The PRUDE Interface
Version 1

Robert McTaggart
Pennsylvania State University

September 20, 1996

Abstract

A graphical interface has been constructed using the Tcl/Tk language that allows one to view and change values for the many variables assigned to each software trigger and each gatemaster trigger. In this version one can only access software triggers, buffers, and service groups that exist in the database.

Contents

1 Introduction

Events recorded by the data acquisition system are run through a filter program to decide which events will be recorded to tape or disk for further analysis. Either the event has passed some set of criteria defined by a software trigger, or the event is chosen simply because a predefined frequency of events are taken.

One may prescale events before they are looked at by the filter, and prescale them before they are written to each buffer. The triggers themselves may need to be reprioritized or assigned to different buffers, and the buffers may need to be assigned to different service groups.

We attempt within this document to detail how these tasks are performed within the interface. Creating new software triggers, buffers, or service groups may be done outside of the interface, but it will be left to another version to produce these items within the interface.

Functions defined in Tcl are called processes. One may use lists or arrays to store information, and one may pass information between different processes by either declaring such lists as global, or including the list as an argument to the process. Two or more Tcl commands may be placed on the same line if a semicolon separates them, otherwise each line has one command. A brief overview of the important aspects of Tcl and Tk to this interface is included in the appendix.

Several key lists used by this interface to communicate with parameters in the database will be pointed out to you throughout this document.

2 Flat Files

Several different configurations of the data acquisition system exist to easily permit the initialization of specific triggers and their associated variables. For instance, taking data with cosmics requires a different set of triggers than taking pedestals with a pulser. Each configuration has its own values for the many database parameters. The default values for these parameters exist in so-called flat files.

Let us examine what is required to change one of these parameters, or add a new parameter without the existence of the interface in the *data* configuration.

The flat files for the *data* configuration exist in the directory

/dart/daqman/dart/config/data/disc.

To change the value of a parameter or add a new parameter, one must edit the respective flat file, such as *filter.disc* or *trigger.disc*. All the flat files used by the DAQ end in '.disc'.

Then while the DAQ is running, one must execute the following commands in a session on fn835x:

```
source /scripts/set_envs data
distcl
```

.

Now you may source the flat file and update the database. Suppose the parameter(s) you wanted to update is(are) in the flat file *filter.disc*. At the distcl prompt, execute the following:

```
source /dart/daqman/dart/config/data/disc/filter.disc
```

.

The database is now updated. However, this can be quite cumbersome at times. The interface allows you change the value of a parameter by typing in a new value within a text widget and hitting the apply button, or enable and disable existing triggers with the click of a button.

However, you must still use this procedure to add a new trigger or reassign triggers to buffers, since these actions require writing strings to the database. At this stage of the interface, there is a problem with reading strings, changing them, writing these strings to the database, and re-reading them successfully due to the way Tcl is made and the routines that read and write to the database, dis_fetch and dis_string, behave.

3 proc triggers

The Tcl process *triggers* in the *.ocprc* file of the configuration is the main process of the interface. This process reads the database, initializes the lists and arrays that are passed between the other processes of the interface, and builds a central menu from which to execute the interface's main functions. It may either be called by some other process, such as Trig_Init, or may be executed from its own button on the main Run Control menu.

The interface may be thought of as being two separate interfaces: one for the gatemaster triggers (i.e. hardware), and one for the software triggers. So changing attributes for the gatemaster triggers will have no effect on the software triggers. The appendix lists the software and gatemaster triggers currently defined.

3.1 Software Lists

The interface fetches two lists from the database concerning the software triggers, and uses them to build the software component of the graphical interface.

The first is */filter/trigs*. This stores the names of every software trigger that exists in the database. The interface stores this list internally as *master_software_list*.

The second list is */filter/trigorder*. This is a list of numbers that reflects (a) which software triggers are active and (b) the current priority of the software triggers. The first element has the highest priority and the last element has the lowest priority. Internally to the interface, this list is *trigorder_list*.

To find the name of the trigger corresponding to the number in */filter/trigorder*, subtract one from this number(Tcl indexes start from zero) to yield the index of the corresponding element in */filter/trigs*. In other words, this number is the index in */filter/trigs* if the indexes started with 1.

Given the number associated with each trigger, the array *soft_trigger* yields the trigger name. Given the trigger name, the array *soft_channel* returns the number. The list *software_list* contains the names of all active software triggers.

The interface will allow you to change parameters for active triggers only. One may make other triggers from */filter/trigs* active and reprioritize them too, as we will see later.

3.2 Hardware Lists

The gatemaster trigger lists are similar to those used by the software. The list of all gatemaster triggers in the database is */trigger/GM/list*. The interface removes elements in this list called 'EMPTY' and stores the result in *master_gatemaster_list*.

Active gatemaster triggers may be found in */trigger/GM/order*, which is stored internally as *gmorder_list*. This list also defines the priority of the gatemaster triggers. The relation between these two gatemaster lists is the same as for the two software lists above, however this relation is set before the 'EMPTY' elements are removed.

The function of the two arrays *gm_channel* and *gm_trigger* are likewise exactly the same as their software counterparts. *gatemaster_list* stores the names of all active gatemaster triggers.

The priority of all gatemaster triggers may be predefined, since their number is fixed. *master_gmorder_list* hard-codes this priority into the interface. This means if a gatemaster trigger is active, it has a higher priority than all other triggers appearing after it in this list.

3.3 Buffer Lists

At this time there is only one parameter read from the database that concerns the various buffers that are sent to different tapes. This is because we can only read strings into the interface, not read, write, and read again. Attempts are being made to rectify the situation so that one may reassign triggers to currently defined buffers or define new buffers.

/filter/bufs contains the names of all defined buffers. Triggers may write to one or more of these buffers, and each trigger has its own prescaling factor for each buffer that it writes to. The interface permits you to change these prescaling factors for the buffers defined in the database. Within the interface, this list is called *dfm_buffer_list*.

3.4 Main Trigger Control Menu

triggers produces a separate window holding several buttons using the Tk graphical language. Each button performs a separate function of the interface, i.e. it pops open another window from which further action may be taken.

The present version of the interface associates the following processes with a button in this main menu: GatemasterTriggers, SoftwareTriggers, ChangeTriggerPriority, EnableSoftwareTriggers, and ExitInterface. Let's examine what each of these processes does.

4 proc GatemasterTriggers

This routine enables and disables the gatemaster triggers(except those that are 'EMPTY'), and sets the prescale factor for this trigger.

For the e+e- trigger,

/trigger/GM/E+E-/enable is 1 if it is enabled, 0 if it is disabled. */trigger/GM/E+E-/prescale* set the prescale factor.

Similar parameters exist for all the gatemaster triggers. The corresponding internal Tcl variables have the same name and are declared to be global. The names of the variables are added to a list that is also declared to be global. Hence any other routine can check to see what the state of these variables are.

When the apply button is clicked, *WriteToDatabase* writes the new variables to the database, and *UpdateGMTriggerLists* updates all the lists that have changed. Triggers that are disabled are removed from the lists, those that are enabled are added to the lists, and then the lists are reprioritized and written to the database if necessary.

5 proc SoftwareTriggers

The software interface constructed by this process allows you to view the trigger mask information from which a particular software trigger is created. Furthermore, one can also change the definition of the trigger by selecting which trigger bits are on or off.

At the top of the interface are buttons for each software trigger. Clicking on one shows the trigger bit-mask information in the checkbutton menu below it. A variable is associated with each checkbutton, so that pressing a checkbutton changes this value to 1, and pressing the same one again switches the value back to 0. These variables are declared global so that other routines that write the trigger mask to the database can access them.

For the *elec* trigger, this mask is set by the database parameter */filter/trigger/elec/mask*. This is a list of integers with 5 elements, each of which was originally supposed to be a 32-bit integer, but we can only work with 31-bit integers(in binary) in the version of Tcl that the DAQ uses. The first 31-bit word represents the bits that are required to be ON for the CCAL analysis. The second element is a 31-bit word that tells you which bits must be OFF for the CCAL analysis. The third represents which bits must be ON for the e+e-/$\phi\phi$ analysis, and the fourth says which bits must be OFF for the e+e-/$\phi\phi$ analysis. The fifth element tells the filter which gatemaster channel this software trigger will accept data from.

Each column of checkbuttons in this interface corresponds to one 31-bit integer. Each of these 31-bit words are stored as decimal integers in the database, so when a trigger is first chosen in this interface, the routine *ShowTrigMask* accesses one element of the mask list and divides it by successively lower powers of two until there is no remainder. Each power of two is associated with a checkbutton, and the corresponding checkbuttons are turned on.

When a new trigger mask is written to the database, the process *ApplyTrigger* accesses the variables associated with each checkbutton, and then adds all the respective powers of two(if that checkbutton is on) to get a new 31-bit word. Then the new list of these words is written to the database.

A new gatemaster channel may be assigned to the trigger by changing its value in the entry box where it is displayed.

None of the changes made to this software interface are made until the apply button is pressed.

6 proc EnableSoftwareTriggers

This particular window in the interface allows one to enable and disable software triggers. However, in contrast to the gatemaster triggers, the software triggers have no a priori list of precedence hard-coded into the interface, since more can be added later.

Once a software trigger is enabled, it is given the lowest priority. You must go to the window controlled by *ChangeTriggerPriority* to place the trigger in its desired location within *trigorder_list*.

A checkbutton menu allows one to enable or disable software triggers. Global variables that take a value of either 0 or 1 are assigned to each checkbutton. When the apply button is pressed, the process *UpdateSoftTriggerLists* deletes any triggers from active lists that have a 0, and add any triggers that have a 1 to the end of the active software trigger lists.

Finally, *UpdateSoftTriggerLists* updates the database and remakes any windows currently shown that use these lists.

7 proc ChangeTriggerPriority

After the desired software triggers have been included in the various lists, the priority of these triggers must be dictated to the interface. The act of removing and adding triggers does not in itself place the triggers in the proper position within the active lists. If some priority were hard-coded into the interface for the software triggers(i.e. new triggers could never be made via the interface), this process would not be necessary.

This window consists of four columns of Tk widgets. The first is a column of buttons, each of which has a name of one of the ACTIVE software triggers. If you want to change the priority of a particular trigger, just click on its respective button. This calls the process *PresetPriority*, which will keep track of this trigger as priorities change. A text widget in this window tells you which trigger is being manipulated. The second column reflects the current priority of these triggers(the one with the highest trigger is placed at the top in both of these first 2 columns). The third column reflects what the new priority will be if the apply button is chosen.

The fourth is a column of buttons with the numbers 1 to how many triggers are included in *software_list*. So if you want a trigger to have the 1st priority, you would click on the '1' button. This will call *NewPriority* and establish the trigger being prioritized in the first position in all the active lists. The other elements will accordingly be pushed down in priority.

Changes are not finalized until the apply button is hit, when the routine

SetPriority updates the lists and writes them to the database.

8 proc ExitInterface

The interface keeps track of all windows that are currently displayed in *window_list*. *ExitInterface* destroys all windows in this list along with all of their associated Tk widgets. It does not write anything to the database. This ends the process *triggers*.

References

[1] John K. Osterhaut. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company. 1994.

A Current Software Triggers

Trigger Name	Trigger Description
elec	e+e- events that are not tagged
goodee	e+e- tagged events (cerenkov and CCAL)
goldee	e+e- tagged events (cerenkov, CCAL, and electron weight)
pbarp	pbar and pi+pi-
phiphi	phiphi
goodgg	2 gamma with invariant mass $>$ 2.5 GeV
goldgg	2 gamma with invariant mass, acoplanarity and akinematics requirements
invmass	neutral event with invariant mass $>$ 2.0 GeV (doesn't pass following 4)
etainvm	found $>=$ 1 exclusive η and invariant mass $>$ 2.0 GeV
piinvm	found $>=$ 1 exclusive π and invariant mass $>$ 2.0 GeV
cmainvm	1st largest energy cluster may be split and invariant mass $>$ 2.0 GeV
cmbinvm	2nd largest energy cluster may be split and invariant mass $>$ 2.0 GeV
etotsoft	$>$ 90% energy in CCAL, Total Energy
etaetot	found $>=$ 1 exclusive η and total energy
pietot	found $>=$ 1 exclusive π and total energy
cmaetot	1st largest energy cluster could be split and total energy
cmbetot	2nd largest energy cluster could be split and total energy

B Current Gatemaster Triggers

Channel	Trigger Name	Description
1	E+E-	e+e- trigger
2	PI+PI-	pi+pi- trigger
3	PHI-PHI	phi-phi trigger
4	2GAMMA	gamma-gamma trigger
5	ETOT80	etot $>$ 80%
6	EMPTY	
7	NTEFF	neutral trigger efficiency
8	ETOT70	etot $>$ 70%
9	LASER	laser trigger
10	STRAW	straw calibration trigger
11	EMPTY	
12	MINBIAS	minimum bias trigger
13	PULSER	pulser trigger
14	FCAL	FCAL cosmic ray trigger
15	EMPTY	
16	EMPTY	

C Overview of Tcl/Tk

Abundant examples of how to use Tcl and Tk are included in the interface. The synopsis here will give you a good start on understanding what the interface is trying to do. For more information on Tcl and Tk, see the bibliography.

C.1 Setting Tcl Variables

Tcl strings may be set in several ways. With few exceptions, changes to strings are not made unless they are done via a *set* command. To set the string 'weather' equal to the string 'cloudy', the following are all equivalent:

set weather cloudy
set weather {cloudy}
set weather "cloudy"

.

However, if you would like 'weather' to be a list of more than one element, only the following are possible:

set weather {cloudy rainy cold}
set weather "cloudy rainy cold"

.

Sometimes it is advised to include each element in a list within braces or double quotes if you want the white space to be included as part of the list element, but most often white space will separate elements in a list.

If you are running a *wish* shell and wish to see what the contents of the Tcl string 'weather' is, type set weather

and the *wish* shell will return ‘cloudy rainy cold’ for example.

Suppose you are within a Tcl process and you want to set the string ’forecast’ to the contents of ’weather’. This may be done by either

set forecast $weather
or
set forecast [set weather]

.

The dollar sign instructs Tcl to place the contents of ’weather’ into the string ’forecast’. ’set forecast weather’ sets the contents of ’forecast’ to be ’weather’ instead of ’cloudy rainy cold’.

Brackets like those shown above may also include different Tcl commands. For instance, if you wanted to remove an item from a list, the command for that would be included in the brackets. This returns the value of the list without that item. To save this change to the same list or a different one, the appropriate command is included within the brackets as an argument to the set command.

Tcl also allows for the storage of information in arrays, however the elements of these arrays do not have to be integers. For instance, if I wanted to keep track of my income throughout the year, the array would be set in the following way:

set income(January) 100
set income(February) 100
set income(March) 40
.
.
.

Then if I wanted to store one element into the string 'cash' I would do the following:

set money $income(January)
or
set money [set income(January)]

.

C.2 Manipulating List Elements

Let us first make a list called 'letters' to use in this example:

set letters {alpha beta gamma}

.

The index of 'alpha' is 0, the index of 'beta' is 1, and the index of 'gamma' is 2. *lsearch* returns the index of a particular element within a list given that element:

set number [lsearch $letters beta]

. Hence, 'number' is 1. *lindex* returns the list element given the index:

set name [lindex $letters 1]

. Hence, 'name' is 'beta'.

One can also remove and/or replace elements from lists, and add new elements. *lreplace* allows you to remove an element and/or replace it with a new one. *lappend* lets you add as many elements as you want to the end of the list.

To delete 'gamma' from the list, the following will do:

```
set letters [lreplace $letters 2 2]
```

. 'letters' would now be {alpha beta}.

The second argument to *lreplace* is the starting index, and the third is the final index which will be affected.

To replace 'beta' and 'gamma' with 'delta' and 'epsilon', do this:

```
set letters [lreplace $letters 1 2 delta epsilon]
```

. 'letters' would be {alpha delta epsilon}.

To add 'delta' and 'epsilon' to the end of the list, use:

```
set letters [lappend letters delta epsilon]
```

. 'letters' would now be {alpha beta gamma delta epsilon}.

It is also fruitful to be able to loop over elements in a list and use them in different functions. *foreach* permits one to accomplish this task. Let's suppose we want to loop over each element in 'letters' and add the index of each element to a different list, say 'indexes'.

```
foreach el $letters {
        set id [lsearch $letters $el]
        set indexes [lappend indexes $id]
}
```

So if 'letters' is {alpha beta gamma}, 'indexes' will be {0 1 2}. What is basically happening is that each element undergoes a 'set el *element*', and $el returns this value when needed by functions inside this loop.

C.3 Tcl processes

A Tcl process is basically a subroutine. The process is called by giving the name of the process and including any additional arguments. If only one process is called, a semicolon is optional at the end of the line.

Let us suppose that we have a process called AddThreeNumbers, which takes 3 different numbers as arguments and stores the answer in the global variable 'result'. In the process that calls AddThreeNumbers, something resembling this code should appear:

```
set x 1
set y 2
set z 3
global result
AddThreeNumbers $x $y $z
```

.

The dollar signs are required to pass the values of these variables to the process AddThreeNumbers. Elsewhere in the code one may define AddThreeNumbers as this:

```
proc AddThreeNumbers {a b c} {
        global result
        set result [expr $a + $b + $c]
}
```

.

When AddThreeNumbers is done, any process that has declared 'result' as global will read its value as '6'. Note that the variables in the Tcl process do not have to be the same as those used by the process calling AddThreeNumbers. In fact, the variables x, y, and z could have been declared global in both processes. Then the current value of x, y, and z at the time the process is called would be used. If no arguments are used, {args} must replace {a b c} above.

An important and critical usage of processes in the interface is the association of a Tcl command with a button widget produced by Tk. For example, in the process GatemasterTriggers, the button that appears in the interface to apply the new choices in this menu is made in the following fashion:

button $fr4.apply -text "Apply To Database" -command "WriteToDatabase $List1;UpdateGMTriggerLists" -background maroon4 -foreground white

.

'button' is a Tk command that produces a button on the interface. Many other kinds of 'widgets' may be created with similar commands: entry, label, checkbutton, menu, and so on. The first argument to the button command is the window pathname for this button in the window hierarchy of the graphical interface. Hence '$fr4' returns most of this pathname. The '-text' option specifies what will appear written on the button. The name must be in double quotes if it is more than one word.

The '-command' option tells Tk what to do when this button is pressed. First, the process WriteToDatabase is called with the argument $List1, and then various lists are updated via UpdateGMTriggerLists. In order to pass variables from the process using the 'button' command to processes or functions associated with this button, the entire command string must be enclosed within double quotes. Using braces will shut everything else out from the command string.

D Communicating with the Database

There are two types of parameters that are found in the database. One is labelled *dis_int* and the other is called *dis_string*. Either type may be fetched from the database in the same way. For instance, the interface first reads in the list */filter/trigs* into the list *master_software_list* in the following manner:

set master_software_list [dis_fetch /filter/trigs]

If one were to write this list back to the database, it would look like

dis_string /filter/trigs $master_software_list

. However, at this time due to some bug in the routine *dis_fetch* or *dis_string*, one cannot read a *dis_string* parameter into the database, write an adjusted list back to the database, and read it once more. Elements in the strings stored int the database are surrounded by their own braces, and currently any manipulation of these strings strips the elements of their own braces, and *dis_fetch* does not recognize white space by itself as separating two elements.

Surprisingly, *dis_int* parameters(i.e. lists of integers) with their elements surrounded by braces do not share this problem. So as a result, the interface does not write any *dis_string* parameters to the database, although it has no problem in reading them.