"Dashboard": A Knowledge-Based Real-Time Control Panel

De Clarke
UCO / Lick Observatory
Santa Cruz, CA

"Dashboard" : A Knowledge-Based Real-Time Control Panel

De Clarke
UCO / Lick Observatory
Santa Cruz, CA 95064
de@ucolick.org

Abstract

This paper describes the use of Tcl and Tk to implement a "soft" or generic GUI for real time control systems. UCO/Lick Observatory is using Tcl/Tk in conjunction with a relational database to implement a suite of code for instrument control and observing at Keck Observatory. One Tcl/Tk application serves as both the GUI builder and the GUI. It relies on information from an authoritative database to configure its behaviour. The project illustrates the use of Tcl and Tk as the common language holding a complex project together, and the particular suitability of Tcl to database applications. It also illustrates a software design philosophy in which an online database engine is an integral part of software design and deployment, rather than the target of the application.

1 Background

1.1 How astronomers handle data

The astronomy community uses, for archival and interchange of image and tabular data, a data storage convention known as FITS. The header of a FITS file consists of a number of keyword/value pairs, embedded in a fixed record format. The FITS standard [FITS] was inspired by Hollerith card images and is somewhat restrictive. Keywords are limited to eight uppercase ASCII characters, and records are strictly constructed by column position. Here is a brief sample from a typical FITS image header.

```
SIMPLE  =                    T / FITS type
BITPIX  =                   16 / bits/pixel
NAXIS   =                    2 / image axes
NAXIS1  =                 2303 / dim in x
NAXIS2  =                 1024 / dim in y
TEMPDET =        -119.72433472 /
DEWARID =                   29 /
TEMPSET =        -119.95216370 /
DWRN2LV =          80.87911987 /
RESN2LV =          32.34042740 /
PWRBLOK =           3.17460322 /
UTBTEMP =           6.50000000 /
UTBTMPS =           5.00000000 /
UTBFANS =                    F /
AUTOSHUT=                    T /
```

A few *mandatory* FITS keywords specify the structure of the subsequent image or table data. Other, "reserved" keywords document common characteristics of data (like DATE, TELESCOPE); the rest of the keyword namespace is commonly used to supply institution-specific, application-specific, or instrument-specific information.

1.1.1 Keywords and values as a control metaphor

At Keck Observatory [KeckObs] in Hawaii, the dome, telescope, and instrument control system is based on the FITS keyword/value model, extended with numerous status and control keywords. Some keywords represent telemetry values which can be read, while others can be written to set instrument operating parameters or to move motors, close solenoids, etc. The control system metaphor is consistent with the archival data storage format, and much status information from the control system is stored in the images acquired there, along with the standard FITS keywords. The control software suite is known as KTL (Keck Task Library) and is described in Keck Software Document Number 28 [KSD28].

UCO/Lick Observatory designed and built the HIRES instrument [HIRES] at Keck-I, and is currently building the DEIMOS [DEIMOS] and ESI [ESI] instruments for use at Keck-II. During early planning for the DEIMOS instrument, the software team wanted some means of managing the large number of new (or slightly variant) keywords the instrument would need. We constructed a relational database schema for modelling FITS keywords, storing keyword attributes such as datatype, format, read/write access, semantics, etc. The schema rapidly grew in complexity, incorporating new concepts such as interkeyword syntactic and semantic relationships, internal/external representation and unit/format conversions, hierarchical keyword grouping, etc.

When nearly complete, the "keyword database" stored in this schema became a powerful resource from which we could generate documentation, sample FITS headers, and certain repetitive sections of source code; we were also able to extend the application of the schema to model database tables (groups of fields, whose attributes are nearly identical to FITS keyword attributes), and to facilitate the automatic conversion of FITS files into database records or tables and vice versa. More information about this project is available at the project Web page [Memes] and in a forthcoming ADASS paper [ADASS].

We used Tcl exclusively for the application language (basically, sophisticated report generation) and Tcl/Tk for the generic forms-based GUI to Sybase which provides our interactive access to the data. We hope to generate a significant portion of the paper documentation for the DEIMOS critical design review, as well as online documentation and substantial chunks of source code for the finished system, by means of these Tcl applications.

2 The "Dashboard" application: a soft realtime GUI

2.1 Functional requirements

Another challenge facing us was our need for a good GUI (or several) for this very complex instrument. We require an engineering interface for bench tests, development, pre-ship qualifications, etc.; and we require a finished, friendly, highly documented GUI for the end-user (astronomer) who will use the instrument at Keck-II. In the past we have hand-crafted GUIs using C and the Xt toolkit, and some personnel at CARA have used the commercial DataViews [DV] product extensively to design custom interface panels for instruments or instrument subsystems.

Our approach to the generation of documentation, code, etc., convinced us of the significant benefits of maintaining one authoritative source of keyword (i.e., design and specification) information to be used by many applications. It was my personal conviction that the UI should not be a specific, handcrafted product tailored for DEIMOS, but a generic "soft" application capable of reading the keyword database and configuring its behaviour accordingly.

The GUI should provide the user with a body of knowledge about keywords and their legitimate use, and with a "toolbox" of graphical and text widgets which could be associated with these keywords. The end result would be a control GUI for any KTL control system (really, for any keyword/value control system). It should not rely on any commercial or license-restricted software; although we started this project using the Sybase RDBMS, one could use the free PostgreSQL RDBMS [PgSQL] with the pgtcl extension). It should be portable to any Unix platform (including Linux on laptops).

In sum, the finished product should be a dashboard *builder*, not just a dashboard. The UI that ships with DEIMOS should be merely one frozen layout created by an inherently dynamic tool. If we could apply the same tool to any KTL system, then the UI for ESI and for DEIMOS would share development costs, and it should be cheap and easy to design supplemental or improved GUI for instruments already deployed. It should be absolutely trivial to change the surface appearance of the GUI at any time, in response to user requests or operational/procedural changes.

2.2 Implementation: how it works

The "Dashboard" application is a fairly slender body of code relying heavily on a large infrastructure (FIGURE 1). It requires a set of Tcl extensions as well as KTL shareable object libraries, and access to the keyword database.

Mr. William Lupton of CARA wrote the "ktcl" extension [KSD98] which adds the KTL API to Tcl. Through this API the application can open a KTL "service", register interest in automatic broadcast updates of KTL keywords, and read/write individual keyword values. "Dashboard" also relies on TclX [TclX], and can use the VU widgets [VUW], the BLT plot widget [BLT], and the LLNL Turn-Dial widget [TurnDial]. It is fairly trivial to add

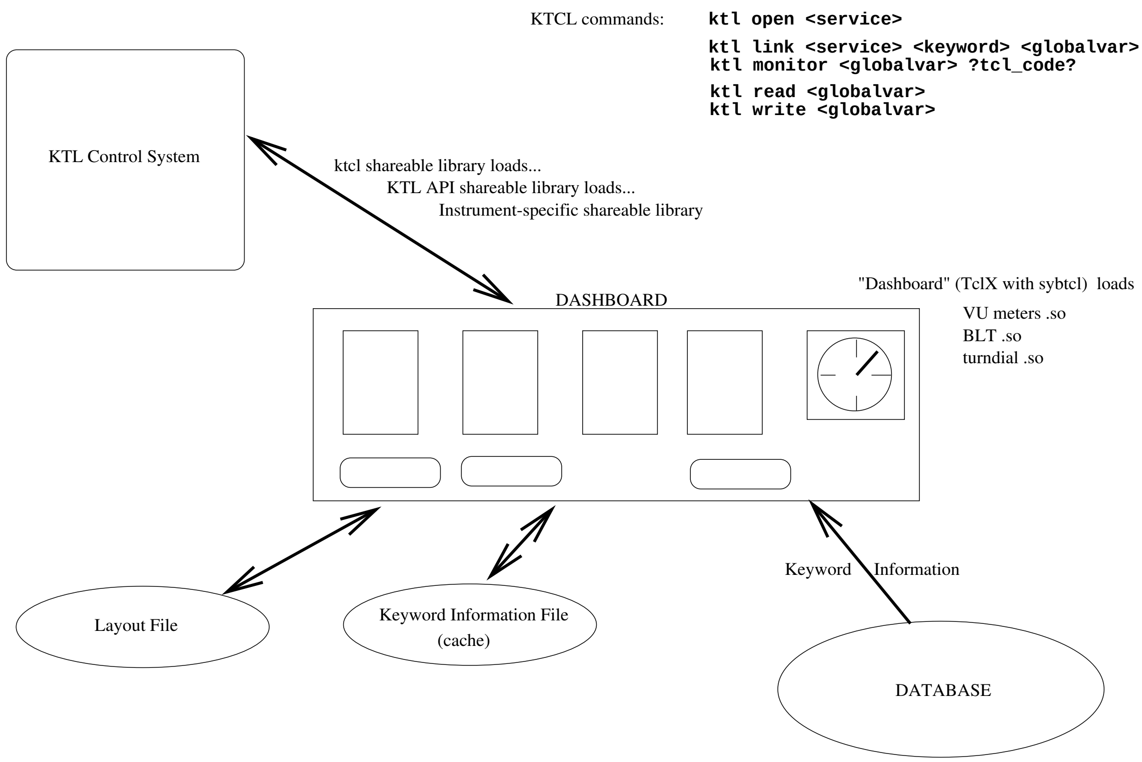

Figure 1: Dashboard Function Diagram

new “meter” types to the dashboard. We used the Sybtcl [SybTcl] extension for access to the keyword database.

The application reads from the database (or from an ASCII save file of information originally read from the database) the complete descriptions of a set of keywords corresponding to a KTL “service”, i.e. the keywords relevant to a particular instrument or subsystem. It creates a canvas on which the user/designer can deploy “meters” and “graphics”. Meters are associated directly with keywords, and display the keyword values. Some meters can also be used to set keyword values. Graphics can decorate and annotate the dashboard, and change their state to flag various conditions (more on this later).

How is this done? The KTL API extension implements a `ktl link` command which associates any given keyword with a global Tcl variable; the global variable can then be used as the argument to a `ktl read` or `ktl monitor` command. If the keyword is being monitored, the KTL control system sends out broadcast messages on each value change, to all processes which have registered an interest in that keyword. The ktcl extension responds to these broadcasts by automatically updating the global variable with the new value (and optionally, by executing a user-defined Tcl code section). The application can explicitly read the value at any time; however, network traffic is reduced by permitting the KTL control system to broadcast values only when they change.

A detailed description of how KTL and the telescope/dome/instrument control mechanism really work is beyond the scope of this present paper. However, for those who are interested in realtime control systems, the Keck Software Documents previously cited plus a Lick Observatory Technical Report [Music] should provide an overview of the infrastructure of which “Dashboard” is the topmost layer.

When the dashboard user creates (e.g.) a “TextBox” meter using the Tk entry widget, the application simply uses the global variable created by `ktl link` as the textvariable for the entry widget, and the display is automatically updated on each KTL broadcast. For widgets which do not have the “associated textvariable” feature, the optional Tcl callback code is used to execute an update procedure for that meter type. In addition to the global

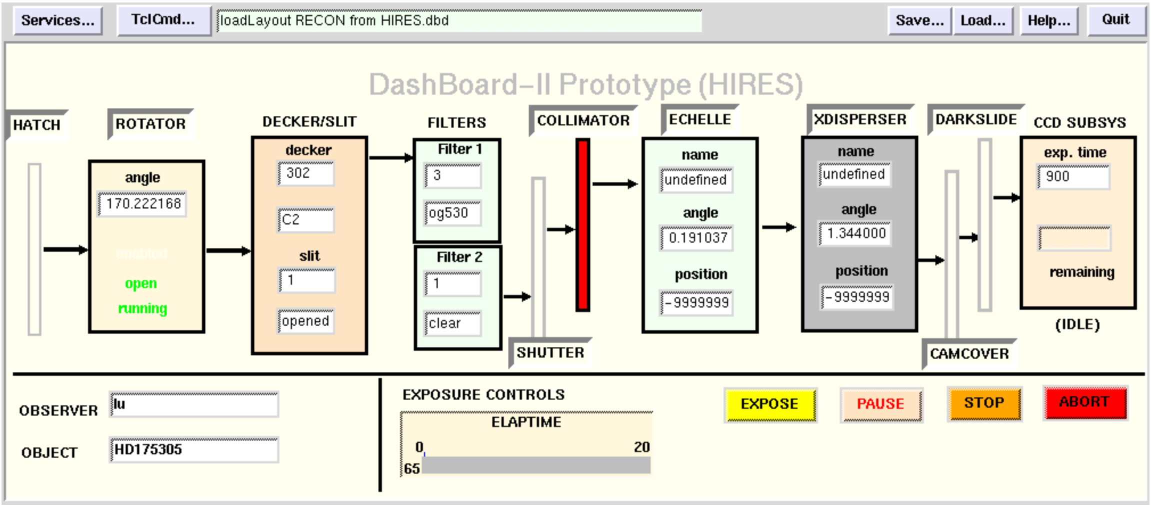

Figure 2: Prototype dashboard for HIRES instrument

variable created by `ktl link`, the application maintains for each keyword a "desired value" variable, which the user can adjust; a Commit operation is supported in which the user's "desired values" are written back to the KTL system.

In detail, the application maintains for each keyword a list of Tcl commands to be executed each time that keyword changes. The callback code uses the keyword name as an index into an array of these lists of commands, retrieves the list, then evaluates each command. Thus any number of meters dependent on one keyword can be be updated when that keyword value changes; each command is `eval`'d, so any number of meters dependent on one keyword value are all updated by the one procedure call. (In the first draft of the code, I used the Tcl `trace` mechanism; but using the callback code option of the `ktl monitor` command reduced the lines of code slightly and looked cleaner.)

2.3 Implementation: features

Using a canvas as the dashboard surface, it was easy to set up bindings for positioning meters by dragging, and bindings for editing meter configurations interactively. Only slightly more challenging was the "detachable" feature, which permits the user to detach a meter entirely from the dashboard into a separate toplevel, then replace it in its correct location at will.

For end-user convenience, the dashboard supports "pseudokeywords": the user/designer can define a pseudokeyword whose value is an evaluable expression involving numeric constants, global variables, and one or more other keyword names (thus a pseudokeyword `REMTIME` can be defined as `EXPTIME - ELAPTIME` so that the user can easily make a count-down meter measuring remaining exposure seconds). The expressions are entered in a simplified form which is expanded and sanity-checked by the application before being evaluated. Expressions are stored in both simple (for user editing) and expanded form.

Graphics are used to decorate the canvas, providing text labels, lines to delimit groups of related widgets, and geometric shapes symbolizing, e.g. hardware or software subsystems with which the user can interact. Bindings for positioning and editing graphical objects are consistent with those for meters.

The ability to evaluate expressions (for pseudokeywords) is also used to implement "conditions", boolean expressions involving keywords and global variables, whose evaluated result can be used to determine the configuration of dashboard elements. Each meter or graphical object (including the canvas itself) can be associated with not just one set of attributes, but an array of attribute sets. Each attribute set is associated with a condition controlling the application of those attributes; the base set is associated with a null or "Normal" condition. The user/designer can easily and quickly add more attribute sets and conditions, to make objects on the dashboard surface change their appearance (such as content, size, background/foreground colour, etc.) in response to KTL events. The designer/user edits meter and graphic attributes, establishes conditions, defines pseudokeywords, etc. using GUI forms invoked interactively from the dashboard. The results of these changes are immediately visible.

The application can be configured to display a "transparent" graphic for an open shutter (value of keyword `SHUTTER` is "open"), and a black (or larger, or both) graphic when the shutter is closed; a large red warning message can appear on the dashboard surface in response to an undesirable condition. Meters can "turn red" when the value they represent exceeds a certain limit; buttons can become inoperable when the instrument condition prohibits their associated action. The attribute set of dashboard objects is the list of their native Tk attributes, plus a small superset for the designer's convenience (`visible` is one such convenient attribute).

In Figure 2, for example, the HATCH rectangle will be solid black when the hatch is closed, but hollow (as shown) when the hatch is open. The horizontal arrows will appear when light is passing between the stages of the instrument at the points indicated, and will disappear when no light is present at those points. The Image Rotator object will turn gray and fade almost into the background when the rotator is out of the light path. Bitmaps representing glowing light bulbs will appear when comparison lamps are turned on . . . and so forth. The chain of "photon arrows" is probably the most challenging of these animation problems, and requires a chain of pseudokeywords with definitions as follows:

```
HATLIGHT
  "HATOPEN == 'open'"

ROTLIGHT
 "( ( HATLIGHT ) && ( ( IROTOUT ==
  'out of light path' ) ||
  ( IROTCVOP == 'opened' ) ) )"

LAMLIGHT
  "LAMPNAME != 'none' "

DSLIGHT
  "( ROTLIGHT || LAMLIGHT ) && ( SLITWID > 0 )"

SHUTLIGHT
  "( DSLIGHT ) && ( SHUTCLOS != 'closed' )"

COLLIGHT
  "( SHUTLIGHT ) &&
  ( ( ( COLLBLUE == 'blue' ) &&
  ( BCCVOPEN == 'opened' ) ) ||
  ( ( COLLRED == 'red' ) &&
  ( RCCVOPEN == 'opened' ) ) )"

...
```

The application can now use those pseudokeywords for conditional configuration of graphics. The arrow between the Rotator and the Decker/Slit modules has two conditions: invisible (normal) and "lit". The "lit" condition is contingent on either a lamp being on or light getting through the rotator:

```
LAMLIGHT || ROTLIGHT
```

The GUI designer needs more flexibility in responding to conditions than merely the alteration of the GUI appearance. Evaluable conditions are also used in a simple error/warning message system with popups, mailed or displayed messages, and optional Tcl code to execute on any condition. The UI designer can easily cause any arbitrary action to take place (including execs of shell commands, delivery of mail messages, etc.) upon keyword-related conditions. In this case the object is an Alarm, and has an associated array of conditions and actions much like the condition/attribute arrays for visible objects. Figure 3 shows the Alarm Editor.

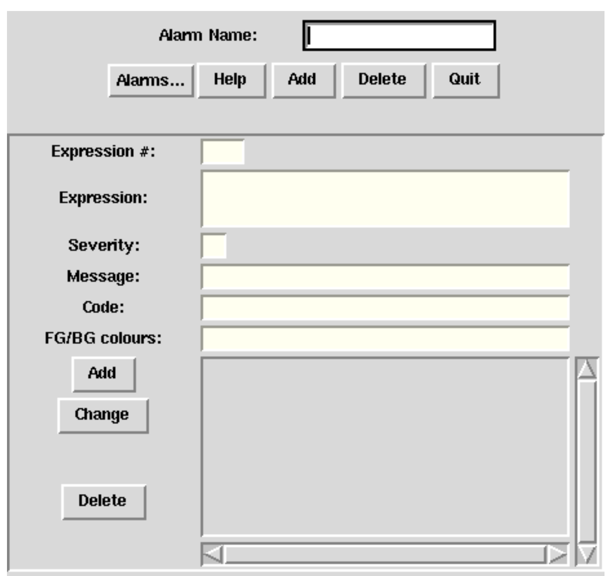

Figure 3: Pop-up editor for defining Alarms

Here is a sample alarm condition in simplified and expanded notation. The simplified notation is what the designer entered into the Condition box in the Alarm Editor.

```
( LAMPNAME == 'none' ) &&
  ( LMIRRIN == 'in light path' )

( \$hires(LAMPNAME) == 'none' ) &&
  ( \$hires(LMIRRIN) == 'in light path' )
```

As you can see, the expansion process consists of converting keyword names and pseudokeyword names to Tcl global variable references. After this, the expression can be evaluated at uplevel #0 (evaluation in this case takes place on any change to the value of either `LAMPNAME` or `LMIRRIN`). If this expression is true when evaluated, a message will appear in a popup alert box:

```
No lamps, yet lamp mirror in.
Occultation?
```

Upwards of 400 keywords are needed to control and monitor the DEIMOS instrument; obviously there is not enough screen real estate on even the largest standard monitors to display this much information at any reasonable size, nor can the average user perceive and understand that much information at one time. Therefore the dashboard needs hierarchy; it can invoke and dismiss sub-dashboards offering detailed control, while presenting an overview of the system at the topmost, introductory screen. The mechanism for invoking sub-dashboards is the familiar and intuitive "double-click" used in many window systems to unfold or invoke windows, bound to any graphical object; however, the user/designer can attach any Tcl code to the double-click binding, as well as or instead of the procedure call to invoke a subdashboard.

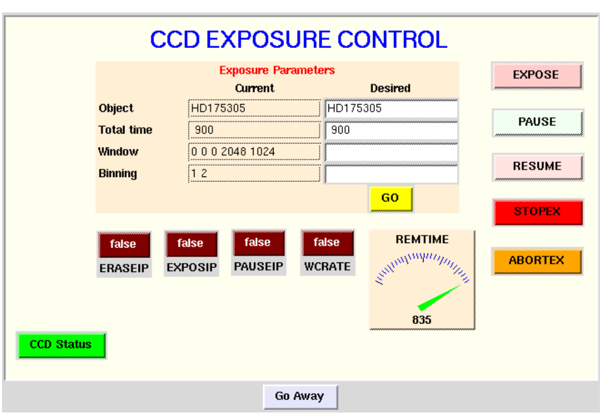

Figure 4: Sub-dashboard invoked from HIRES main dashboard

Figure 4 shows a subdashboard which was invoked from the CCD stage of the main HIRES dashboard with a double-click. This subdashboard offers more detailed control of the CCD control subsystem. The "CCD Status" button offers a third level of detail (Figure 5) in which a BLT graph widget is used to plot detector temperature against the liquid nitrogen level in the dewar. The PS button makes a PostScript file of the current plot.

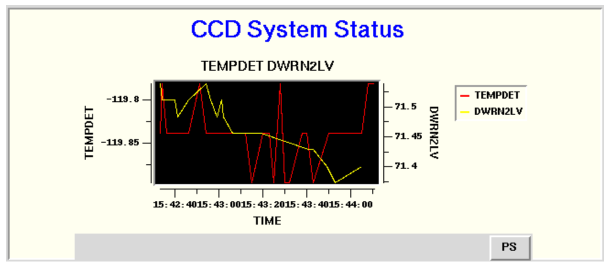

Figure 5: Sub-sub-dashboard invoked from CCD Status button

For design flexibility, the dashboard supports three hybrid screen items: a generic Button, generic Menu, and generic Entry widget. Unlike a Meter, these items are not bound to any particular keyword, and unlike a Graphic they are standard Tk widgets embedded in a canvas, not native canvas items. The user/designer can configure the generic menubutton and button's "command" attribute freely, to execute any arbitrary Tcl code; the generic entry widget can be associated with any global variable (for example, if the application requires the UI to display Desired as well as Actual values simultaneously). These generic widgets support the same conditional attribute mechanism as the other dashboard components. (Example: several of the stage labels in Figure 1 are actually menus offering control of the covers or other basic functions for the optical stages. The menu items would be, e.g., "Open Cover," "Close Cover," "Into Light Path," "Out of Light Path.")

The dashboard operates in several (nonexclusive) modes. In "engineering" mode, the user can edit the dashboard freely; each writable Meter has individual edit and commit buttons, and some extra function buttons are provided in the frames around the central canvas. In "observer" mode, the screen is smaller and less complicated, omitting extra function buttons and invidual Meter buttons. In "developer" mode, Meters have large obtrusive frames which make them easy to reposition and edit, whereas in "deployment" mode these frames are invisible and the user can neither reposition nor edit items on the screen. In "safety" mode, where any KTL write command would normally have been executed the application will instead log a message to stderr. In "fake" mode, the dashboard will not even try to connect to a KTL service but will use fake values for all keywords; this permits early design to be done before the supporting keyword libraries are compiled, or before a running instrument control system is available.

Release 1 of the dashboard knows how to use the VU Dial Gauge, Bar Gauge, and Stripchart meters, the BLT plot meter, and the TkTurndial meter. It also provides a TextBox (decorated entry widget), Odometer (undecorated entry widget), Grid (gridded frame of entry widgets with actual and desired values), On/Off Light, and Keyword Action button. It offers the Line, Rectangle, Oval, Arc, Polygon, and Bitmap canvas items as graphics, and the non-keyword Button, Menu, and Entry. It supports conditional configuration of all meters and graphics, and delivery of alarms (or any arbitrary Tcl code execution) on any condition. A command-line window and a simple script editor are provided for the user, for the direct typing of Tcl commands as an alternative to GUI interaction. Screen layout can be saved to and reloaded from plain ASCII files; all pseudokeyword and condition information is saved as well as the layout and configuration of the main and all sub-dashboards.

3 Positive aspects of the "Dashboard" approach

3.1 Practical advantages

The immediate appeal of the dashboard is in its flexibility and the speed with which interfaces to KTL-based instruments can be generated. The interface shown in Figure 2 was created just as a demo, to exercise and debug the first release of "Dashboard"; it took about 4 hours to create the toplevel screen, and about another 3 hours to create four subdashboards (which pop up on double-clicks from the main board).

The dashboard is also (usually) "live" during design and prototyping; there is no compile/link/test cycle. As soon as the keyword information is absorbed and a widget is created, that widget is "real", watching a real keyword value in a running KTL system (unless the designer is running in fake mode).

If the engineers change their minds about hardware/software design or function, those changes are first reflected in the keyword database. The application then effectively retools itself to match these design changes, always provided that the keyword database is properly maintained. Since the keyword database is the foundation of a number of applications, not just one, the likelihood of its proper maintenance (and of someone's noticing any errors or inconsistencies) increases with the number of applications that depend on it. Also, the availability of interactive X11 forms (Figure 6) and other GUI and command-line tools for editing database information makes it relatively easy for developers to keep the central knowledge base current and accurate (as opposed to the manual editing of many distinct copies of the same information in the form of code, config files, man pages, typeset documents, etc.)

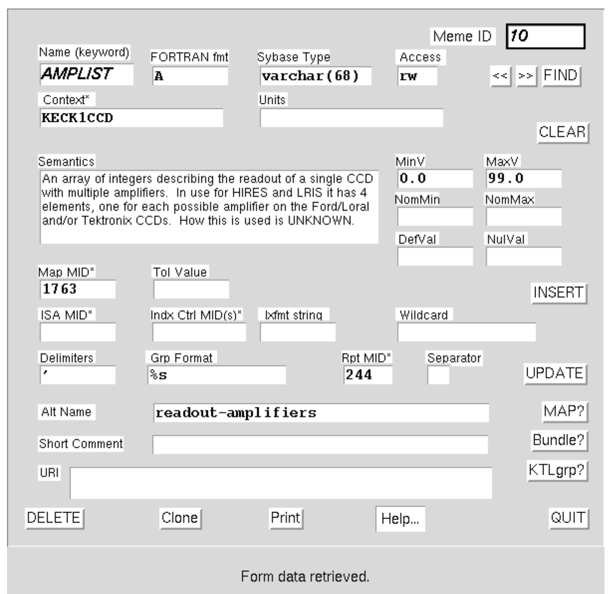

Figure 6: Typical GUI form used to edit keyword data

The dashboard is a single application, with one maintenance cycle and one investment in development, which can be used to provide engineering/diagnostic interfaces, deployed user interfaces, prototype UI designs, etc., for any number of instruments sharing the KTL control protocol. In the past, we developed individual interfaces per instrument per application, and these interfaces were burdensome to improve or repair; as a result, user complaints and feature requests were seldom resolved. The "softness" of the dashboard means that the advanced user can tweak its appearance to suit his/her own tastes, or design entirely original personal dashboards for specific purposes; the deployment mode means that it can be given to naive end users as a "canned" UI. The designer who must support a deployed version can easily and quickly implement most user requests, without tedious recoding and recompilation, by replacing a single platform-independent layout file.

Being based entirely on publicly-available code, the dashboard is free and portable. It runs as well on our Linux laptop, or my Linux home computer, as it does on Dec Alpha or Sun Sparc platforms.

3.2 Design philosophy

All of the above features result in cost and time savings. However, most of them are merely side-effects of what is in my opinion the single really interesting feature of the dashboard, which is its symbiotic relationship to a knowledge base embodied in an online database. In most Tcl/Tk applications involving databases, the database is the *target* of the application; i.e., the information in the database is manipulated by the application. In my "fosql" Tk forms GUI for Sybase, the forms designs themselves were stored in Sybase tables, and the data in the primary forms design table could be edited using the form for that table. However, the *target* of the fosql application was still the data in the rest of the database.

In the "Dashboard" application, the database is not the target of the app; the telescope, dome, and instrument and the data gathered by the instrument are the target of the app. The database is an intrinsic part of the application itself. It contains "information about information", or "meta-data" which the application uses to configure itself and to make certain inferences about its own function. This could be seen as one way of implementing object orientation; a database is the ideal way of representing objects and their attributes, and the code is merely the methods which apply to the objects. We could also regard the database as the equivalent of hundreds or thousands of C source "structs".

Many other applications, such as mail tools, use resource files of one kind or another to configure themselves, thus avoiding recompilation across changes of appearance or function; but in general, each application has its own resource file which is not shared with any other applications. (The X resource database is one exception, being a true online database with a known API, but most X clients have private resource files applicable only to their instance or their class.)

In contrast to the "private resource file" model, a central authoritative body of knowledge about the information on which the application operates – implemented as an online database – forms a conceptual hub about which many applications (such as the dashboard) can be constructed with maximum generality at relatively low cost. The tedious and repetitive type of tabular information which (in our older control systems) is replicated many times in different C sources, vxWorks sources, etc. is here available in one consistent place, accessible online. (It can be cached as FITS files or other ASCII formats in case we lose access temporarily to the live source.)

I should perhaps note here that as well as keyword syntax and semantics, overall system design is also expressed in our database as tables of hardware and software "agents," which pass "keyword" information between them in various formats and media. Using the digraph tools from Lucent Technologies [Dot] we can easily generate information flowcharts for the hardware and software subsystems. Thus the majority of our project design information is online in a highly standardized, codified, machine-readable form; this in turn means that 80 percent or more of our project documents are auto-generable.

The logical conclusion of this conceptual strategy is that design, documentation, and by inference a certain percentage of generable source code, are all manifestations of one body of knowledge, expressed once and maintainable at one central point.

All the usual reasons for choosing Tcl/Tk apply: speed of prototyping and development, low cost of modification and deployment, portability, lack of commercial restraint on distribution. However, there were certain project-specific reasons as well. From prior experience I had already become convinced that Tcl(X) was a near-ideal language for database applications (largely because of its typelessness and its solid list processing features). Lastly, because of the `eval` feature combined with the above, it is remarkably easy to write selfconfiguring multi-purpose applications in Tcl (using methods which have no equivalent in C or other compiled languages, such as the dynamic generation of variable names and on-the-fly generation and execution of code). The "Dashboard" application was a logical outcome of previous positive experience with Tcl and its extensions.

So far, we feel that the "Dashboard" application has been a Tcl success story.

4 Potential applications

Nothing restricts the use of the "Dashboard" application to astronomy or to KTL systems. Any keyword/control system with an API could use the dashboard code, by writing a different Tcl extension and altering about 30 of the 12K lines of code in the dashboard-II application. The "trace" mechanism could substitute for the monitor/callback mechanism, or polling could be implemented for a control system with no monitor/broadcast facility. Dashboard is a shell, in other words, which could be inhabited by applications other than KTL, just as the "FITS keyword" database is a shell which could be inhabited by other kinds of syntactic and semantic information.

5 Future plans

Since the dashboard code is basically object-oriented, one might ask why I didn't use extended [incr Tcl] instead of plain TclX to implement it. An object-oriented Tcl would have been a more natural choice; I implemented some OO-like features the hard way. When itcl becomes a standard extension requiring no core modifications, I'll probably convert "Dashboard" to itcl, and the code will then become smaller and cleaner.

The code currently saves its layout to an ASCII file (Tcl source). I would prefer that it saved this information to a database schema, like its ancestor the "fosql" package. However, development was so rapid during the first six months that I decided to skip the overhead of schema revision and work from flat files. Storing the layouts in database tables would offer far more power and ease of access for global modifications, queries about the tool design, hunting down particular widgets and bindings, etc. I am used to this convenience in the "fosql" package and am already feeling the lack of it.

There is currently no provision for communication between multiple running copies of the dashboard. This seems a serious shortcoming, one which must be remedied before ship date. Since partnered observing is very common (remote observer on the mainland or in Waimea, in close communication with observers on the summit), the ability to share dashboard configurations and information seems essential. This raises all the usual issues of distributed applications (registry, trust, etc) and rather than reinventing all those wheels I'll probably evaluate existing Tcl distributed applictions and copy or adapt a successful design (with the author's permission).

The code does not yet use KTL features like "callback on move complete" which would permit us visual indication of moves in progress, and also to detect and visually flag any drift in stage positions. We are still struggling (on the KTL side) with issues like "estimated time to completion," "stage position tolerance," and so forth. However, once we have decided how to encode these concepts in the schema, or how to implement them as keywords with readable values, it will be trivial to "teach" the dashboard how to use them to construct conditional behaviours.

6 Acknowledgments

The application would not be half as successful without a few very useful features of Tcl/Tk and the TclX extension. In particular, the grid geometry manager is a very welcome new Tk feature which significantly reduced coding time; arrays of keyed lists (tclX) are a perfect analogue to database tables and hence are heavily used in most of my code. Tcl and TclX list functions, and expression evaluation, were essential. TclX's package/library system has been very useful and appropriate for code maintenance.

Much thanks to Mark Diekhans and Karl Lehenbauer (TclX), W. F. Lupton (ktcl), T.Poindexter (sybtcl), G. Howlett (BLT), P.-L. Bossart (tkTurnDial), and the original authors of the VU widget set (F. Gardner, L. Miller, R. Dearden of VU Wellington, NZ). Thanks also to Steve Allen (FITS expertise and KTL code porting) and the rest of the DEIMOS team for their ongoing suggestions and

challenges. And as always, thanks to Per Cederqvist and friends for CVS!

7 Availability

The dashboard code is freely available. Contact me if you would like to experiment with it.

References

[ADASS] De Clarke and S. L. Allen, *Practical Applications of a Relational Database of FITS Keywords*, ADASS Conference 1996 (Virginia)

[BLT] The BLT extension
`ftp://ftp.neosoft.com/tcl/ftparchive/sorted/devel/BLT2.1.tar.gz`

[DEIMOS] The DEIMOS Instrument Project,
`http://www.ucolick.org/~deimos`

[Dot] The graphviz package and tcldg extension
`ftp://research.att.com/dist/drawdag`

[DV] The DataViews Toolkit,
`http://www.dvcorp.com`

[ESI] The ESI Instrument Project,
`http://www.ucolick.org/~loen/ESI/esi.html`

[FITS] The FITS Standard,
`http://www.gsfc.nasa.gov/astro/fits/fits_home.html`

[HIRES] The HIRES Instrument Project,
`http://www.ucolick.org/~hires`

[KeckObs] Keck Observatory,
`http://www.keck.hawaii.edu:8080`

[KSD28] W. F. Lupton, KTL Programming Manual (KSD 28)
`http://www.ucolick.org/~de/KSD/ksd28.ps`

[KSD98] W. F. Lupton, Tcl/Tk/KTL Interface (KSD 98)
`http://www.ucolick.org/~de/KSD/ksd98.ps`

[Music] Lick Observatory Technical Reports: Music
`http://www.ucolick.org/~de/KSD/music.ps`

[Memes] A Database Schema for Representing Meaning,
`http://www.ucolick.org/~de/deimos/Memes`

[PgSQL] PostgreSQL,
`http://www.postgresql.org`

[SybTcl] The Sybtcl extension
`ftp://ftp.neosoft.com/tcl/ftparchive/sorted/databases/sybtcl-2.4`

[TclX] The tclX extension
`ftp://ftp.neosoft.com/pub/tcl/TclX`

[TurnDial] The tkTurndial widget extension
`ftp://redhook.llnl.gov/pub/visu/tkTurndial-2.0b.tar.gz`

[VUW] The Victoria University widgets extension
`ftp://ftp.ucolick.org/pub/UCODB/VUtk41.tar.gz`