Tcl/Tk Tools for EPICS Control Systems.

R. Fox
National Superconducting Cyclotron Laboratory[1]
Michigan State University
East Lansing, MI 48824-1321

***Abstract*—The Experimental Physics and Industrial Control System (EPICS) is a control system in wide use in the control systems of accelerator laboratories across the world as well as in large-scale particle physics experiments. This paper will describe a Tcl package that provides access to EPICS control systems and a set of widgets that allow user interfaces to EPICS systems to be easily constructed. The extension will be compared and contrasted with the et_wish EPICS aware extended wish, and a justification for choosing to write a new extension will be given.**

I. Introduction and Outline

The Experimental Physics and Industrial Control System (EPICS)[1] is a distributed control system that is heavily used in nuclear and high energy physics experiments and accelerators. Los Alamos National Laboratories and Argonne National Laboratories originally developed EPICS and the EPICS organization supports further development and international use.

The National Superconducting Cyclotron Laboratory at Michigan State University is the leading accelerator laboratory in unstable heavy ion research in the United States and one of the leaders in the world. Our accelerator and beam-line controls are built around the EPICS control system. Several facility experimental devices, such as the S800 spectrometer [2], also feature EPICS in their slow control paths.

Recently several factors pushed me to investigate the use of Tcl to produce applications that interface with the NSCL EPICS system:

1. In my role as the software lead for the data acquisition system, I was getting an increasing number of requests to interface the data taking system in a read-only manner with data that could be obtained from the EPICS system. These requests ran the gamut from on-line monitoring of EPICS system channels during experimental data taking to inclusion of time varying control system parameters in the main event flow.
2. The Gas Stopping Cell[3], an experimental system, which performs high precision mass and half-life measurements on unstable nuclei could be run more efficiently and more effectively if it had available to it a system that sequenced several data taking runs while making new controls settings for the beam-line and gas cell EPICS parameters between runs.
3. The accelerator controls development group at the NSCL, after several years of "Windows only" console subsystems was looking for ways to create portable console applications.
4. The accelerator operators were looking for ways to get faster turn-around for desired changes in console applications and new console application development.

The remainder of this paper is organized as follows:

- Section II will provide a brief structural summary of EPICS and how EPICS control systems are typically implemented in the field.
- Section III will describe past work on interfacing Tcl/Tk to EPICS, why we did not choose to use prior art and what our requirements and desirements for an EPICS interface package were. A discussion of how we would structure our software is given as well.
- Section IV breaks in to three sub-sections. The first describes the low-level compiled extension that provides Tcl/Tk applications with access to EPICS control system channels. The second describes a set of Tk mega widgets that can be used to meet some control system needs irrespective of the underlying control system. The third describes a set of "EPICS aware" mega widgets that can be used to quickly build control system applications in Tcl/Tk.
- Section V will describe the status of the software, its level of adoption amongst the various development groups at the NSCL, and availability for outside use.

[1] The National Science Foundation under grant number PHY0606007 funded this work.

II. INTRODUCING EPICS

EPICS is a distributed control system that was originally developed collaboratively at Los Alamos National Laboratories and Argonne National Laboratories in 1989 as an off-shoot of the Ground Test Accelerator (GTA) control system at Los Alamos National Labs. EPICS has been adopted to control over 30 accelerators world wide, several large detection systems, telescopes and is also in use in several commercial applications/industrial applications. [4].

In the initial versions of EPICS, work was allocated to I/O controllers (IOCs), and console systems. The IOC systems at the time were typically board level embedded products running the WindRiver vxWorks Software[5]. As i386 computing became increasingly powerful and cost-effective, EPICS IOC software has migrated to these systems and can run on Windows32, Linux, and Solaris86 operating systems. Furthermore, for smaller systems, the line between the IOC and console computer blurs since general-purpose computers are capable of running elements of both components.

A typical EPICS deployment is shown below in Figure 1:

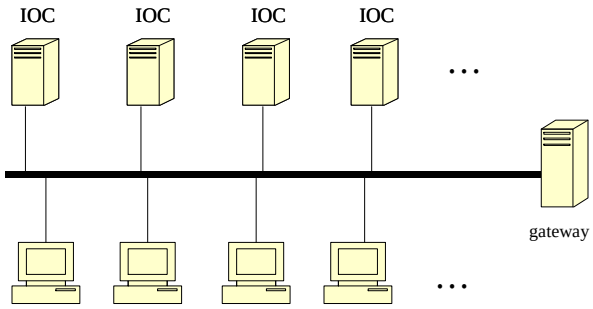

Figure 1 A typical EPICS Deployment.

IOC nodes are attached to the hardware either directly or, increasingly, via serial links and private subnets that they gateway on behalf of the EPICS channel access protocol. As more and more hardware interfaces are network capable, the IOC role is increasingly that of a protocol translator. Console systems run applications with which humans. The gateway system servers two purposes:

1. It is an access point that can determine which systems outside the EPICS control system are allowed to access EPICS channels and how.
2. It does broad/multi-cast traffic filtering. The EPICS channels (or process variables as they are called) are not listed in a centralized database. Instead a broadcast discovery protocol similar to ARP is used to locate the node that serves a specific process variable.

The EPICS process variable is stored as an IOC resident 'database record'. The name of the entry (e.g. ATHING) can typically be read to retrieve some hardware value. Descriptive information about ATHING may be found by reading other fields of the ATHING record. For example, the engineering units of ATHING are, by convention stored in ATHING.EGR.

The interesting thing about the EPICS channel access layer from the point of view of the console application is that there is no actual distinction, other than convention between accessing a process variable that represents hardware and a process variable that is some other field in the database record associated with that hardware.

The IOC software operates by cycling through database records calling handlers for each record that are intended to update the record's fields from the hardware and the hardware from the record's fields. Consider a simple example, a power supply. The power supply has a request voltage and an actual voltage. It an be turned on or off. It has a status that can describe its state that might be any of on, off, or interlocked. A record for this hypothetical power supply may have the structure shown in Table 1 below:

Table 1 A Sample EPICS database record.

Field	Meaning
PS1	Requested Voltage (write) Actual Voltage (read)
PS1.EGR	Engineering units of the requested voltage (read; returns "Volts").
PS1.STATUS	Status of the supply (read; returns "On", "Off" or "Interlocked").
PS1.REQ	Requested voltage (read only)
PS1.ON	Write 1 to turn on, 0 to turn off.
PS.TYPE	Type of record e.g. PSUPPLY

Note that by convention the name of the record is written to set the device and read to retrieve the actual value of the device. The database driven structure of EPICS provides several advantages.

1. Having created a record structure, and driver new instances of a power supply can be created by simply creating new database records and connecting them to the driver software (record fields not shown could provide actual hardware connection information to the driver, e.g. the serial port device the power supply was connected on, or a TCP/IP address).
2. Having described a power supply controller via a database record, only a new driver needs to be written to control a new type of power supply with similar application layer control characteristics.
3. Changing the hardware allocations of specific named devices is not a matter of changing software, but only

of changing the database and can be done while the system is running.

4. EPICS supports creating new devices by creating new database record types (structures), creating instances of them and device driver software to support them. Database records are described via a database meta-language that is used, in con junction with database definitions, to create record instances.

Each channel has a 'native data type', but all channels can be read as a string. This is a concept that is similar in nature to the dual ported Tcl_Obj used in the Tcl internals and API, however the 'native type' port is fixed and cannot be changed. Nonetheless, to some extent, software can be written that reads and controls EPICS channels that adhere to the Tcl EIAS (Everything Is A String) philosophy. It is also possible to obtain a process variable's 'native data type' and we will show in Section IV how we use that in the epics package to perform more accurate string conversions that EPICS itself does.

III. Tcl and Epics in the past

Research indicates two existing Tcl/Tk packages that support EPICS. These are ET[6], and IT[7]. These are both bundled in the EPICS caTCL extension. It turns out that IT is simply an extension of ET that can export data to the IDL data visualization and analysis tool[8]. I will therefore not discuss and analyze the strengths and weaknesses of IT as they are identical to ET with the additional requirement that IDL be available to make full use of its capabilities.

ET is delivered as an extended wish shell, et-wish. Et-wish provides the command **[pv]**. The **[pv]** command is an ensemble that allows Tcl applications to link Tcl variables to EPICS process variables, set process variables from EPICS channels and check the status of the connection between EPICS channels and the underlying application variables. There are some drawbacks however:

- Et is not a loadable package and requires a special shell; et-wish
- Et requires blt and internally uses its vector type.
- Et usage is not very Tclish in particular:
  - Tcl variables are type sensitive giving the impression that Everything Is Not a string
  - Tcl linked variables are not automatically updated by et-wish but must be manually updated and manually set.
  - Process variables themselves don't actually have a good object model. There's the PV command, and there are variables linked, there's no direct handle for a process variable that is being manipulated by the program.
- Et does not interface well with Tk, (because of the need to manually update linked variables)
- ET forces application designers to build widgets appropriate to control rather than providing a library of control widgets.

I felt the drawbacks of et-wish were sufficient to justify the effort required to build a new Epics interface to Tcl/Tk. Furthermore, since I already had epics channel access layer encapsulating classes, I felt I had a good leg up on that development process by interfacing these classes to Tcl through my Tcl++ partial encapsulation of the Tcl API.

The vision I had for Tcl/Tk support for EPICS is shown in the software-layering diagram below:

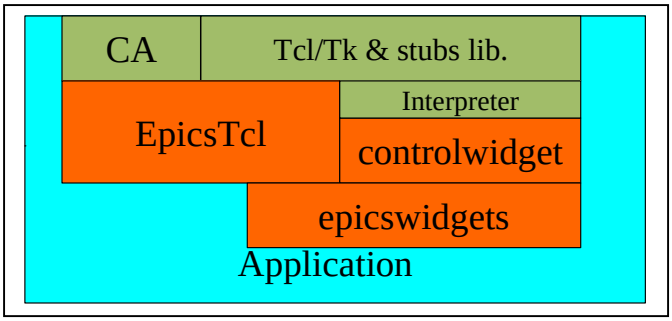

Table 2 below is a key to the boxes in the figure in figure 2.

Table 2 Key to figure 2.

Item	Meaning
CA	The EPICS Channel Access library.
Tcl/Tk&stubs lib	The Tcl API and the stubs library that provides a version independent front-end to it.
Epics/Tcl	A new loadable package that is stubs enabled providing Tcl-ish access to EPICS process variables.
Interpreter	A Tcl interpreter instance
Controlwidget	Pure Tcl widgets for arbitrary control applications.
Epicswidgets	EPICS aware mega widgets.
Application	A console application.

In the next section, we will describe the red components of this diagram.

IV. NSCL Support for Epics and Tcl/Tk

A. *The epics package*

The epics package is about 9000 LOC of C++ software, much of it (4300 LOC) the TCL++ wrapping of the Tcl API, and much of the rest (3000 LOC) a previously written C++ wrapping of the EPICS channel access layer (ca). The epics package (epicstcl for short) provides an object-oriented interface to EPICS process variables. This support is summarized in Figure 3 below:

The **epicschannel** command creates a new epics Process Variable object and a Tcl command that has the same name as the process variable. Operations on the process variable are performed via that command, which, as Figure 3 shows is an ensemble command.

```
epicschannel pvname
pvname get ?count?
pvname set value-list ?format?
pvname link tclVariableName
pvname unlink tcvVariableName
pvname listlinks ?pattern?
pvname updatetime
pvname values
pvname size
pvname delete
```

Figure 3 epicstcl Command summary.

Prior to describing how the package operates, I want to make a slight digression to describe some of the support epicstcl provides for 'programming in the large'. Programming in the large support considers the fact that almost certainly the same process variable will appear in different places on the same application simultaneously. This can lead to code sequences separated physically and temporally by a large distance like those shown below:

```
epicschannel achannel
achannel link achannelVariable1
…
epicschannel achannel
achannel link achannelVariable2

…
achannel delete
…
achannel delete
```

Which raise questions like:

1. What should the second **epicschannel** command on the same process variable as the first do?
2. What should the second **link** subcommand on the same process variable do?
3. What should **delete** do?
4. What should **unlink** do in the event a process variable is unlinked from its Tcl variable?

Good support for programming in the large requires that "the left hand not have to know what the right hand is doing" so that tight module and user interface coupling can be avoided. Therefore three design decisions were made to support programming in the large:

1. Process variable objects have a reference count and the epicstcl package internally maintains knowledge of the process variables that have been created. Duplicate process variables don't actually create another object, but instead increment the reference count. The **delete** operation similarly decrements the reference count and only deletes the underlying channel object/command when the reference count reaches zero.
2. The mapping of process variables to Tcl variables is one to many. That is more than one Tcl variable can be simultaneously linked to an epics process variable. Changes to the process variable are reflected in all linked Tcl variables, and a Tcl scripted change to any linked variable will cause a set to the underlying process variable (which eventually will cause a change in the value of the process variable that in turn will update the value of all the other linked Tcl variables).
3. Linked Tcl variables also have a reference count and epicstcl maintains internal knowledge of these links in a manner similar to the channel objects themselves. This supports a channel being linked to the same Tcl variable more than once.

The EPICS ca library provides 'channel access'. Ca allows access to EPICS process variables. In addition to allowing the application to poll the current values of a process variable, and to set new values, EPICS has an event model that supports notifying the application when an epics variable has "significantly changed". The significance of a change can be defined in the EPICS database records for a process variable.

EPICS performs this notification via threading, and the notification may occur in an arbitrary thread relative to the thread that requested the notification. It is therefore important to get the threading model right with respect to Tcl in order to avoid thread related failures in the Tcl interpreter.

Tcl/Tk supports an apartment-threading model. This model states that:

- A thread can have many interpreters.
- Each interpreter can for the most part be interacted with only in the thread that created it (each interpreter has only one thread).
- API Functions exist to post events to the event loop of an interpreter running in an arbitrary thread.

On the other hand, it is not possible to predict which application thread will receive an EPICS update notification. Therefore the epicstcl loadable package updates Tcl variables by posting an event to the interpreter that owns that variable rather than directly updating the variable itself.

Initial versions of the package always read the string version of the channel in keeping with Tcl's EIAS philosophy. Users discovered, however that EPICS's floating point to string representation conversion functions were inadequate, especially for process variables containing small values. For example, a beam current monitor that was displaying a few nano-amps of beam (e.g. $5x10^{-9}$ nA) would be converted to the string "0.000". Therefore, the epicstcl package reads each process variable in its native type. When the channel connection event is processed, a native-type to string converter is associated with the native data type.

The threading model of EPICS also leads to some interesting edge cases. Consider the script:

```
epicschannel achannel
achannel     delete
```

The first command expresses an interest in the EPICS process variable *achannel*. This:

1. Creates a new Channel object in the extension. The channel object requests an attachment to the process variable named *achannel*.
2. Creates the Tcl command **[achannnel]**
3. In a separate thread, EPICS will notify the Channel object that the process variable was successfully located and attached. This happens asynchronously.
4. Once the channel has been successfully attached, the channel object can express an interest in update notifications.
5. Update notifications can then proceed asynchronously and in an arbitrary thread.

The second command declares the application is no longer interested in the channel. This:

1. Detaches the channel from EPICS
2. Deletes the **[achannel]** command.
3. Deletes the channel object

The script shown will typically delete the channel object before the asynchronous notification that the channel has been connected and, often, prior to the actual connection itself. Thus care must be taken to cancel these notifications or to discard notifications for channels that have been already deleted.

Similarly each low-level channel object has associated with it a semaphore object (implemented on Unix-like systems as a pthreads semaphore and on Windows systems as a Critical Section) to ensure synchronization of internal data structures within the multiple threads that may be executing in an object. These are wrapped in objects that acquire the synchronization primitive on construction and release on destruction so that code of the form:

```
{
    CriticalRegion lock(id);
     ….
}
```

Will maintain the appropriate lock discipline even in the presence of C++ exceptions. Tcl semaphores are not used because this level of the code is intended for re-use in non-Tcl applications.

B. The controlwidget system independent widgets.

While EPICS is the dominant device control system at the NSCL, there are other control systems in simultaneous use. These include various small Labview systems as well as some ad-hoc systems for special purpose applications.

The Widget support for building console applications is therefore broken into two layers. The lowest layer provides some re-usable widgets that are independent of the control system. These operate very much like normal Tk widgets in the sense that they may have **–variable** options or **set/get** methods that some control system aware software can use to manipulate the widget appearance to correspond to the appearance of some control system parameter.

Snit[9] was used to create these widgets. I have had many pleasant experiences using Snit as a mega widget framework, and this project was no exception.

The following widgets were written:

- Led – An indicator that simulates a light.
- Meter – A vertically oriented rectangular meter.
- RadioMatrix – A rectangular array of radio buttons that can be used to chose one possibility from several.
- TypeNGo A type in widget coupled with a button that commits the value in the entry to the control system. The entry supports validation that is invoked when the button is clicked. This allows the application to be certain that a variable that expects a number gets a number e.g.

To give a sense for how these widgets work, Figure 4 below shows a test script for the meter widget. In this case, the 'control system' is just a proc that runs every 100ms and jitters the meter value.

C. *The controlwidget EPICS aware widgets*

The ultimate intent of our work is to make it easy to create control system applications for EPICS at the NSCL. To do this I have also written a set of EPICS aware widgets. In most cases, EPICS awareness means that these widgets have a **–channel** option that binds the widget to display/control a specific process variable in the EPICS control system.

The EPICS aware widgets have been implemented as a mix of **snit::widget** and **snit::widgetadaptor** 'classes'. Where possible, they are implemented on top of the widget set described in part B. of this section. For example, there is an **epicsMeter** widget. This is implemented in terms of the **meter** widget described in section B.

```
package require meter
namespace import controlwidget::*

set metervar      0.5

set jiggleMax     5
set jiggleAmount  0.1

meter .meter -variable metervar \
             -from -1.0 -to 1.0
pack .meter

proc jiggle ms {
    global metervar
    global jiggleCount
    global jiggleMax
    global jiggleAmount

    after $ms [list jiggle $ms]

    set jiggle [expr   \
          rand()*$jiggleAmount -  \
        $jiggleAmount/2.0]

    set metervar [expr $metervar + \
                $jiggle]


}

jiggle 100
```

Figure 4 Test script for meter.

The EPICS aware widgets that have been written include:

- **EpicsButton:** provides several types of epics aware buttons including a pair of buttons for e.g. on/off a single button that can toggle on/off states, and a button that can a process variable to an arbitrary value when clicked.
- **EpicsEnumeratedControl:** provides a wrapping of the **RadioMatrix** widget described in part B of this section.
- **Epicsgraph:** provides a wrapping of the BLT graph widget that allows one to graph the time evolution of one process variable against the time evolution of a second (see also **Epicsstripchart**)..
- **EpicsLabel, EpicsLabelWithUnits:** provides a read-only display of a process variable or a process variable with its engineering units.
- **EpicsLed:** an EPICS aware wrapping of the **Led** widget described B. above.
- **EpicsMeter:** an EPICS aware wrapping of the **meter** widget described above.
- **EpicsBCMMeter:** an EPICS aware wrapping of the **meter** widget along with range controls suitable for use with NSCL Beam current monitor devices.
- **EpicsPullDown:** an EPICS aware pull down menu that can present a set of choices for the value of a process variable.
- **EpicsSpinBox:** an EPICS aware spinbox.
- **EpicsTypeNGo:** an EPICS aware wrapping of the **typeNGo** widget.
- **EpicsStripChar:t** an EPICS aware wrapping of a BLT stripchart widget that allows time series data for epics process variables to be displayed.

V. Software status and level of Adoption

The software is currently stable at version 1.4-001. This version has been tested on Windows XP,2000 Linux and MAC OS-X. I have used this software routinely in my work providing EPICS interfaces to the experimenters. It is provided on the conference CD along with some installation instructions. See the software subdirectory of the CD subdirectory for this paper.

I have not been able to interest the NSCL controls group in this software. Instead they have embarked on an ambitious project to build portable user interfaces using Qt and C++. They estimate this to be a multi-year project, however in the meantime, laboratory administration has given the go-ahead to accelerator operators with software development experience to use this to develop their own user interface software and they have done so with great enthusiasm.

The epicstcl package and associated mega widgets have served as an enabling platform for the NSCL accelerator operators to get functioning control panels that meet their needs with better turn-around times than they have had in the past, and without waiting for the completion of the Qt/C++ application

framework described in the previous paragraph. The project has yielded benefits both for our experimental user group and for the operations program at the NSCL.

VI. References

[1] http://www.aps.anl.gov/epics/index.php
[2] http://www.nscl.msu.edu/tech/devices/s800
[3] http://www.springerlink.com/content/r3224712r7n17864/fulltext.pdf
[4] http://www.aps.anl.gov/epics/projects.php
[5] http://www.windriver.com/products/platforms/
[6] Bob Daly http://www.aps.anl.gov/epics/EpicsDocumentation/ExtensionsManuals/TclTk/et.tcltk.html
[7] Bob Daly http://www.aps.anl.gov/epics/EpicsDocumentation/ExtensionsManuals/TclTk/it.tcltk.html
[8] http://rsinc.com/idl/
[9] http://en.wikipedia.org/wiki/Snitsnit