The following paper was originally published in the
Proceedings of the Fourth USENIX Tcl/Tk Workshop
Monterey, CA, July 10-13, 1996

Tcl/Tk in the Development of User-Extensible Graphical User Interfaces

John M. Skinner, Richard S. LaBarca, and Robert M. Sweet
Biology Department, Brookhaven National Laboratory, Upton, NY
Carnegie Mellon University, Pittsburgh, PA

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Tcl/Tk in the Development of User-Extensible Graphical User Interfaces

John M. Skinner[*], Richard S. LaBarca[‡], Robert M. Sweet[*]
[*]*Biology Department, Brookhaven National Laboratory,*
Upton, NY 11973 skinner@bnl.gov, sweet@bnl.gov
[‡]*Carnegie Mellon University, Pittsburgh, PA 15213 rlabarca@cmu.edu*

Abstract

This paper describes the use of Tcl/Tk in the development of *user-extensible* Graphical User Interfaces. We describe a GUI that provides two facilities, employing Tcl/Tk, that allow its users to extend the basic GUI at runtime. The first facility is a built-in mini Tcl/Tk GUI builder that gives users the ability to generate new screens that are automatically incorporated into the existing GUI. This is accomplished at runtime and does not require the user to write any code. A second feature allows for the incorporation, control, and access of data from, pre-existing Tcl/Tk screens without modification to the original source code of those screens. Program extensions accomplished with these facilities are integrated seamlessly into the original program, and are available immediately, as well as in subsequent executions of the program. In addition to the demonstration of a specific application of these methods, we will show that the techniques are transferable to other applications.

Background

The main subject of this paper is a GUI that is used to run instrument-control programs [1] at Brookhaven National Laboratory's (BNL) National Synchrotron Light Source (NSLS). The NSLS is a world-class research facility used by over 2500 researchers each year to perform experiments in physics, chemistry, biology, materials science, and various other technologies. The instrument-control programs originally had text-based, command-line driven interfaces. Since there are only a handful of facilities such as ours in the world, and many scientists are able to use the available research time, users are under significant pressure to make the best use of the limited time they are given. The result is that they are often sleep-deprived, stressed, and prone to making errors. In response to this situation, we have invested a significant effort in the development of user-friendly software for data acquisition and analysis.

The main GUI for our instrument control programs was written in C with the aid of the Builder Xccessory GUI builder [2]. Developing the GUI presented an interesting challenge in that the underlying programs provide a C-like macro language that allow users to define new functions at runtime. Users frequently develop customized macros for these programs at numerous experimental stations. A significant percentage of a user's interaction with these programs is through these new macros, which are typically different at each station using the software. Since it was impossible to anticipate what interfaces for future program extensions should look like, or to construct interfaces for all of the important macros already written, we realized that users would need to be able to extend the interface themselves. Clearly, we could not assume that users of the program would have the programming abilities or the time necessary to modify the code of the basic GUI. In addition, since the macros can be defined and incorporated at runtime, we felt that the interface extensions should also be able to be developed and applied without exiting the program.

It seemed clear that if an end-user were to be able to extend the GUI without writing any code, that some sort of "GUI builder" would need to be provided. Although Tcl/Tk builders such as SpecTcl [3] are excellent tools for program developers, someone using these builders would still need to have some knowledge of Tcl/Tk to write the callback procedures. In addition, we felt that the comprehensiveness of a tool such as SpecTcl required some initial investment of time on the part of the user in order to learn how to use the builder properly. We therefore decided to write our own built-in, mini GUI builder. This would enable us to ensure that the code generated by this tool had the capability for two-way communications with the existing program without requiring modification. We also could write our builder so that no coding was necessary to add new screens.

Design Decisions

Our main objective in designing our builder was ease-of-use. We wanted the users to be able to

generate useful interfaces without having to spend time thinking about how to use the builder. This led to trade-offs. Clearly, the more features we included, the more complicated the builder would be to use and, we felt, the less people would be willing to use it. We decided to limit ourselves to the generation of labels, entries, and pushbuttons. Although the resulting screens would contain only this limited set of GUI components, these would be sufficient to produce useful interfaces that provide distinct advantages over command-line execution. When using one of the generated screens, the user will immediately know information about the program command, such as its name, the number of arguments, something about those arguments from their labels, and reasonable default values. In addition, the screens can display important results or status during and after execution of the command, and the user may easily re-execute with minimal or no editing.

Although other components such as toggle buttons and radioboxes would enhance the appearance of the resulting screens, it was not felt that the potential for enhancement outweighed the corresponding increase in the complexity of using the system. We also bore in mind that we planned to supply a facility, described later, that would accommodate those wishing to invest the time to create more elaborate interfaces.

Generating New Screens

Figures 1 through 3 illustrate the steps involved in generating a new screen with our "GuiBuilder". Pushing the "GuiBuilder" button in the "Beamline Macros" menu will cause a form to pop up which will request the name of the command to call, the number of inputs, and the number of outputs (Figure 1). Based on this information, a final form will be displayed that will request label names and default values (Figure 2). Applying this information will result in the generated screen being displayed (Figure 3). It will contain the requested input and output fields, default values, and "Go" and "Close" buttons. Pushing the "Go" button will execute the command with the supplied input. Results can be displayed in the output fields. In addition, a button will be added automatically to the Beamline Macros menu (Figure 3) so that this screen may be called up again. The fact that this command will now be accessible from a pulldown menu is in itself a significant advantage. The menu of extensions serves to remind the user of the most important commands in the program and only requires that they recognize a command that they need to run, as opposed to having to remember its

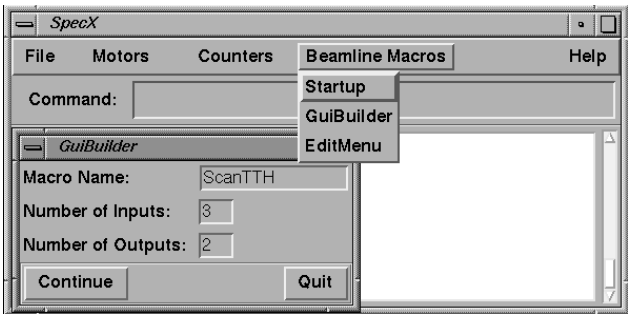

Figure 1. GuiBuilder: step 1

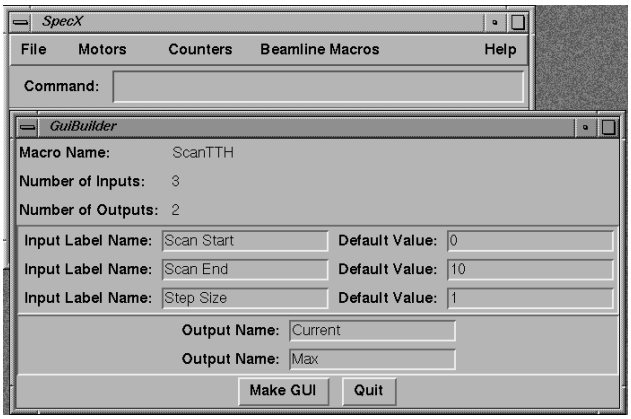

Figure 2. GuiBuilder: step 2

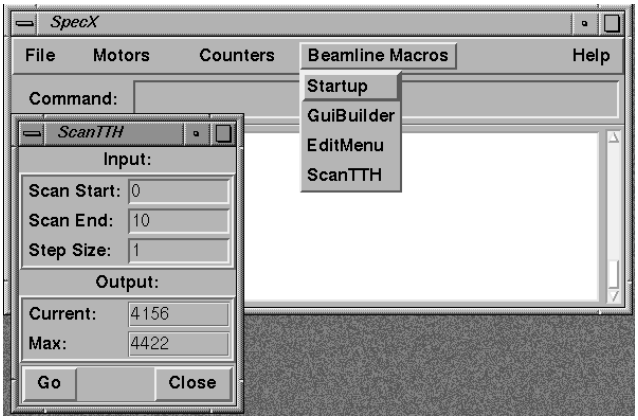

Figure 3. GuiBuilder: result

name, or be forced to query the program for all available commands.

The following is a listing of the macro that the new screen was generated for:

```
def ScanTTH '
#$1=start,$2=end,$3=step size
maxint = 0
for (cp=$1; cp<=$2; cp+=$3) {
```

```
  mva tth cp
# next line puts intensity in val
  read_intensity

# the next line causes the
# intensity to appear in the
# "Current" field of the generated
# screen
  sendgui "ScanTTH" "Current" val

  if (val > maxint) {
    maxint = val

# display the maximum intensity in
# the GUI
    sendgui "ScanTTH" "Max" maxint
  }
}
'
```

The GuiBuilder produces different results based on the user's entering a zero in the number of input and/or output fields requested when interacting with the first form. If only output fields are requested, then the resulting screen will be configured as an output-only status screen. These screens will not contain input fields, default values, or a go button. They are intended to make important status information easily visible to the user.

If a user specifies zero inputs and zero outputs in the first form, then no second form will be displayed, and the result will be a command button added to the "Beamline Macros" menu.

Incorporating User-Written Screens

To accommodate experienced Tcl/Tk programmers who wish to develop more complex screens and integrate them into the GUI, we have provided a second extension facility, called "EditMenu". This aspect of the program is used to incorporate user-written Tcl/Tk screens that have no special provisions for communications with other processes. After being added to the existing GUI with EditMenu, the Tcl/Tk variable values of the newly incorporated screens may be accessed from the instrument-control program by calling "guival", and set with "sendgui". More detailed control of a Tcl/Tk screen can be accomplished by using the "gui_exec" call from the underlying program. This function allows one to send Tcl/Tk code segments to a screen. This can be used for such actions as modifying values, changing colors, adding new GUI components, and binding new actions. One may actually use "gui_exec" to perform all of the functions that are accomplished with "sendgui" and "guival". When a screen is added with EditMenu, it will then become an item in the Beamline Macros menu. We also supply a "popgui" command that will provide us with the same communication mechanisms as EditMenu, but will not result in an addition to the Beamline Macros menu.

The access and control of the screens incorporated with EditMenu or "popgui" is accomplished without editing the Tcl/Tk source. Our method does not require any particular X server authorization scheme, and is done in a way that does not compromise the security of the system. One may also edit the Tcl/Tk scripts so that they issue commands to the program from which they are controlled. This is done by calling "send_command" from inside the Tcl script. "send_command" will send any string, typically a command, to the underlying program. It can be called by any script that has been incorporated with EditMenu or popgui. It is also possible to use "gui_exec" from inside a macro to bind a send_command call to a widget instead of editing the script. Figure 4 details the profiles and usage of the calls that interface the underlying program with the Tcl/Tk scripts.

The “SpeedControl” window in figure 5 is a Tcl/Tk GUI that was constructed with SUN's SpecTcl GUI Builder. This was done independently of the existing GUI. That is, there is no special code included to facilitate communications with other procedures or

Tcl/Tk GUI Communications Command Summary

sendgui - Send output to a Tcl/Tk GUI.
usage: sendgui <screen name><variable name><value>
example: sendgui "speed control" "speed" speed_variable

guival - Retrieve a value from a Tcl/Tk GUI.
usage: guival <screen name> <variable name>
the_value = input() #need to read the result
example: guival "Controlit" "var1"
var1_value = input()

gui_exec - Send Tcl/Tk code to a screen to be executed.
usage: gui_exec "screen name" code
example: gui_exec "SpeedCtl" "bind .quit <Button-1> exit"

popgui - Pop up a user-defined GUI.
usage: popgui <screen name>
example: popgui "temperatures"

send_command - Send a command from Tcl/Tk to the underlying program.
usage: send_command <some_command_string>
example: button .go -text "Go" -command\
{send_command drive_motor}

Figure 4. Communications commands summary table

processes and the buttons have no command bindings. The bindings are added by the macro listed below for the purpose of demonstrating the gui_exec call.

```
def wire_speed_buttons '
  code = sprintf("bind .go
<Button-1> {send_command
RunMotor}")
  gui_exec "SpeedControl" code
  code = sprintf("bind .quit
         <Button-1> exit")
  gui_exec "SpeedControl" code
'
```

By using the EditMenu facility (Figure 5), the screen is merged into the existing GUI, and will be used to control the speed of some imaginary device. For demonstration purposes, this device will be another Tcl/Tk screen containing some scale and entry

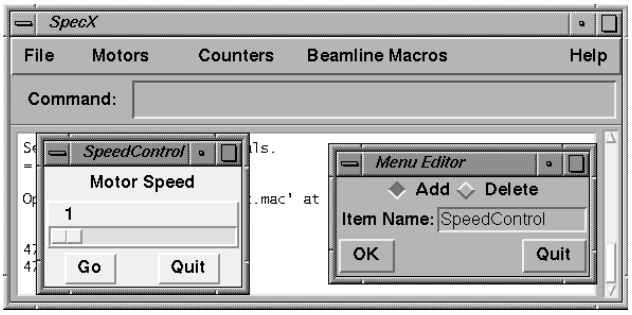

Figure 5. Extending the program with EditMenu

widgets. To demonstrate how to implement communications between the added Tcl/Tk screens and the underlying program, the “SpeedControl” screen (Figure 5) will have its scale widget variable value retrieved and used to control the update rate of the second screen. Listed below is the program macro that communicates with these two interfaces:

```
def RunMotor '
for (i=0; i<50; i++) {
#the following 6 lines are used to
#drive an imaginary device, which
#in this case is another Tcl/Tk
#screen
  var1 = (i*20)%100
  var2 = 100 - (i*20)%100
  var3 = (i*10)%100
  sendgui "RunningMotor" "var1"
var1
  sendgui "RunningMotor" "var2"
var2
  sendgui "RunningMotor" "var3"
var3
  guival "SpeedControl" "speed"
  sleep_time = input()
  sleep_time = 1 - sleep_time/10
  sleep(sleep_time);
}
'
```

In this example, we did not edit any of the code generated by SpecTcl. Our only requirement is that we rename the generated Tcl file to be the project name used by SpecTcl.

Below is a macro code segment that utilizes "popgui" and "gui_exec":

```
def get_motor_code '
{
local i
  popgui "MotorSelect"
  for (i=0;i<MOTORS;i++) {
    code =
sprintf(".listframe.motors
insert end %s",motor_mne(i))
    gui_exec "MotorSelect" code
  }
}
'
```

The “Motor Select” screen that it communicates with is seen in Figure 6. This is a realistic example. A macro performs some operation on some motor. Instead of prompting the users with a text string and forcing them to respond at the command line it would be better to present the user with a list of all motors, and allow them to use their mouse to select the desired one. Since the list is really used more like a dialog as opposed to an interface for execution of a function, we do not need to see it under the Beamline Macros menu. Instead we will call "popgui" from the macro to display the listbox when needed.

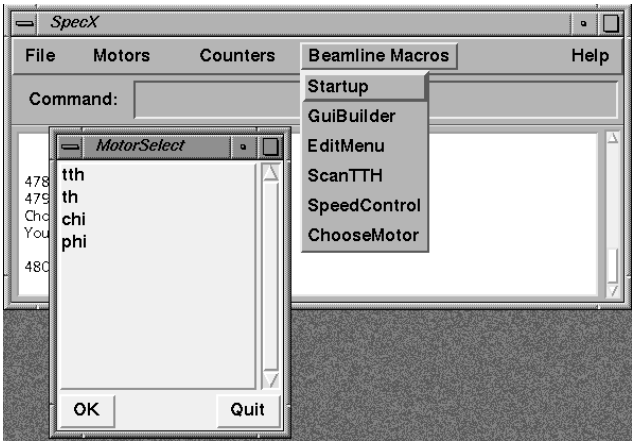

Figure 6. MotorSelect listbox

By selecting "delete" in the EditMenu screen, one may remove any of the program extensions that were added with GuiBuilder or EditMenu.

Implementation Details

Figure 7 illustrates the architecture of the system. The basic GUI, which is a C program, forks off the instrument-control program, and communicates through the standard I/O channels that have been redirected to pipes connecting the processes. An extra pipe is added that is used for the passing of information from the underlying program to the GUI. This information is parsed by the GUI and used to update interface components. We have used this communications scheme to convert other tty-based programs to be GUI-driven and have described the details in [4].

"GuiBuilder" and "EditMenu" are executed with UNIX "system" calls. The Tcl/Tk screen extensions are indirectly executed with the Unix "popen" call, and the returned file descriptors are saved in an array of records that contain screen names and their corresponding file descriptors. Since the basic GUI's standard output has been redirected to the underlying program, it follows that any program spawned from the GUI will automatically have its standard output directed to the child program as well. Therefore, communications from the Tcl/Tk screens to the underlying program are accomplished by simply writing and flushing stdout.

The flow of information from the instrument-control program, through the main GUI, to the Tcl/Tk screen is a bit more complicated. As mentioned above, the Tcl/Tk scripts are popen'd "indirectly". We have written a Tcl/Tk program that we have named "tcl_bridge" that allows us to access and control other Tcl/Tk programs. tcl_bridge performs a *fileevent* call on its standard input and registers a procedure to parse the standard input and do what we request based on commands that we have defined. The last line of tcl_bridge calls Tcl's "source" command with the name of the script that we originally wanted to execute. For example, the result of a call such as "tcl_bridge mytclprog", is whatever "mytclprog" did before plus its monitoring of its standard input for requests to update variables, print variable values, or execute new code segments that we send it. More will be said about tcl_bridge later.

The two-way communications between the child program and the Tcl/Tk screens are accomplished as

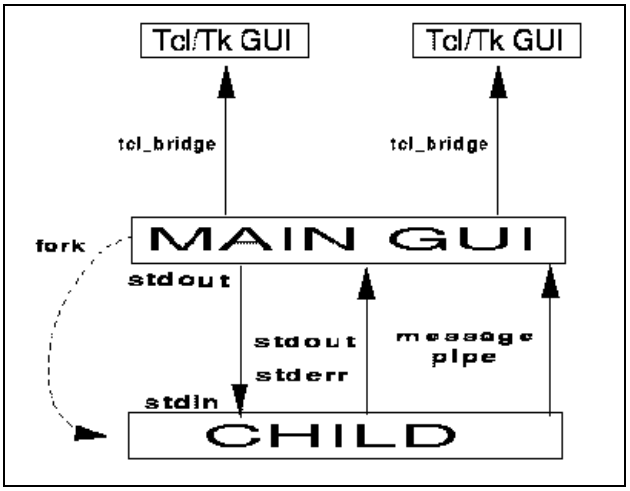

Figure 7. System architecture

follows: To retrieve a value from a Tcl/Tk screen variable, the macro makes a call to "guival" with the name of the screen and the variable value from that screen that we wish to retrieve. "guival" is a macro that will put this information into a format that is recognized by the main GUI as a request to be forwarded to a Tcl/Tk script. The main GUI will traverse the list of active Tcl/Tk scripts and forward the message to the appropriate one. The script will then parse the message and take the desired action, in this case, writing out the value of the variable that was requested. The underlying program can then retrieve this value with a read statement immediately following the guival call. The sendgui call behaves in a similar manner but results in a different request being sent to the Tcl/Tk script. In this case the variable value will be updated instead of output. The "gui_exec" call behaves as the previous two but results in its second argument being executed with an "eval" call in the target Tcl/Tk script.

Saving Extensions

Currently, all GUI extension screens are saved by saving their corresponding Tcl/Tk scripts in the directory in which the main GUI is running. In addition, a file listing these extensions is created, maintained, and read by the main GUI so that the extensions will be present upon subsequent executions. It would be easy to change this scheme but it has worked well for us so far due to the fact that our users typically run these instrument-control programs from the same directory each time. Another approach might be to have an environment variable point to a list of directories in which extensions and their databases may be stored.

Tclbridge vs. Send

When we considered how we might implement the communications between the Tcl/Tk applets and the rest of the system we initially thought that Tk's "send" command would be the answer. However, using the send command required us to switch all workstations using our software to xauth-style server authorization, or to compile Tk with it's security flags off so that it would allow the use of "send" in an insecure manner. We consulted with a number of colleagues on the possibility of converting from xhost to xauth-style server authorization. We found that most people had heard of xauth, few knew how to set it up, and no one was eager to change from what they had been using. Based on this we felt that requiring xauth's use would be a major obstacle in the program's use at research stations that we did not administer. Furthermore, the idea of having Tk compiled with its security flag off on government computers controlling, in some cases, millions of dollars worth of instrumentation was clearly not acceptable. Therefore we decided to develop our own method of controlling and accessing data from Tcl/Tk programs. The resulting tcl_bridge has provided us with a reliable and secure solution without dictating how we permit access to our servers.

A Generic Version

In the application that was just described, we developed macros for the instrument control programs that allow for the easy access and control of Tcl/Tk screens. We have also developed a package of C routines that accomplish the same goals. These routines are used in concert with a "boiled-down" generic version of our GUI that, for example, can be used to rapidly develop an extensible GUI for an existing command-line driven program. Typing "xrun <some_program>" will result in the program specified being controlled by the XRun GUI. Commands typed into the command window will be communicated to the underlying program with the output from that program being displayed in the scrolled window of XRun. In addition, the same facilities for communication with Tcl/Tk in the instrument-control program are available in this version. For example, one may call the "sendgui" C routine from the child program to update screens generated by GuiBuilder or incorporated with EditMenu. Figure 8 displays XRun controlling an unmodified version of gnuplot [5]. As a simple illustration, a trivial screen calling gnuplot's "plot" command was generated with GuiBuilder and can be used to modify the graph displayed by gnuplot. Using XRun, converting a command-line driven C program to

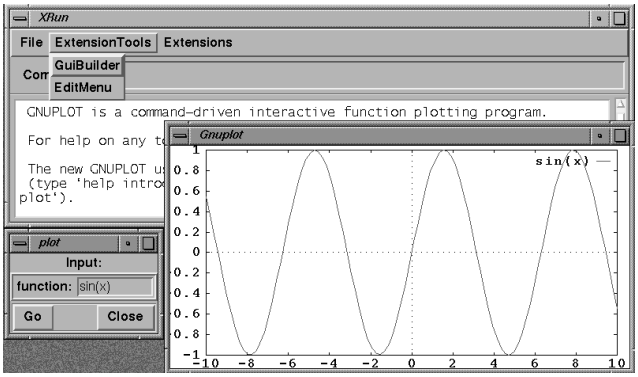

Figure 8. XRun controlling Gnuplot

one that is controlled by a user-extensible GUI should only require minor modifications to the original C program source. This is the same approach that we have used in developing the instrument-control GUI described previously as well as other programs in use at our facility. We have also used a modified version of the C code that provides the communications with the Tcl/Tk screens via tcl_bridge, to develop a Tcl/Tk front end for a robot control program developed at Carnegie Mellon University.

Tcl_bridge and Perl

We have also found tcl_bridge to be very useful as a means of interfacing Perl with Tcl/Tk. We liked Perl for executing and interpreting the output of our data-analysis programs and generating reports and command files based on the output. At some points of the execution of these Perl scripts, we wanted to display information and solicit user input in a graphical fashion. It was trivial for us to write Perl routines that could use tcl_bridge to run, access, and control Tcl/Tk scripts. We looked briefly at the possibility of using TkPerl or PerlTk for these tasks, but found that we could accomplish everything that we wanted by using tcl_bridge to provide the interface between Perl and Tcl/Tk. We liked this solution because it was not dependent on any particular version of Perl or Tcl/Tk. Also, the Perl and Tcl/Tk scripts were separate and developed independently of each other, and we did not need to learn a new syntax which looked like a hybrid of the two languages as appeared to be the case with TkPerl and PerlTk.

Conclusions and Experiences

This work was initiated to solve the specific problem of providing a user-friendly front end that could be easily extended by others, irrespective of their

programming abilities, and without the need for its author to intervene. We are confident that this effort was successful owing to the fact that we have witnessed users of the program interacting with screens that they have created themselves. Although the codes were at first developed specifically for our instrument control application, we have seen that, particularly in the case of the tcl_bridge and associated libraries, their usefulness is quite general. Part of the reason for this is that in many environments few software systems are monolithic. When developing a software extension it is rarely accomplished by modifying the source of one large program. The solution is often to integrate a new component into the system, thus preserving the reliability of the original software. The first choice for a language to implement a new component is often a scripting language such as Tcl/Tk, Expectk, or Perl. This component approach requires an easy way to interface the different processes written in various languages and our method has worked very well for us in this respect in a number of varied applications.

Availability

Documentation and codes described in this paper can be found on the World Wide Web at http://lsx12e.nsls.bnl.gov/x12c/tcltools.html. Interested parties may freely copy, use, and modify the code. We ask that those who make use of any of this work please provide us with feedback.

Acknowledgments

We acknowledge the important help and suggestions that have resulted from our interactions with Kate Feng-Berman and Gerry Swislow. They are the respective authors of the instrument control programs ACE [6,7] and SPEC [8] that the GUI described in this paper was written for. This work was supported by the U.S. Department of Energy under contract DE-AC02-76CH00016 with Associated Universities, Inc. and by a biological instrumentation grant from the National Science Foundation.

References

1. S. Kate Feng, D. Peter Siddons, Lonny Berman, "NSLS Beam Line Data Acquisition and Analysis Computer System". Nuclear Instruments and Methods in Physics Research A347(1994)603-606.

2. Integrated Computer Solutions Inc., "The Builder Xcessory Reference", Cambridge, Ma., 1991

3. Uhler, Stephen, A Graphical User Interface Builder for Tk, Tcl95 Workshop Proceedings, Toronto, Canada 1995

4. J.M. Skinner, R.S. LaBarca, R.M. Sweet, A User-Extensible Graphical User Interface for control of either instrument-control program, ACE or spec. Nuclear Instruments and Methods in Physics Research, Submitted April, 1996

5. T. Williams, C. Kelley, GNUPLOT Interactive Plotting Program, Copyright (C) 1986 - 1993

6. J.M. Skinner, J.W. Pflugrath, and R.M. Sweet, SHARE 80 Proceedings, Session I224, (1993) San Francisco, Ca.

7. S. Kate Feng, D. Peter Siddons, John Skinner, Robert M. Sweet, NSLS Beamline Control and Data Acquisition Computer-System Upgrade", Proceedings of ICALEPCS95 conference (1995), Chicago, Illinois.

8. **spec** is a trademark of Certified Scientific Software, PO Box 390640, Cambridge, MA 02139-0007, tel:617-576-1610, fax:617-497-4242, http://www.ceritf.com, info@certif.com