQuest, stdin- and stdout-dependent programs into VisiQuest.

### Use by the CLUI

The lines in the UIS file representing stdin and stdout are ignored by the CLUI.

- The -e Stdin line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition.
- -e Act Opt OptSel +x+y 'Title' Var

**Act:** The *activation* field is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the stdin selection described by the (-e) line is to be deactivated.

**Opt:** The *optional* field is set to 1 if the selection is optional, 0 if it is required.

**OptSel:** The *option selected* field; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 0 if the selection is optional and the default action is not to utilize its value. **+x+y:** The (x,y) position of the stdin selection within its parent, where the position is specified in floating point character widths and heights, from the upper-left corner of the parent.

**Title:** The *title* string must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the stdin selection. If no title is desired, specify ' '.

### Var:

1. **GUI:** The variable name is not used in the GUI.
2. **GUI Code Generation:** The *variable* name is used by the xvroutine code generator to create a variable associated with the stdin selection. This variable will appear in the PaneInfo structure as defined in "form\_info.h".  
Suppose that the stdin selection was defined on a pane defined with the variable '*pane1*', and that the variable for the stdin selection was '*stdin*'. In this example, *pane1\_info->stdin\_struct* can be passed to *xvf\_set\_attribute()*
3. **CLUI:** The stdin selection causes stdin to be used as input to the program represented by the pane object.

- -e 1 0 1 +1+4 'Stdin' stdin



In this example, we have a required stdin selection; the title of this stdin selection is "Stdin." The name of the field in the Form Information structure generated by the xvroutine code generator will be identifiable by the letters "stdin."

### C.3.23. Stdout (-o) Line



---

**Figure 28:** A stdout selection is used only on pane objects that are being used to integrate non-VisiQuest programs (that are dependant on stdout for output) into the VisiQuest system.

---

- The Stdout (-o) line allows the user to specify that output should be taken from standard output.

#### Item Description

VisiQuest can be used as an *integration system*. This is appropriate when there exists a number of non-VisiQuest programs upon which one wishes to enforce standardized documentation, a consistent user interface, and accessibility from the VisiQuest visual language, VisiQuest. The procedure that is followed when one is doing such an integration is to create a *pane object* for each program that is to be integrated; a pane object provides a VisiQuest GUI for each non-VisiQuest program. The stdin and stdout selections provide a mechanism whereby non-VisiQuest programs depending on stdin and stdout may be integrated into VisiQuest.

It is important to understand that programs written under the VisiQuest development system do not depend on stdin or stdout; instead, they have a formalized command line user interface where input and output files are specified using input file and output file arguments, as in "-i image:ball" or "-o my\_image.viff".

However, if the programs to be integrated into VisiQuest depend on stdin and stdout for input and output, then some mechanism must be provided from within VisiQuest to accommodate them. The stdin and stdout selections were created for this reason.

#### Use by the GUI

The stdin and stdout selections are only used in the \*.pane files for *pane objects* that are created in order to integrate non-VisiQuest, stdin- and stdout-dependent programs into VisiQuest.

#### Use by the CLUI

The lines in the UIS file representing stdin and stdout are ignored by the CLUI.

- The -o Stdout line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition.

□ -o Act Opt OptSel +x+y 'Title' Var

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the stdout selection described by the (-o) line is to be deactivated.

**Opt:** The *optional field* is set to 1 if selection is optional, 0 if it is required.

**OptSel:** The *option selected field*; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the selection is required. Use 1 if the selection is optional and the default action is to use the default value of the selection. Use 0 if the selection is optional and the default action is not to utilize its value. **+x+y:** The (x,y) position of the stdout selection within its parent, where the position is specified in floating point character widths and heights, from the upper-left corner of the parent.

**Title:** The *title string* must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the stdin selection. If no title is desired, specify ''.

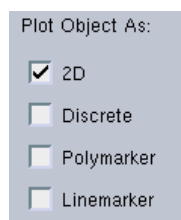
**Var:**

1. **GUI:** The variable name is not used in the GUI.
2. **GUI Code Generation:** The *variable* name is used by the xvroutine code generator to create a variable associated with the stdout selection. This variable will appear in the PaneInfo structure as defined in "form\_info.h". Suppose that the stdout selection was defined on a pane defined with the variable '*pane1*', and that the variable for the stdout selection was '*stdout*'. In this example, *pane1\_info->stdout\_struct* can be passed to *xvf\_set\_attribute()*
3. **CLUI:** The stdout selection causes stdout to be used as output from the program represented by the pane object.

□ -e 1 0 1 +1+4 'Stdout' stdout

In this example, we have a required stdout selection; the title of this stdout selection is "Stdout." The name of the field in the Form Information structure generated by the xvroutine code generator will be identifiable by the letters "stdout."

### C.3.24. Toggle (-T) Line



**Figure 29:** This toggle of flags allows a user to select the desired plot type.

---

**Explanation:**

Collectively, the [-T to -E] Toggle definition specifies a toggle. The (-T) line *must* be matched by an End (-E) line to end the group of selections that will make up the toggle. Legal toggle members include InputFiles, OutputFiles, Flages, Logicals, Integers, Floats, Doubles, and Strings. ALL toggle members must be of the same type. The data type of the toggle members dictate the type of toggle as a whole. Toggle values may be specified by number; toggle members are always numbered from 1 to N, where N is the number of members in the toggle. Altrnatively, toggle values may be specified by their value, where the data type of that value is determined by the type of toggle in question.

**Item Description**

The Toggle provides a method of allowing the user to choose between a finite number of pre-defined choices, where all choices are of the same data type. Supported toggle types include InputFiles, OutputFiles, Integers, Logicals, Flags, Floats, and Strings. Usually, the value of the toggle as a whole is determined by the default value of the chosen toggle member. For example, if a toggle has three string members with default values of "red," "green," and "blue," the value of the toggle as a whole will be "red" when the first toggle member is selected. The exception to this is with Logical and Flag toggles, where the value of the toggle is the integer number of the toggle member; for example, if a flag toggle has 5 members and the fifth member is selected, the value of the flag toggle will be 5. It may be used in the \*.form file for an *xvroutine*, or in the \*.pane file for any VisiQuest program.

**Use by the GUI**

On the GUI, a *toggle selection* consists of an optional box (if the toggle as a whole is optional), a title, and a backplane containing its toggle members. Each toggle member consists of an optional box and a title. When the optional box of a toggle member is highlighted, then that toggle member determines the value of the toggle as a whole.

**Use by the CLUI**

On the CLUI, a toggle forces the user to enter one of a set of predefined choices, either by value or by title. For example, if a toggle has three string members with default values of "red," "green," and "blue," the user will have to enter "red" or "1" for the first option, "green" or "2" for the second option, and so on. They will not be allowed to enter anything other than the predefined choices. The program will have access to both the number and the label associated with toggle value entered.

**UIS File Placement**

The -T Toggle line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition.

**Use by the CLUI**

A toggle may be used on the command line to allow the user to provide one of a pre-specified group of values to the program.

**syntax:**

-T Act Sel Opt OptSel Live GS TOffset Def 'Title' 'Desc' Var

**Act:** The *activate field* is set to 1 (TRUE) or 0 (FALSE). Only set it to 0 if the entire toggle is to be deactivated.

**Sel:** The *selected field* is always set to 0.

**Opt:** The *optional field* is set to 1 if use of the toggle is optional, 0 if it is required.

**OptSel:** The *option selected field*; a 1 turns the optional box on, while a 0 turns it off. Use 1 if the toggle is required. Use 1 if the toggle is optional and the default action is to use the default value of the toggle. Use 0 if the toggle is optional and the default action is not to utilize its value.

**Live:** The *live field* is set to 1 if control is to be returned immediately to the application when the user changes the value of the toggle, 0 otherwise.

**GS:** The *geometry field*, wxh+x+y. The geometry describes, in characters, the size of the toggle and its position. The geometry should be large enough to accommodate all items within the toggle set. Geometry strings may be set to float or integer values.

**TOffset:** The *title offset field*, +x+y, specifies, in characters, the distance from the upper-left corner of the backplane of the toggle that the title should appear.

**Def:** The *default field* specifies which of the selections in the toggle set will provide the default value for the toggle. This *default field* on the (-T) line will be the NUMBER OF THE DESIRED SELECTION, where the number of the selection is determined by ORDER OF ITS APPEARANCE in the toggle set. For example, if the third selection in the toggle set is to provide the default value, the *default field* must be set to 3. Selections are numbered from 1 to N, where N is the total number of selections in the toggle.

**Title:** The *title string* must be delineated by tic marks; it should be as short as possible, yet should still give a clear idea as to the purpose of the toggle set. If no title is desired, specify ' '.

**Desc:** The *description field* should contain a short description of the toggle. The code generator will use this description as a comment when generating code for the command line user interface. The description must be surrounded by tic marks.

**Var:**

1. **GUI:** The *variable name* is used to give the toggle a unique name for internal bookkeeping and as a reference in app-defaults files. playback.
2. **GUI Info Structure:** The *variable name* is also used by the xvroutine code generator to create variables associated with the toggle selection. These variables will appear in the FormInfo, SubformInfo, or PaneInfo structure as defined in "form\_info.h," depending on whether the toggle appears on a master form, guidepane, or pane, respectively. The first will contain the value of the toggle when control is returned to the application; the type of the variable will vary according to

the type of toggle. The second has a specialized purpose, and the third is generated only if the toggle is optional or live.

Suppose that a toggle was defined on a pane defined by a (-P) line with the variable 'panel,' and that the variable on the (-T) line was 'datatype.' In this example,

\* *panel\_info->datatype\_val* holds the string value of a string toggle, the filename value of an input file or output file toggle, the int value of a flag, logical, or integer toggle, or the float value of a float toggle.

\* *panel\_info->datatype\_num* holds the number of the currently chosen selection, where the number of the selection is determined by the order of its appearance in the toggle set. This number will range from 1 to N, where N is the total number of selections in the toggle.

\* *panel\_info->datatype\_struct* can be passed to \* *xvf\_change\_gui()*

\* *panel\_info->datatype\_selected* will be generated only if the toggle is live; it will be set to TRUE when the user changes the value of the toggle.

\* *panel\_info->datatype\_optsel* will be generated only if the selection is optional; it will be TRUE if the user has the optional box highlighted.

Variable references are constructed similarly for toggles on the master form and guidepane.

3. **CLUI Arguments:** The *variable* name is used by the code generator to determine the name of the flag that will mark this toggle argument to the program. Supposing that the variable on the (-T) line is 'datatype', the command line syntax will be [-plot\_type {filename/string/integer/float}], where the data type of the argument is determined by the type of toggle used.

The code generated for the CLUI will include two fields corresponding to the toggle argument, indicating whether or not the toggle was specified on the command line, and the integer value specified. In our example,

\* *program->datatype\_flag* will be TRUE if [-func] appeared on the command line

\* *program->datatype\_toggle* will hold the string value of a string toggle. It will hold the filename value of an input file or output file toggle. It will hold the integer value of a flag, logical, or integer toggle. It will store the float value of a float toggle.

**example:**

```
-T 1 0 0 1 0 20x1+1+5 +0+0 4 'Plot Type' 'type of plot' plot_type
```

In this example, we are describing a toggle that will give us the desired plot type. From the (-T) line alone, one cannot tell if this will be an integer, logical, float, or string toggle--this will be decided by the type of selections that follow, although we can probably assume from the title that it will be an integer toggle (if plot types are #defined) or a string toggle (if we will be using the names of the plot types). The default is 4, indicating that the fourth selection to follow the (-T) line will provide the default in its *default* field.

### C.3.24.1. Notes on Toggle Use

**Initializing the Toggle:**

All selections within the toggle definition must have their *optional* fields set to TRUE (1).

*One and only one* selection within the toggle must have its *option selected field* = 1. All others *must* have this field set to 0. The one selection that does have its *option selected* field set to 1 will be the default choice of the toggle; the number of this selection **MUST** match the *default field* on the (-T) line. For example, if the (-T) line has 4 in its default field, the 4th UIS line in the toggle definition may be the only one with its *option selected* set to 1.

For toggles of strings, input files, and output files, the label of each string selection should be **IDENTICAL** to its default value.

### **Value Returned by the Toggle:**

The *toggle value* that is returned to the application program is the same type as the items inside the toggle. Thus, for a float toggle, the toggle value returned from both the CLUI and the GUI will be a float having the value of the *default field* specified for the float selection that is currently selected.

For toggles of logicals and flags, the value returned will be the number (in order of appearance) of the logical or flag that is currently selected. This is because the values that can be represented by logicals and flags (0 and 1) do not offer enough choices to be very useful for providing values for a toggle with more than 2 members. Therefore, in a toggle of logicals or flags, if the Nth member is selected, the toggle will take on a value of N.

### **Variables of Toggle Members:**

For all selections that are toggle members, the variable name **DOES NOT MATTER**; they may all be the same.

### **Geometry Strings of Toggle Members:**

All selections within the toggle have the *x* and *y* values of the *geometry string* **RELATIVE** to the geometry on the (-T) line. This is in contrast to every other case, where all *geometry strings* use absolute values. Geometry strings may be set to float or integer values.

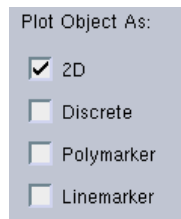
### **Default Values of Toggle Members:**

Don't forget that it is the *default fields* of each selection within the toggle definition that provide the possible values for the toggle. Therefore, the default fields on the selections that make up the toggle must be set carefully!

## **C.3.24.2. Toggle of Flags**

A *toggle of flags* is a toggle which returns integer values of 1 to N, where N is the total number of flag selections in the toggle. The following UIS file excerpt shows an example of a flag toggle:

```
-T 1 0 0 1 0 20x1+1+5 +0+0 4 'Plot Type' 'type of plot' plot_type
-t 1 0 1 0 0 10x1+0+1 '3D' '3D plot' dummy
-t 1 0 1 0 0 10x1+9+1 'Scatter' 'scatter plot' dummy
-t 1 0 1 0 0 10x1+18+1 'Mesh' 'mesh plot' dummy
-t 1 0 1 1 0 10x1+27+1 'Horizon' 'horizon plot' dummy
-t 1 0 1 0 0 10x1+36+1 'Contour' 'contour plot' dummy
```




---

**Figure 30:** This toggle of flags allows a user to select the desired plot type.

---

This is one way in which a toggle might be used to define a variable which will describe a plot type. The toggle (i.e, the integer "plot\_type" variable) will take on values of 1, 2, 3, 4, or 5, depending on which of the flags is selected. In this situation, it is often simplest to have #defines designating a 3D Plot as 1, a Scatter Plot as 2, and so on.

The *variable fields* on the flags will not be used--we put in "dummy" to remind us of this fact. We 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, so everything is fine--our default will be 4. Finally, we check the geometry, to make sure we have specified each of the y fields relative to the (-T) line; the geometry of the flags will specify 2 rows of toggle items, the top row with three selections, and the bottom row with two selections.

### C.3.24.3. Toggle of Logicals

A *toggle of logicals*, like the toggle of flags, is a toggle which returns integer values of 1 to N, where N is the total number of logical selections in the toggle. The logical toggle operates in exactly the same way as a flag toggle. The following UIS file excerpt shows an example of a logical toggle:

```
-T 1 0 0 1 0 20x1+1+5 +0+0 1 'Plot Type' 'type of plot' plot_type
-1 1 0 1 1 0 10x1+0+1 0 '3D' 'False' 'True' '3D plot' dummy
-1 1 0 1 0 0 10x1+9+1 0 'Scatter' 'False' 'True' 'scatter plot' dummy
-1 1 0 1 0 0 10x1+18+1 0 'Mesh' 'False' 'True' 'mesh plot' dummy
-1 1 0 1 0 0 10x1+27+1 0 'Horizon' 'False' 'True' 'horizon plot' dummy
-1 1 0 1 0 0 10x1+36+1 0 'Contour' 'False' 'True' 'contour plot' dummy
-E
```




---

**Figure 31:** The toggle of logicals defined by the UIS file excerpt above looks and acts identically to the toggle of flags given in the first example.

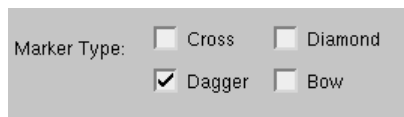
---

The toggle definition shown above produces exactly the same functionality as the flag toggle shown in the previous example; therefore, we will not reiterate the same points made earlier. The use of the flag toggle is preferable to the use of a logical toggle because the syntax of Flag UIS lines is simpler. Also, the flag toggle is ever so slightly more efficient. Toggles of logicals are provided mainly for backwards compatibility with VisiQuest 1.0 and 2.0.

### C.3.24.4. Toggle of Integers

A *toggle of integers* is more flexible than the toggle of flags, since a toggle of integers is not restricted to returning values between 1 and N. In contrast, an integer toggle may be designed to return any set of integers. The integers specified may be positive or negative, and may or may not be in a contiguous range.

```
-T 1 0 0 1 0 20x1+1+5 +0+0 2 'Marker Type:' 'marker type' marker_type
-i 1 0 1 0 0 15x1+12+0 0 0 21 0 1 'Cross' 'cross' dummy
-i 1 0 1 1 0 15x1+12+1 0 0 23 0 1 'Dagger' 'dagger' dummy
-i 1 0 1 0 0 15x1+22+0 0 0 28 0 1 'Diamond' 'diamond' dummy
-i 1 0 1 0 0 15x1+22+1 0 0 17 0 1 'Bow' 'bow' dummy
-E
```




---

**Figure 32:** The toggle of integers defined by the UIS file excerpt is appropriate when the integer values associated with the toggle are not in a contiguous range, or when they start at a value not equal to 1.

---

Here, an integer toggle is used to define the `marker_type` variable. Suppose that the different marker types in the application have been given `#define`'d values. However, in contrast to the examples given for the logical and flag toggles, here they do not begin at 1 and increment consecutively; instead, the Cross Marker is `#defined` to 21, the Dagger Marker is `#defined` as 23, and so on. Marker types are not `#define`'d in a contiguous range. This situation calls for an integer toggle, since the value of the `marker_type` toggle will take on the value of the default field of the chosen item, which can be set as desired to our `#define`'d numbers.

This toggle is optional, and since the option selected field on the (-T) line is set to 1, the default action will be to use this toggle unless the users specifically indicate that they do not wish to use it. This time, the default field of the (-T) line is set to 2, and accordingly, it is the second (-i) line that has its *option selected* field set to 1; the default of this toggle, therefore, will be 23. The position given by the *y* value of the relative *geometry strings* of the (-i) lines indicates that the items in the toggle will be aligned along the left hand side of the pane, starting with the toggle title "Marker Type" at an area 5 characters down from the top of the pane. Since bounds on the integers will not come into play, they are set to 0.



### C.3.24.5. Toggles of Floats & Doubles

Toggles of floats or doubles are useful when a variable may have one of several floating point or double precision values.

```
-T 1 0 0 1 0 15x4+1+9 +0+0 4 'Float Toggle' 'Float toggle selection' toggle1
-f 1 0 1 0 0 10x1+0+1 0 0 -6.2831853 0 0 0 '-2*pi' '-2*pi' dummy
-f 1 0 1 0 0 10x1+8+1 0 0 -3.1315927 0 0 0 '-pi' '-pi' dummy
-f 1 0 1 0 0 10x1+16+1 0 0 0 0 0 0 '0' '0' dummy
-f 1 0 1 1 0 10x1+24+1 0 0 3.1315927 0 0 0 'pi' 'pi' dummy
-f 1 0 1 0 0 10x1+32+1 0 0 6.2831853 0 0 0 '2*pi' '2*pi' dummy
-E
```



---

**Figure 33:** This toggle of floats offers a choice of various multiples of pi.

---

In this example, a required toggle lets the user choose one of various multiples of pi. By default, the value of the toggle is 3.1315927, the default value of the fourth toggle member. Float toggles work the same as integer toggles except that they take on float values. The toggle is not "live," so the user will have to click on an action button or a "live" selection before software control is returned to the application. Double toggles work identically to float toggles except that they use double precision values.

### C.3.24.6. Toggle of Strings

A *toggle of strings* is used when a variable is needed which will take on the value of one of a predefined set of strings. The strings specified may be

```
-T 1 0 0 1 1 55x2+1+4.5 +0+0 1 'Plot Type' 'type of plot' plot_type
-s 1 0 1 1 0 'String' 25x1+2+1 '2D Plot' '2D Plot' '2d plot' dummy
-s 1 0 1 0 0 'String' 25x1+30+1 'Discrete' 'Discrete' 'discrete plot' dummy
-s 1 0 1 0 0 'String' 25x1+2+2 'Histogram' 'Histogram' 'histogram plot' dummy
-s 1 0 1 0 0 'String' 25x1+30+2 'Polymarker' 'Polymarker' 'polymarker plot' dummy
-E
```



---

**Figure 34:** The string toggle defined here is nice when one of a predefined set of strings is to be chosen by the user.

---

In this example, we have a required toggle that is "live;" that is, as soon as the user changes the value of the toggle, control will be immediately returned to the application program so that it can take some kind of action in response to the change of plot type. This is a toggle of strings: the value returned in the "plot\_type" variable will be a string, and it will have as its value the *default* field of one of the string selections: "2D Plot," "Discrete," "Histogram," or "Polymarker." To avoid confusion, in each string selection, the *title* is identical to the *default*. The default of this string toggle is the *default* of the first string selection, or "2D Plot".

### C.3.24.7. Toggles of Input & Output Files

Toggles of input or output files are nice when the user may need to select between a number of filenames.

```
-T 1 0 0 1 1 38.5x5.81818+2+1.5 +0+0 1 'List of Example 24-bit (RGB) Images:' 'lists supplied
-O 1 0 1 1 0 1 15x1+3+1 'image:lizard-rgb' 'Happy Lizard (xvimage)' 'input image filename'
-O 1 0 1 0 0 1 15x1+3+2 'image:mandril-rgb' 'Famous Mandrill (xvimage)' 'input image filename'
-O 1 0 1 0 0 1 15x1+3+3 'image:colortest' 'Color Bars (ppm)' 'input image filename'
-E
```



**Figure 35:** This toggle of output files offers a choice of various VisiQuest images. Such toggles are used often as data glyphs in Cantata.

In this example, a required toggle of input files lets the user choose one of the standard input images that are distributed with VisiQuest. The default is the "ball" image. The default value of the chosen toggle member will determine the value of the toggle as a whole; for example, if the user selects 'wolf.xv,' the filename returned to the program will be '\$SAMPLEDATA/data/images/wolf.xv.'

Types of Toggles	
<i>Type</i>	<i>Use</i>
Flag Toggle	Used when values are incremental integers beginning at 1. Preferred over Logical toggles. The most common toggle type.
Logical Toggle	Same as Flag toggle; used when values are incremental integers beginning at 1. Provided for backwards compatibility with VisiQuest 1.0.5.
Integer Toggle	Used when values are non-incremental integers, or when incremental integers begin at a number other than 1. Common when representing #define'd values which are not necessarily incremental.
Float Toggle	Used when values are floating point. Common with toggles involving mathematical constants such as pi.

### C.3.25. MutExcl (-C) Line

#### Explanation:

Collectively, the [-C to -E] definition specifies a mutually exclusive group. A mutually exclusive group allows the user to specify a value for one and only one of its members. The (-C) line *must* be matched by an End (-E) line to end the group of selections that will make up the group. Legal group members include InputFiles, OutputFiles, Integers, Floats, Strings, Cycles, Lists, Toggles, Flags, StringLists, and Mutually Inclusive groups. Nesting of other Mutually Exclusive groups is prohibited, because it is redundant. Group members may be of different types. Group members **MUST** be optional.

#### Use by the GUI

Selections within the group are created as usual; however, only one of the optional selections may be chosen at any one time.

#### UIS File Placement

The -C MutExcl line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. It may also be nested (one level of nesting only) inside a Mutually Inclusive [-B to -E] group, or inside another Mutually Exclusive [-C to -E] group.

#### Use by the CLUI

A mutually exclusive group may be used on the command line to allow the user to provide one and only one of a group of arguments to the program.

#### Initializing the ME Group:

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.

Like the toggle, **ONE** and **ONLY ONE** selection within the group must have its *option selected* field set to 1. This selection will be the default choice of the mutually exclusive group. Unlike the toggle, there is no *default* field on the (-C) line that must match the number of the selection.

ALL selections in the group must have a valid default value provided if the *required* field is FALSE (0). As always, lower and upper bounds on integers and floats should be set carefully.

#### syntax:

-C Required

**Required:** The *required* field is 0 if the mutually exclusive group *as a whole* is optional, 1 if it is required. In other words, a 1 in the *required* field says, "the user **MUST** provide **ONE** of these selections," whereas a 0 in the *required* field says, "the user **MAY** provide **ONE** of these selections if desired."

### example 1:



**Figure 36:** A mutually exclusive group containing an input file, an integer, and a float. Currently, the "Real Constant" float parameter is selected for use; the other two selections are desensitized to emphasize that they will not be used.

```
-C 1
-I 1 0 1 1 0 1 45x1+1+13 './' 'Input Path' 'input path' inpath
-O 1 0 1 0 0 1 45x1+1+14 './' 'Output Path' 'output path' outpath
-s 1 0 1 0 0 45x1+1+15 './' 'Other String' 'string' string
-E
```

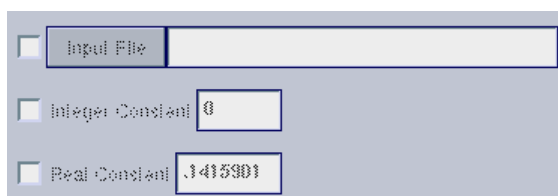
This mutually exclusive group is required; that is, the user **MUST** provide a value for one and only one of the items. On the command line, users will be prompted repeatedly until they provide a value for one of the three; on the graphical user interface, they will be forced to have one of the three with its optional box turned on at all times.

Defaults for items in the group are not used on the command line; the user will be forced to specify a value for one of the selections and will have to explicitly enter a value, even if they want the "." directory specified as the default for each.

Defaults for items do appear in the GUI; each selection will initially have the default "." in the parameter box.

Other things to notice about this example are that none of the items in the group are "live;" in a mutually exclusive group, one should never have "live" selections, as this interferes with the operation of the group as a whole. Also, notice that as opposed to a toggle group, items in a mutually exclusive group do not have relative positioning. As in every other part of the UIS (except toggle groups), positioning indicated by the geometry strings is absolute. In addition, see how the positioning of the items keeps them in a physical as well as operational group. Items in a mutually exclusive group should never be separated from one another by another (isolated) item, as this makes a graphical user interface confusing to the user.

### example 2:



**Figure 37:** This mutually exclusive group is the same as that in example 1, except that it is

optional. Because it is optional, the user is not *forced* to choose one of the three selections. Currently, none of the three are selected; because of this, they are all de-sensitized.

---

```
-C 0
-I 1 0 1 1 0 1 45x1+1+13 './' 'Input Path' 'inpath' inpath
-O 1 0 1 0 0 1 45x1+1+14 './' 'Output Path' 'outpath' outpath
-s 1 0 1 0 0 45x1+1+15 ' ' 'String Sel' 'string' string
-E
```

Example 2 specifies an optional mutually exclusive group rather than a required one. The user *may* use one and only one of the items, but they will not be forced to use any of them. On the command line, the user may elect not to specify any of the selections. On the GUI, the user may turn off all the optional boxes of the three selections.

### C.3.26. MutIncl (-B) Line

Collectively, the [-B to -E] definition specifies a mutually inclusive group. A mutually inclusive group allows the user to specify values for all or none of its members. The (-B) line *must* be matched by an End (-E) line, to end the group of selections which will make up the group. Legal group members include InputFiles, OutputFiles, Integers, Floats, Strings, Cycles, Lists, Toggles, Flags, StringLists, Mutually Exclusive groups, and other Mutually Inclusive groups. Group members may be of different types. Group members **MUST** be optional.

#### Use by the GUI

Selections within the group are created as usual; however, the user will be forced to choose either all or none of the selections at any time.

#### UIS File Placement

The -B MutExcl line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. It may also be nested (one level of nesting only) inside a Mutually Exclusive [-C to -E] group, or inside another Mutually Inclusive [-B to -E] group.

#### Use by the CLUI

A mutually inclusive group may be used on the command line to force the user to provide none or all of a group of arguments to the program.

#### Initializing the MI Group:

All selections in a mutually inclusive group **MUST** be specified as optional. The user will have to "turn on" all of the mutually inclusive selections (by highlighting one of the optional boxes) together, or "turn off" all of the mutually inclusive selections together.

Either NONE or ALL of the selections within the group must have their *opt\_sel* fields set to 1 (TRUE). If all the selections have their *opt\_sel* fields all set to 1, the default will be TO USE ALL the members of the group. If all the selections have their *opt\_sel* fields set to 0 (FALSE), the default will be NOT TO USE ANY of the members in the group.

**syntax:**

-B Required

**Required:** The *required* field should be set to 0 on a -B line. Logically, this 0 in the *required* field implies "the user MAY provide ALL or NONE of these selections." Putting a 1 in the *required* field would say, "the user MUST provide ALL of these selections." However, if this is the case, there is no need to make a group of the selections; simply making them all required would achieve the same effect. Thus, it is considered a logic error to specify the *required* field as 1 for a -B line. It is worth noting here that the -B line has the required field in order to maintain consistency with the -C line and to support flexibility and nesting.

**example:**



<input checked="" type="checkbox"/>	Name	John Browne
<input checked="" type="checkbox"/>	Title	Civil Engineer
<input checked="" type="checkbox"/>	S.S.#	532-08-9761

---

**Figure 38:** This mutually inclusive group forces the user to specify all of the information, or none of it.

---

```
-B 0
-i 1 0 1 1 0 25x1+1+4.78261 2 2 525015432 0 0 'Serial Number' 'id' serial
-s 1 0 1 1 0 35x1.05+1+2 'John Doe' 'Name' 'name' name
-s 1 0 1 1 0 35x1.04348+1+3.34783 'Private' 'Rank' 'rank' rank
-E
```

With a mutually inclusive group, if the user gives any of the information, they must give it all. On the command line, an error message will be generated if any of the three is specified without the others. On the graphical user interface, the optional boxes for the three items will turn on and off as a group.

### C.3.27. LooseGroup (-K) Line

#### Explanation:

Collectively, the [-K to -E] definition specifies a loose group. A loose group makes the user specify a value for at least one of its members. The (-B) line *must* be matched by an End (-E) line to end the group of selections that will make up the group. Legal group members include InputFiles, OutputFiles, Integers, Floats, Strings, Cycles, Lists, Toggles, Flags, StringLists, Mutually Exclusive groups, and other Mutually Inclusive groups. Group members may be of different types. Group members **MUST** be optional.

#### Use by the GUI

Selections within the group are created as usual; however, the user will be forced to choose at least one of the selections. The user may choose more than one of the selections if desired.

#### UIS File Placement

The -K LooseGroup line may be specified within a [-S to -E] master form definition, a [-G to -E] guide pane definition, or a [-P to -E] pane definition. It may also be nested (one level of nesting only) inside a Mutually Exclusive [-C to -E] group, or inside a Mutually Inclusive [-B to -E] group.

#### Use by the CLUI

A loose group may be used on the command line to force the user to provide at least one of a group of arguments to the program.

#### Initializing the Loose Group:

All selections in a loose group **MUST** be specified as optional. The user will be able to "turn on" as many of the grouped selections as desired, but they will be forced to have at least one item selected.

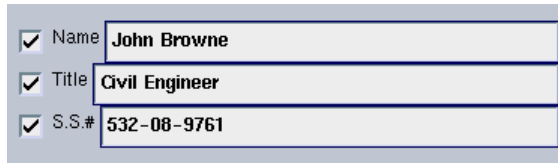
At least one of the selections within the group must have its *opt\_sel* fields set to 1. For each selection having its *opt\_sel* field set to 1 (TRUE), the default will be TO USE that member of the group.

#### syntax:

-K Required

**Required:** The *required* field should be set to 1 on a -K line. Logically, this 1 in the *required* field implies "the user **MUST** provide **AT LEAST ONE** of these selections." Putting a 0 in the *required* field says, "the user **MAY** provide **AT LEAST ONE** of these selections." However, if this is the case, there is no need to make a group of the selections; simply making them all optional would achieve the same effect. Thus, it is considered a logic error to specify the *required* field as 0 for a -K line unless that -K line is starting a loose group that is nested within a mutually inclusive or mutually exclusive group.

example:



<input checked="" type="checkbox"/>	Name	John Browne
<input checked="" type="checkbox"/>	Title	Civil Engineer
<input checked="" type="checkbox"/>	S.S.#	532-08-9761

---

**Figure 39:** This mutually inclusive group forces the user to specify all of the information, or none of it.

---

```
-K 0
-i 1 0 1 1 0 25x1+1+4.78261 2 2 525015432 0 0 'Serial Number' 'id' serial
-s 1 0 1 1 0 35x1.05+1+2 'John Doe' 'Name' 'name' name
-s 1 0 1 1 0 35x1.04348+1+3.34783 'Private' 'Rank' 'rank' rank
-E
```

With a loose group, the user must give at least one piece of the information. On the command line, an error message will be generated if none of the arguments is specified. On the GUI, the user will be forced to leave at least one of the optional boxes for the three items turned on.



Use of Grouping (Single-Level & Nested) in the UIS		
<i>Name</i>	<i>UIS Model</i>	<i>Meaning</i>
Required Mutually Exclusive Group	-C 1 -? ... A -? ... B -? ... C -E	Choose one and only one of A or B or C.
Optional Mutually Exclusive Group	-C 0 -? ... A -? ... B -? ... C -E	Choose one of A or B or C, or choose none.
Mutually Inclusive Group (Always Optional)	-B 0 -? ... A -? ... B -? ... C -E	Choose all of A and B and C, or choose none.
Loose Group (Always Required when Single-Level)	-K 1 -? ... A -? ... B -? ... C -E	Choose one or more of A, B, C, but you must choose at least one.
Required Mutually Exclusive Group with Mutually Inclusive Group Inside	-C 1 -? ... A -B 0 -? ... B -? ... C -E -E	Choose A or choose (B and C).
Required Mutually Exclusive Group with Loose Group Inside	-C 1 -? ... A -K 0 -? ... B -? ... C -E -E	Choose A or choose (B or C or both).
Optional Mutually Exclusive Group with Mutually Inclusive Group Inside	-C 0 -? ... A -B 0 -? ... B -? ... C -E -E	Choose A or choose (B and C) or choose none.

Use of Grouping (Single-Level & Nested) in the UIS		
<i>Name</i>	<i>UIS Model</i>	<i>Meaning</i>
Optional Mutually Exclusive Group with Loose Group Inside	-C 0 -? ... A -K 0 -? ... B -? ... C -E -E	Choose A or choose (B or C or both) or choose none.
Mutually Inclusive Group with Mutually Exclusive Group Inside	-B 0 -? ... A -C 0 -? ... B -? ... C -E -E	Choose (A and (B or C)) or choose none.
Mutually Inclusive Group with Loose Group Inside	-B 0 -? ... A -K 0 -? ... B -? ... C -E -E	Choose (A and (B or C or both)) or choose none.
Loose Group with Mutually Exclusive Group Inside	-K 1 -? ... A -C 0 -? ... B -? ... C -E -E	Choose (A or (B or C)) or choose (A and (B or C)).
Loose Group with Mutually Inclusive Group Inside	-K 1 -? ... A -B 0 -? ... B -? ... C -E -E	Choose (A or (B and C)), or choose (A and (B and C)).

## D. GUI Details

### D.1. Geometry Of The Master Form, Guide Pane, And Pane

Geometry is a key factor in determining how the GUI is presented to the user. The geometry strings for master forms, guide panes, and panes specify the *minimum* size of the backplane. Any master form, guide pane, or

pane will always be large enough to encompass all of the GUI items in it, regardless of the geometry on the (-F), (-M), or (-P) lines. Thus, selections with geometry strings indicating *x,y* positions far away from the upper-left corner will cause the master form, guide pane, or pane to become larger. If you would like a master form, guide pane, or pane to be bigger than it needs to be to contain the GUI items within it, use the width and height specifications to do so.

## D.2. Understanding The Master Form

The [-S to -E] Master Form description indicates that a separate master form is desired. The characteristic buttons on the *Master* form are the *subform buttons*. These buttons, when clicked on by the user, will automatically bring up the specified *subforms*. The use of a master form enables you to divide up the input coming from the user into major activities so that one subform may deal exclusively with one aspect of the application program, while a different subform may deal exclusively with another.

### D.2.1. Contents of the Master Form

At least one (-d) subform button must be specified to make the master form legal. Subform buttons will each bring up a different subform and re-direct user attention to the pane(s) on that subform. Action buttons will return control to the *xv* routine, so that the *xv* routine may take the desired action. Every master form must have a *Help* button that should provide an introduction to and overview of the entire application program. A workspace may be useful on the master form: it provides a general-use VisiQuest Manager Object of the specified geometry in which you may display graphics or images, or which you may use as a backplane for creating other visual objects. There should always be a *Quit* button on the master so that the user may exit the program. You may also include any of the general GUI selections in addition to toggles and mutually exclusive groups. However, it is advisable *not* to clutter your master form with selections if those selections can be intuitively located on a guidepane or pane.

### D.2.2. Omission of the Master Form

If the [-S to -E] master form description is omitted, the (-F) StartForm UIS line can be followed immediately by a *single* [-M to -E] subform definition. Thus, this constitutes the description of a single subform. The key to deciding whether or not to use a master form is whether or not you need more than one subform. If only one subform is needed, a master form is unnecessary.

### D.2.3. Mutual Exclusion of Subforms

If a master form is to be used, the (-S) line indicates whether or not the subforms are to be mutually exclusive. If the third field on the (-S) line is TRUE (1), this indicates that there *is* to be mutual exclusion. When the forms are displayed using *preview*, you will notice that only one subform may be brought up at a time. Alternately, the mutual exclusion field on the (-S) line may be set to FALSE (0). When subforms are not mutually exclusive, all the subforms may be brought up at the same time.

### D.2.4. Setting A Default Subform

Usually, when a master form is used, there is no default subform. The master form is mapped by itself, and no subforms appear until the user clicks on a subform button. To use this presentation, you set *all* "selected" fields on the (-d) subform button lines within the master form [-S to -E] definition to be 0 (FALSE).

However, in some rare cases it might be desirable to have a particular subform mapped immediately, in addition to the master form. To make a certain subform map at the same time as the master form, set the "selected" field on its (-d) subform button line to 1 (TRUE).

### D.3. Understanding The Subform

The *subform* is generally divided into two parts: the *guide pane*, and the *panes* themselves. The guide pane is a section of the subform that, according to the geometry string specified on the (-G) StartGuide line, may encompass the far left side of the subform, the far right side of the subform, the top or the bottom of the subform. Panes hold the bulk of the selections for user I/O to the xvroutine.

### D.4. Understanding The Guide Pane

The [-G to -E] description indicates that a guide pane is desired. The characteristic buttons on the guide pane are the *guide buttons*. These buttons, when clicked on by the user, will automatically bring up the specified panes. The use of a guide pane enables you to divide up the input coming from the user into divisions. For example, one pane on a subform might deal exclusively with input, while the other pane on the subform could deal exclusively with output.

#### D.4.1. Contents of the Guide Pane

At least two (-g) guide buttons must be specified to make the guide pane legal. Guide buttons will each bring up a different pane and re-direct user attention to the selections on that pane. Action buttons will return control to the xvroutine so that the xvroutine may take the desired action. Every guidepane must have a *Help* button that should provide online help to describe the functionality of the subform as a whole, and gives a brief description of the use of each pane. A Workspace may be useful on the guidepane, especially if all the panes on the subform deal with the same display of image or graphics. There should always be a *Close* (Quit) button on the guidepane so that the user may close the subform. You may also include any general selections on the guidepane; however, it is better *not* to clutter your guidepane with selections, especially if those selections can be more intuitively located on one of the panes of the subform.

#### D.4.2. Omission of the Guide Pane

If the [-G to -E] guide pane description is omitted, the (-M) StartSubForm UIS line can be followed immediately by a *single* [-P to -E] subform definition. This constitutes the description of a single pane. The key to deciding whether or not to use a guide pane is whether or not you need more than one pane. If only one pane is needed, a guide pane is unnecessary.

#### D.4.3. Setting The Default Pane

Suppose that in our UIS, we have a subform with a guide pane containing several guide buttons, the first of which is labeled, *Input*. We would like this subform to come up with the *Input* guide button selected, and the corresponding "Input" pane mapped. This is done by setting the "selected" field to 1 (TRUE) on the (-g) guidebutton line that defines the *Input* guide button. Usually, when a guide pane is used, a particular pane is chosen as the default.

However, it might be preferable to have our hypothetical subform come up without any guide button selected and with no pane mapped. To do this, we would set *all* "selected" fields on the (-g) lines within the guide pane [-G to -E] definition to be 0 (FALSE).

## **D.5. Understanding The Pane**

The [-P to -E] description is what defines a pane on a subform. The characteristic items on a pane are *selections*. Most of the actual user I/O takes place on panes.

### **D.5.1. Contents of the Pane**

Most of the GUI selections are typically located on the appropriate pane. Selections include InputFiles, Output Files, Integers, Floats, and so on. Toggles and mutually inclusive/exclusive groups are also viewed as selections. While selections are legal on guidepanes and master forms, selections should always be relegated to panes whenever possible in order to minimize clutter on the controlling regions of the GUI.

### **D.5.2. Use of Action Buttons**

The pane action button is viewed as one of the most important items on a pane. The correct use of the pane action button is crucial when designing a GUI, since it is the user's click on a pane action button that returns control to the application program. The user might enter integers, floats, and strings, set toggles and logicals, etc., all day long if desired, but control *will not be returned to the application until the user clicks on a pane action button* (with one exception to this, discussed in the next section).

For this reason, it is always best if you decide beforehand what operations will be available on a particular pane, and for each operation, which values will need to be set. A GUI for an xvroutine will include a set of necessary selections in which the user may enter appropriate values, followed by a pane action button labeled with the name of the operation to be performed using those values. When the user clicks on the pane action button, the GUI toolkit will return control to the application, so that the application may gather up the pertinent values from the GUI and take the desired action accordingly.

### **D.5.3. Use of Live Selections**

The exception to the rule detailed above is when selections are made *live*. An inputfile, outputfile, integer, float, string, logical, or toggle selection may be made live by setting the appropriate field on the UIS line that describes it. When a selection is live, control will return to the application program whenever the user presses <Return> within the parameter box of that selection. Thus, if the only piece of information is needed before the application program can take some particular action, a live selection may fit the bill better than a non-live selection followed by a pane action button. If more than one piece of information is needed, however, the pane action button is still the method of choice for returning control to the application program.

# Table of Contents

A. Introduction To The User Interface Specification (UIS) File . . . . .	10-1
A.1. UIS File Creation / Modification / Editing . . . . .	10-1
B. Most Common UIS Line Fields . . . . .	10-2
B.1. Activate . . . . .	10-2
B.2. Selected . . . . .	10-2
B.3. Optional . . . . .	10-3
B.4. Opt_Sel (Option Selected) . . . . .	10-3
B.5. Live . . . . .	10-4
B.6. Geometry String . . . . .	10-5
B.7. Title Offset . . . . .	10-5
B.8. Default . . . . .	10-5
B.9. Title . . . . .	10-6
B.10. Description . . . . .	10-6
B.11. Variable . . . . .	10-6
C. UIS Line Syntax . . . . .	10-6
C.1. UIS Lines That Structure The GUI and Required UIS Lines . . . . .	10-6
C.1.1. StartForm (-F) Line . . . . .	10-6
C.1.2. StartSubform (-M) Line . . . . .	10-8
C.1.3. StartPane (-P) Line . . . . .	10-9
C.1.4. StartMaster (-S) Line . . . . .	10-10
C.1.5. StartGuide (-G) Line . . . . .	10-11
C.1.6. End (-E) Line . . . . .	10-12
C.2. UIS Lines That Are Positionally Restricted . . . . .	10-12
C.2.1. SubformButton (-d) Line . . . . .	10-13
C.2.2. MasterAction (-n) Line . . . . .	10-14
C.2.3. GuideButton (-g) Line . . . . .	10-16
C.2.4. SubformAction (-m) Line . . . . .	10-17
C.3. UIS Lines Representing General Purpose Items . . . . .	10-19
C.3.1. Help (-H) Line . . . . .	10-19
C.3.2. Quit (-Q) Line . . . . .	10-21
C.3.3. InputFile (-I) Line . . . . .	10-22
C.3.4. OutputFile (-O) Line . . . . .	10-24
C.3.5. Integer (-i) Line . . . . .	10-27
C.3.6. Float (-f) Line . . . . .	10-29
C.3.7. Double (-h) Line . . . . .	10-32
C.3.8. String (-s) Line . . . . .	10-35
C.3.9. Flag (-t) Line . . . . .	10-37
C.3.10. Logical (-l) Line . . . . .	10-39
C.3.11. Cycle (-c) Line . . . . .	10-41
C.3.12. List (-x) Line . . . . .	10-45
C.3.13. DisplayList (-z) Line . . . . .	10-49
C.3.14. StringList (-y) Line . . . . .	10-52
C.3.15. Blank (-b) Line . . . . .	10-54
C.3.16. PaneAction (-a) Line . . . . .	10-55
C.3.17. Workspace (-w) Line . . . . .	10-58
C.3.18. Routine (-R) Line . . . . .	10-59
C.3.19. StartSubMenu (-D) Line . . . . .	10-62

C.3.20. IncludeSubform (-k) Line . . . . .	.10-63
C.3.21. IncludePane (-p) Line . . . . .	.10-64
C.3.22. Stdin (-e) Line . . . . .	.10-64
C.3.23. Stdout (-o) Line . . . . .	.10-66
C.3.24. Toggle (-T) Line . . . . .	.10-67
C.3.24.1. Notes on Toggle Use . . . . .	.10-70
C.3.24.2. Toggle of Flags . . . . .	.10-71
C.3.24.3. Toggle of Logicals . . . . .	.10-72
C.3.24.4. Toggle of Integers . . . . .	.10-73
C.3.24.5. Toggles of Floats & Doubles . . . . .	.10-74
C.3.24.6. Toggle of Strings . . . . .	.10-74
C.3.24.7. Toggles of Input & Output Files . . . . .	.10-75
C.3.25. MutExcl (-C) Line . . . . .	.10-76
C.3.26. MutIncl (-B) Line . . . . .	.10-78
C.3.27. LooseGroup (-K) Line . . . . .	.10-80
D. GUI Details . . . . .	.10-83
D.1. Geometry Of The Master Form, Guide Pane, And Pane . . . . .	.10-83
D.2. Understanding The Master Form . . . . .	.10-84
D.2.1. Contents of the Master Form . . . . .	.10-84
D.2.2. Omission of the Master Form . . . . .	.10-84
D.2.3. Mutual Exclusion of Subforms . . . . .	.10-84
D.2.4. Setting A Default Subform . . . . .	.10-84
D.3. Understanding The Subform . . . . .	.10-85
D.4. Understanding The Guide Pane . . . . .	.10-85
D.4.1. Contents of the Guide Pane . . . . .	.10-85
D.4.2. Omission of the Guide Pane . . . . .	.10-85
D.4.3. Setting The Default Pane . . . . .	.10-85
D.5. Understanding The Pane . . . . .	.10-86
D.5.1. Contents of the Pane . . . . .	.10-86
D.5.2. Use of Action Buttons . . . . .	.10-86
D.5.3. Use of Live Selections . . . . .	.10-86