smt_ped_tools User Guide

(for v00-01-00)

Simon J.H. Dean, University of Manchester
8th October 2002

Contents

1 Overview **2**

2 How to Build and Run **3**

3 Using the Browser **4**
3.1 Browser Layout 4
3.2 Making a ROOT File 4
3.3 Loading and Examining an SDAQ Run 5
3.3.1 Problematic HDI Listings 6
3.3.2 List All HDIs 7
3.3.3 List Crates in Run 7
3.3.4 List Barrels and Disks in Run 8
3.3.5 The Slideshow Feature 8
3.3.6 Other Histograms 8
3.4 Loading and Comparing a Reference Run 9
3.5 Committing Changes 11
3.6 Saving Graphics 11

4 For The Programmer **12**
4.1 ROOT Classes 12
4.1.1 TSdaqBrowser 13
4.1.2 ThresholdInputDialog 17
4.1.3 Rules For ROOT Classes 19
4.2 DØOM Classes 20
4.3 Non-Persistent C++ Classes 21
4.4 smt_ped_tools Executables 22
4.4.1 create_ntuple 22
4.4.2 rootFromRun 23
4.4.3 sdaqbrowser 23

1 Overview

smt_ped_tools allows the user to interact with the DØ online production database for the purposes of calibrating the Silicon Microstrip Tracker (SMT.)

The package makes use of pedestal values measured during specialised Secondary Data Acquisition (SDAQ) runs. These runs are performed between stores when the normal running conditions of the SMT can be reproduced while no proton-antiproton collisions are taking place. Instead of the Level 3 trigger system, the detector response is processed by dedicated CPUs that calculate the average signal during the run (the pedestal) and its associated RMS error for each channel in the silicon detector. These values are then committed to the online production database. SDAQ runs are taken roughly once per week.

Such calibration is necessary to calculate the thresholds that are used by the silicon detector during normal data-taking. The SMT contains approximately 800,000 channels but in order to optimise the available computing power online it is necessary to restrict the number of recorded channels to those that are involved in physics events. This is achieved by operating the silicon read-out in 'sparse' mode. There are 128 channels connected to every SVXII-E chip in the SMT and every chip stores one threshold value. The threshold corresponds to the number of ADC counts a channel must measure in an event before it is read out. The immediately neighbouring channels are usually also recorded for use in clustering algorithms.

Each SVX chip has a corresponding threshold value stored in the database. All of the chips are periodically re-initialised and configured with values from the database ('downloaded') and if the threshold in the database has changed then the value stored in the chip is updated. Conditions in the detector change with time and so it is important to make new threshold calculations regularly and commit them to the database. The threshold must take into account usual levels of electrical noise in the read-out chain and this is well modelled under the conditions of an SDAQ run.

smt_ped_tools provides methods for accessing the database and extracting the pedestal information for requested SDAQ runs. The information is then saved locally in the form of ROOT files. The package also includes a graphical front-end ('browser') that can load the ROOT files as well as directly linking to the online production database. Histogramming functions show the current behaviour of individual components in the SMT and the changes that occur over time. New threshold values can be calculated and committed to the database.

The package makes use of ROOT to provide graphical analysis classes and the DØ Object Model (DØOM) to provide a database connection. The code for ROOT file generation from the database and many of the DØOM classes were developed by Freya Blekman. Some of the ROOT classes are heavily based on code developed by Michelle Reschke.

2 How to Build and Run

smt_ped_tools must be built and run on the DØ online cluster. Log into an online node[1] and type the following commands to create a release directory ('>' represents the command prompt):

> setup D0RunII <release_version>
> newrel -t <release_version> <directory_name>

where <release_version> is the current online release and <directory_name> is any usable directory name. To install and build the package, type the following commands:

> setup d0cvs
> cd <directory_name>
> addpkg -h smt_ped_tools
> cd smt_ped_tools/scripts
> ./build

The 'build' script will install the relevant version of dØomORACLE and build both packages. This process will also run a setup script that will copy some necessary files into the user's home directory.

The smt_ped_tools build will produce three executables - rootFromRun, create_ntuple and sdaqbrowser. These are run by scripts in the smt_ped_tools/scripts directory to cut down on user input. The process of creating a ROOT file containing calibration data can either be called from the command line or accessed from within the graphical browser. The command- line method is called by typing

> ./makerootfile <run_number>

in the smt_ped_tools/scripts directory, where <run_number> is the run number of an SDAQ run. When this facility is accessed from within the browser, the user is presented with a list of run numbers that are currently available. To launch the graphical browser, type

[1]The online cluster should be accessed using the fssh command. 'fssh d0ol21' logs the user into an online Application Development node, for example.

> ./browse&

in the smt_ped_tools/scripts directory. The ampersand ('&') ensures the browser runs as a background process. The 'browse' script can also be copied to any other directory and run from there instead. The browser can not currently look any higher in the file structure than the directory in which it was run so this allows the user to pick a convenient directory to run from.

3 Using the Browser

3.1 Browser Layout

Figure 1 shows the layout of the graphical browser when started. The format borrows heavily from that of the ROOT 'TBrowser' and in fact this browser can be used in exactly the same way. The upper left window is used to browse the current directory structure while the lower left window shows the contents of highlighted directories.

If the user double-clicks on any ROOT file then it will appear in the 'ROOT Files' directory and from there, the user can explore the tree structure of the ROOT file. Any histograms are drawn in the 'Analysis' window. As a result of basing the browser on the ROOT system, the user can take full advantage of all graphical facilities provided by ROOT such as resizing axes and changing fill attributes. As a rough guide, use the left mouse button for 'drag-and-drop' features and use the right mouse button for listing available object functions. Please see [1] for more details.

3.2 Making a ROOT File

The browser has a lot of specialised functions that become available when an smt_ped_tools ROOT file is loaded. These files are characterised by the name format 'SmtSdaqRun_<run_number>.root' and are denoted by a slightly different icon (see Figure 1). To create a ROOT file of this type, click on the 'Make ROOT File' tab.

The browser queries the online production database for all available SDAQ run numbers and lists them on-screen. Use the slider on the right to look through the list and when the desired run number is found, select it using the left mouse button. To create the ROOT file, click on the 'Make ROOT File For Selected Run' button. This runs a script in the background so the program does not wait for the file to be created before proceeding. The ROOT file will be created in the <release_directory>/bin/Linux<details> directory, where <release_directory> is the release directory and <details> are Linux version and compiler details.

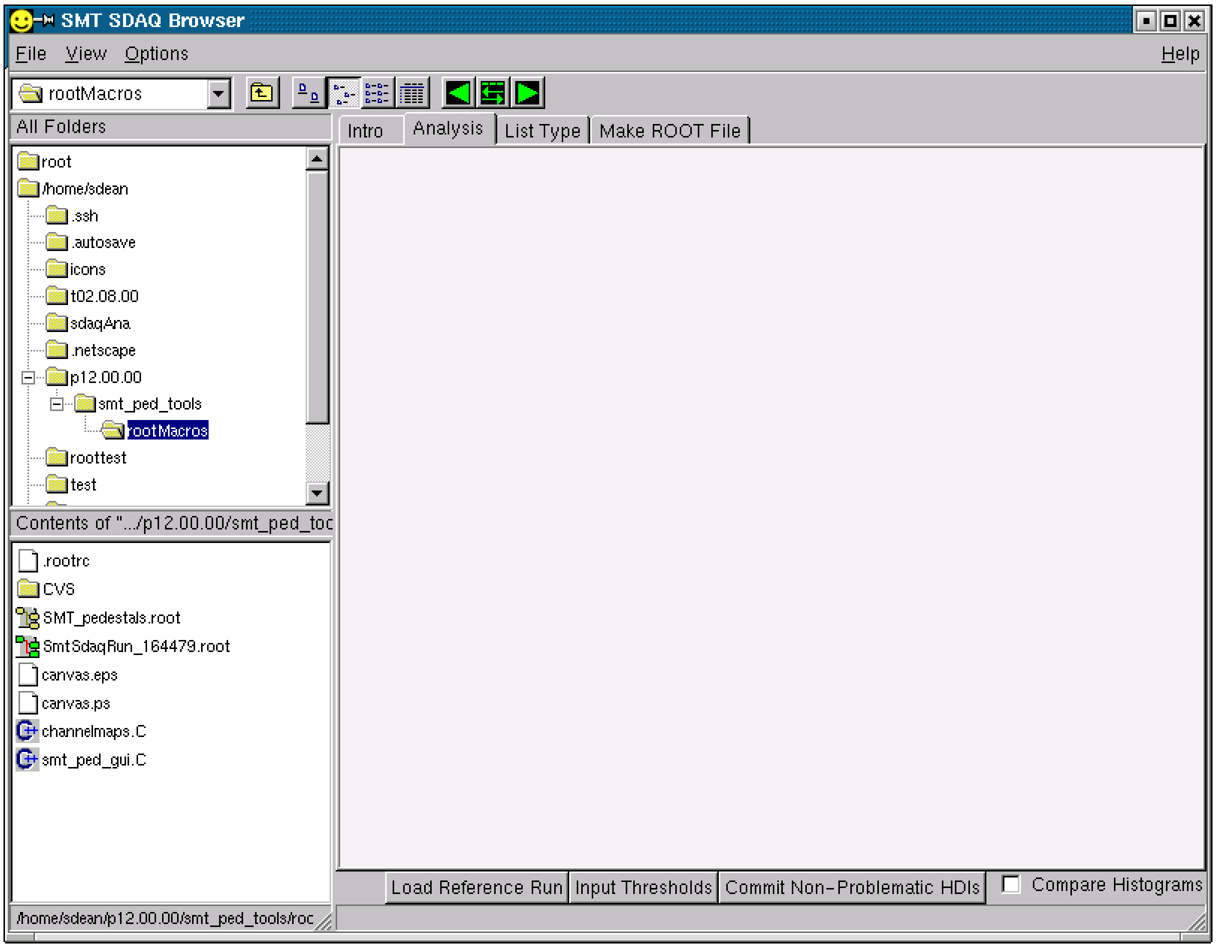

Figure 1: Original layout of the smt_ped_tools graphical browser

3.3 Loading and Examining an SDAQ Run

To load an SDAQ run, simply find it in the file browser and double-click on it.

The browser opens the ROOT file and fills object structures in memory that reflect the physical layout of the SMT. Channel data is grouped into SVX objects that correspond to the SVX chips in the detector. Depending on where the chip is in the SMT, it is grouped together as one of 3, 6, 8 or 9 chips that connect to one High Density Interconnect (HDI) so the browser groups them in the same way. Every HDI is listed with a 'serial number' consisting of a letter and three numbers (such as 'B2-2-2'.) The letter indicates whether the HDI is contained within a barrel ('B'), an F-disk ('F') or an H-disk ('H'). The numbers correspond to either

(barrel number) - (layer number) - (ladder number) for HDIs in barrels or

(disk number) - (view number) - (wedge number)

for HDIs in F-disks or H-disks. The barrel or disk number increases going from the north side to the south side of the SMT. Every barrel has eight numbered layers and every disk has two numbered views. The view number is 1 for p-side and 2 for n-side wedges. Please see [2] for a detailed description of the SMT geometry.

The browser can group the HDIs into various different types of list. These are accessed from the 'List Type' tab and are described in the following sections. The default listing is 'Problematic HDIs'

Probably the most systematic way to examine an SDAQ run is to list the HDIs of interest and to step through them using the slideshow facility. This procedure is described in section 3.3.5. There are other facilities that allow the user to inspect the calibration data in different ways and these are presented in section 3.3.6.

3.3.1 Problematic HDI Listings

After filling the HDI objects, the program attempts to establish whether or not each HDI can be considered 'problematic'. The browser connects to the database and extracts the threshold values currently used in the detector. Two different algorithms also calculate thresholds from the pedestal information in each SVX chip.

The 'Ped + 3σ' algorithm loops over all 128 channels and finds the mean pedestal and the mean of the pedestal errors from those channels that have a non-zero pedestal and have a pedestal error less than 6 ADC counts. The mean of the pedestal errors is referred to here as 'sigma' but it should not be confused with the standard deviation of the pedestals. The threshold is then found by multiplying sigma by three and adding the mean pedestal. In the special case that this number exceeds 240 ADC counts, the threshold is set equal to the mean pedestal instead.

The 'Ped + 6' algorithm fills two arrays with the pedestal values and the pedestal error values and sorts them so that they are both in ascending numerical order. The total number of 'bad' channels is found where a bad channel is defined as having an error larger than 6 ADC counts or a pedestal that is either zero or greater than 200 ADC counts. This number of channels is cut from the end of both arrays and the middle three-fifths of the remaining samples are classified as 'good' and used to calulate the mean pedestal value and sigma, the mean value of the pedestal errors. In the case that there are less than 5 good channels and more than 100 bad channels, these two averages are set equal to zero instead. The threshold is calculated by adding 6 ADC counts to the mean pedestal. If sigma is greater than 4 ADC counts or the calulated threshold is either below 10 ADC counts or above 230 ADC counts then the threshold is changed to 254 ADC counts (the maximum).

The program also counts the number of strips in each chip that fulfill certain criteria. 'Below threshold' strips have a pedestal value that is at least 25 ADC counts less than the 'Ped + 3σ' threshold. 'Noisy' strips have a pedestal error larger than 6 ADC counts.

Once all of these calculations have been performed for every chip in an HDI, the program tests to see whether it falls into any of the following categories.

Noisy A 'Noisy' HDI contains at least 4% noisy strips.

Below Threshold A 'Below Threshold' HDI contains at least 2% strips that are below threshold.

Big Threshold Shift The difference between the current database threshold and the 'Ped + 6' threshold is calculated for every chip in the HDI and if any of these differences exceeds 5 ADC counts or the average difference for all of the chips exceeds 3 ADC counts then the HDI is considered to have a 'Big Threshold Shift'.

Big Threshold Difference If the difference between the 'Ped + 3σ' threshold and the 'Ped + 6' threshold is greater than 5 ADC counts then the HDI is considered to have a 'Big Threshold Difference'.

Problematic Any of the above are true for this HDI

Non-problematic None of the above are true for this HDI

These six categories are provided as lists in the 'List Type' tab.

3.3.2 List All HDIs

This lists all HDIs included in the SDAQ run.

3.3.3 List Crates in Run

The HDIs in the run can be listed according to which VRB readout crate they are connected to. When this listing is selected, a list of crate icons is displayed in the list box corresponding to those crates that were included in the SDAQ run. Double-clicking on any crate icon will list all HDIs assigned to that crate.

3.3.4 List Barrels and Disks in Run

The HDIs in the run can also be listed according to their physical location in the detector. Selecting this list type produces a list of barrels and disks that were involved in the SDAQ run. Double-clicking on any of these icons will list all of the associated HDIs.

3.3.5 The Slideshow Feature

This facility allows the user to step through a selected list of HDIs and look at calibration histograms for each one. Progress through the list is controlled by the three slideshow buttons at the top of the browser ('Slideshow Previous','Slideshow Start' and 'Slideshow Next'.) To begin a slideshow, first display an HDI list in the list box by clicking on the relevant button in the 'List Type' tab. Click the 'Slideshow Start' button to display the calibration histograms for the first HDI in the list. Use the 'Slideshow Next' or 'Slideshow Previous' buttons to move to the next or the previous HDI in the list at any time. Use the 'Slideshow Start' button again to move back to the first HDI in the list.

An example of the calibration histograms displayed during a slideshow are shown in Figure 2 (overleaf). The two histograms show the pedestals (upper histogram) and the pedestal errors (lower histogram) measured in ADC counts plotted for each channel. The channels are arranged so that the first 128 channels correspond to strips 1 to 128 in the first chip, the next 128 channels correspond to strips 1 to 128 in the second chip and so on.

The pedestal histogram also shows the two calculated thresholds by plotting the 'Ped + 3σ' threshold in yellow and the 'Ped + 6' threshold in green.

3.3.6 Other Histograms

The calibration histograms described in the previous section can also be plotted by two other methods. Double-clicking on an HDI icon in the list box will produce the corresponding calibration plots and right-clicking on an HDI icon will display a 'context menu' that lists available object functions. Selecting 'createCalibHisto' will also draw the calibration histograms.

Selecting 'createPedestalHisto' from the context menu will plot a frequency histogram of pedestals while selecting 'createSigmaHisto' will plot a frequency histogram of pedestal errors.

'showHDIhistory' uses threshold data stored in the database to produce a 3-d plot of this HDI's threshold history. The x-axis shows time in units of years and has one entry

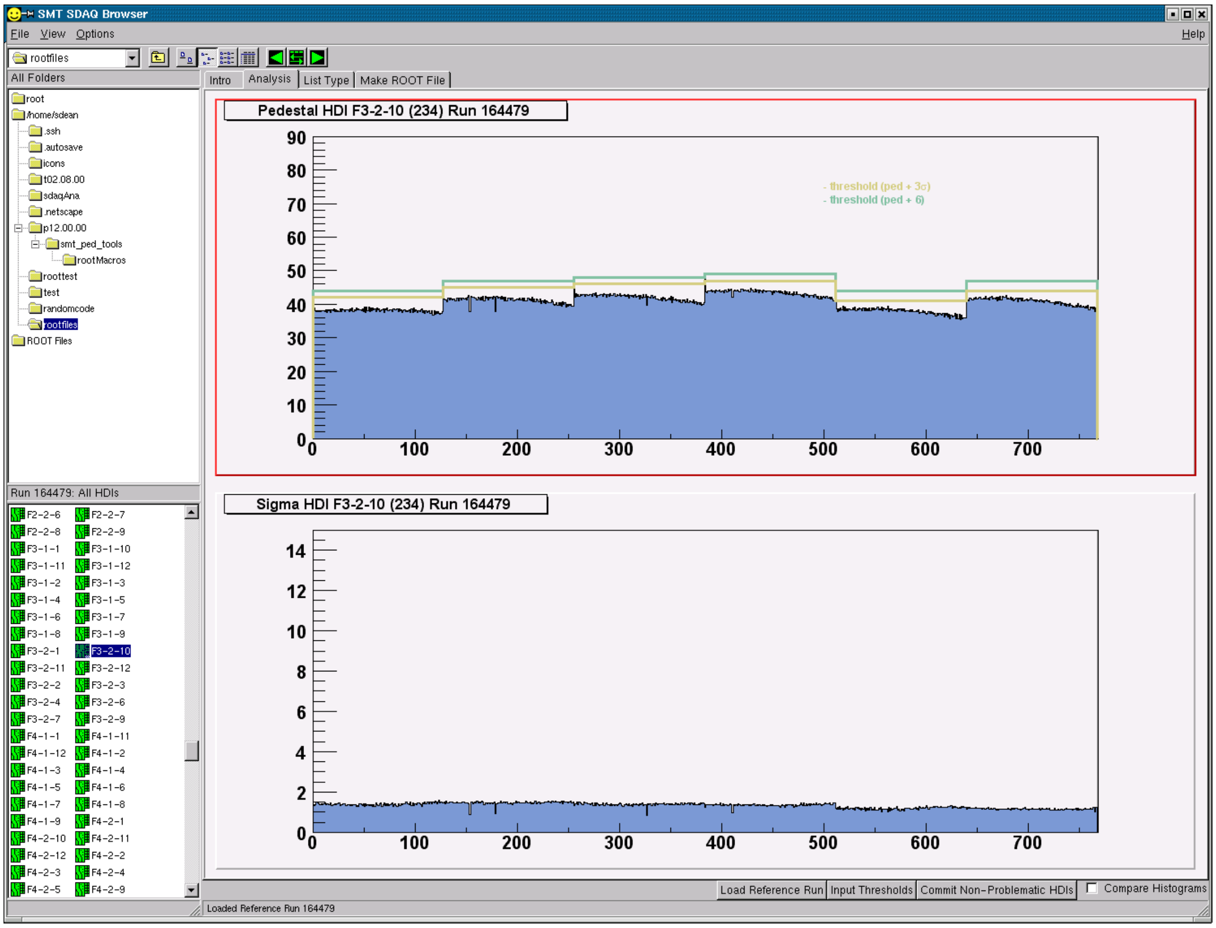

Figure 2: Calibration histograms for HDI F3-2-10

per 'day'[2] , the y-axis shows chip number and the z-axis shows the threshold. The time scale runs up to the present and has the 'Ped + 6' threshold plotted for the next ten days.

'showRecentHDIhistory' is similar to 'showHDIhistory' except that the x-axis represents SDAQ runs instead of pure time. The last ten committed threshold values are displayed for every chip and they are numbered 1 to 10 on the x-axis where 10 is the most recent. The eleventh bin is filled with the 'Ped + 6' calculated threshold.

3.4 Loading and Comparing a Reference Run

The user can load a second SDAQ run in order to make comparisons with the data in the current run. To load a reference run, click on the 'Load Reference Run' button

[2]The plotting algorithm allocates 31 days to every month so there are 372 days in a year

and find the desired ROOT file in the browser dialog that appears. Select the ROOT file and click on 'Open' to load the file.

Once a reference file is loaded, the calibration plots for an HDI can be presented in a slightly different way. If the 'Compare Histograms' check box is checked then six histograms will be plotted instead of two. These histograms are shown in Figure 3. The upper three histograms plot (from left to right) the pedestal from the reference run, the pedestal from the current run and the difference between these two pedestals against channel number. The lower three histograms present the pedestal error information in the same way. The calculated thresholds are displayed in both the reference and the current pedestal plots. The 'Ped + 3σ' threshold and 'Ped + 6' threshold are shown in brown and green respectively for the reference plot and yellow and green respectively for the current plot.

Figure 3: Calibration histograms for HDI B3-2-6 with a reference run loaded

3.5 Committing Changes

There are two ways to commit changes to the online production database. Clicking on the 'Commit Non-Problematic HDIs' button causes the program to run through the list of non- problematic HDIs and update each corresponding threshold stored in the database with the 'Ped + 6' threshold.

Clicking the 'Input Thresholds' button launches a dialog that shows threshold information for the last selected HDI and provides functions for editing and plotting these data and committing them to the database. The calibration histograms for the last selected HDI are redrawn and the thresholds currently stored in the database are plotted in red on the pedestal histogram(s). The 'Input Thresholds' dialog is illustrated in Figure 4.

The dialog displays rows of buttons showing the calculated thresholds from the reference run if one is loaded, the calculated thresholds from the current run and the database thresholds. There is one button for every SVX chip in the HDI along each row and the chip number is displayed at the top. Clicking on any button will enter the displayed value in the text box for that chip at the bottom of the dialog. Clicking on the 'edit' radio button at the end of a row will add all of the displayed values into the text boxes at the bottom. The text boxes originally contain the 'Ped + 6' values from the current run and these are automatically plotted in blue on the pedestal histogram(s).

The 'Plot' column indicates which values are currently drawn and each check button allows the user to remove and replot the values in that row.

Clicking the 'Commit' button saves the values in the text boxes into the database. The 'Close' button exits the dialog and the 'Next' and 'Previous' buttons exit and open another dialog for the next and previous HDIs in the list respectively. 'Commit + Next' does the same thing as clicking 'Commit' and then 'Next'.

It is possible to use the main browser while the 'Input Thresholds' dialog is open. If the 'Analysis' window is redrawn for any reason then the user can replot the calibration histograms for the HDI featured in the dialog by clicking the 'Replot' button. The program goes to the featured HDI's position in the list. If a new list has been displayed since the dialog was opened then clicking 'Replot' will also reselect the original list type.

3.6 Saving Graphics

It is possible to save the contents of the 'Analysis' pad to a PostScript file. Selecting 'Save' from the File menu will save the output as 'canvas.ps' in the current directory.

Selecting 'Save As' from the File menu brings up a dialog for choosing the filename before saving.

4 For The Programmer

This section provides a general reference for the code in smt_ped_tools. The classes are grouped into three categories - ROOT, DØOM and Non-Persistent C++ classes - and briefly described. There then follows a section with descriptions of the three smt_ped_tools executables.

4.1 ROOT Classes

ROOT provides useful data analysis and graphical user interface (GUI) classes which makes it ideal for supporting the smt_ped_tools browser. The following classes are all ROOT classes by inheriting directly or indirectly from the ROOT base class TObject. Direct inheritance will have been necessitated either because some of the ROOT GUI classes will only interact with other ROOT classes by requiring a TObject pointer or because the ROOT memory handling system occasionally has trouble dealing with a non-ROOT class.

HdiContainer This class forms the basis of the HDI lists used in the browser. HdiContainer contains an array of HDI pointers and has methods for manipulating this array.

SdaqRootTuple This is a base class that provides methods for accessing the information in ROOT files that were created in smt_ped_tools. The code was created using the TTree::MakeClass() method.

SmtSdaqRootTuple This class inherits from SdaqRootTuple. The member function Loop() creates HDI objects in free memory and fills them with the appropriate calibration data from a ROOT file. The Cratemap class (section 4.3) is also used to find the corresponding VRB crate number for each HDI. The HDIs are added to a list of pointers for later access.

TSdaqBrowser See section 4.1.1.

TSdaqGuiFactory This class inherits from TGuiFactory and handles the creation of ROOT GUIs in 'sdaqbrowser'. A TSdaqBrowser is spawned when a TBrowser is requested.

TSdaqIconBox This class provides methods for handling the graphical entries in the TSdaqBrowser list box (lower left in the browser GUI) each of which is stored as a TSdaqObjItem. The AddObjItem function accepts an object and creates a TSdaqObjItem, using the object's name to allocate an appropriate icon when displayed on-screen. The .root.mimes file which is saved to the user's home directory during setup is consulted to match a name to an icon.

TSdaqObjItem This class inherits from TGFileItem and has no methods of its own other than the constructor. It represents one entry in a TSdaqIconBox.

ThresholdInputDialog See section 4.1.2.

HDI The HDI class contains pedestal and pedestal error information specific to one HDI in the silicon detector. It contains an array of pointers to the SVX objects which represent the chips that are physically connected to this HDI. As well as methods for accessing the calibration data and the HDI identification data, there are various graphical methods and a method for committing all thresholds for the associated SVX chips to the database using the DatabasePortal. This last function either commits the 'Ped + 6' threshold values or a set of values supplied in the argument list. The graphical methods use the stored calibration data and database information to fill and display various histograms for this HDI.

Any functions declared in the header followed by the comment '// *MENU*' are included in the HDI context menu. This is a feature general to all ROOT objects.

The 'problematic' status of the HDI is stored using a series of bools (such as 'fNoisy') that are made accessible through member functions (such as 'IsNoisy()').

RefRun RefRun inherits from SDAQ and simply indicates that the data contained is intended for reference purposes.

SDAQ This class creates an SmtSdaqRootTuple. Once the SmtSdaqRootTuple has loaded a ROOT file and created the necessary HDIs, the SDAQ loops over the HDIs and creates the appropriate SVX objects for each one. The SVXs are filled and then the threshold calculation algorithms are invoked and the database thresholds retrieved before establishing the 'problematic' status for each HDI.

The SDAQ accesses all associated HDIs and SVXs via two arrays of HDI and SVX pointers. Member functions are provided for externally accessing these pointers. There are also graphical methods for filling and displaying histograms but these are not used.

4.1.1 TSdaqBrowser

The TSdaqBrowser class contains the code for the main browser window in 'sdaqbrowser'. All other windows and subwindows are called from this class.

The constructor calls the CreateBrowser() method which lays out all frames within the browser window including the menubar (a TGMenuBar), the upper left-hand file browser (a TGListTree viewed through a TGCanvas), the lower left-hand list box (a TSdaqIconBox viewed through a TGListView) and the main multi-tab area (a TGTab with four associated tab frames.) The 'Analysis' tab frame contains a TRootEmbeddedCanvas which is the pad used by TSdaqBrowser to display all histograms. ROOT class descriptions can be found at [4]. Various member variables and objects are also initialised in CreateBrowser().

One important concept that should be explained is the way in which 'events' are handled. An event is generated every time something happens on-screen, such as a mouse-click. Every graphical object must be able to handle events and the ROOT graphical classes accomplish this using the ProcessMessage() function. ProcessMessage() takes three Long_ts as arguments and translates them into an event using the GET_MSG() and GET_SUBMSG() functions. Every ROOT object that generates an event emits a signal in the form of these three Long_ts for later unpacking in ProcessMessage(). The layout should look something like the example shown below.

```
Bool_t <class_ name>::ProcessMessage(Long_t msg, Long_t parm1, Long_t parm2)
{
  switch (GET_MSG(msg)) {
  case kC_COMMAND:
       switch (GET_SUBMSG(msg)) {
       case kCM_CHECKBUTTON:
            switch(parm1){
            case 77:
              <button_ function>
              break;
            default:
              break;
            }
       default:
         break;
       }
  default:
    break;
  }
}
```

where <*class_ name*> is the name of the class. In this example <*class_ name*> would only have one ‘active’ member, a TGCheckButton which was defined with the button ID number 77. TGCheckButton is described in [4] although it is sufficient to know that the button ID number is taken as an argument in the TGCheckButton constructor. The class description also tells us that

> Selecting a check button will generate the event:
> kC_COMMAND, kCM_CHECKBUTTON, button id, user data.

The example shows that the GET_MSG() and GET_SUBMSG() functions take the first Long_t, msg, and translate it into a message and a sub-message. The message ‘kC_COMMAND’, sub-message ‘kCM_CHECKBUTTON’ and Long_t parm1 equalling ‘77’ therefore corresponds to a TGCheckButton with a button ID of 77. If this check button were selected, the function <*button_ function*> would be called. The Long_t parm2 was not needed in this example.

Every ROOT object that generates an event inherits from the class TGWidget, which has a member function ‘Associate()’. After the object is created, Associate() must be called with a TGWindow pointer passed as the argument. ProcessMessage() for the specified TGWindow will be called when the TGWidget generates an event. TSdaqBrowser is a TGWindow by indirect inheritance.

TSdaqBrowser contains a lot of TGWidgets such as buttons, menu items and list box entries. All of them are dealt with by code fragments or function calls in the appropriate part of TSdaqBrowser::ProcessMessage.

The ’List Type’ tab contains nine buttons. When any of these buttons are pressed, MakeListDecision() is called after some initial conditions have been set. MakeListDecision() is a function that ascertains what type of HDI or HdiContainer list needs to be created and runs the appropriate code. When the list has been created, the TSdaqIconBox is filled with the constituent objects and refreshed so that the new contents are displayed.

When the ‘Make ROOT File’ tab is clicked, MakeListOfRuns() is called. This function clears the list box in the ‘Make ROOT File’ tab frame and uses a DatabasePortal object to extract a list of available SDAQ run numbers from the database. This list of run numbers is added to the list box which is then refreshed to display the new contents. If the ‘Make ROOT File For Selected Run’ button is clicked then the browser executes a system command which runs a background script for running ‘rootFromRun’ with the selected run number as input.

If any of the slideshow buttons are clicked, the program tests to see whether an HDI list is currently displayed. If this is the case then the integer fSelectedHDInumber, used to indicate the position in the list of the last selected HDI, is modified accordingly and functions are called to plot the calibration histograms for the HDI in the modified list position.

If an object is selected in the file browser window then the function ListTreeHighlight() is called before redrawing the main browser window. ListTreeHighlight() uses member data from the selected object to set some bookkeeping variables before calling BrowseObj(). BrowseObj() clears the TSdaqIconBox and calls other functions for listing items appropriate to the selected object, such as directory contents.

If an object is double-clicked in the TSdaqIconBox then the object name is scanned to determine the appropriate action to take. If the selected object is an smt_ped_tools ROOT file then a new SDAQ is initialised and the MakeListDecision() function is called. The default list type is set in the TSdaqBrowser constructor as 'Problematic HDIs' and this list will be drawn unless the user has already selected a different list type. If the selected object is an HDI then the appropriate function will be called to draw calibration histograms. If the selected object is an HdiContainer then the MakeListOfHdis() function is called to list the contained HDI objects. In the case that the selected object does not correspond to any of the above options, DefaultIconBoxAction() is called to decide what action to take based on the object's ROOT-based attributes.

If an object is right-clicked in the TSdaqIconBox then the TBrowser function GetContextMenu() is used to call the TContextMenu function Popup(), which displays a context-sensitive menu relevant to the selected object.

If the 'Load Reference Run' button is clicked then a TGFileDialog file browser dialog is created for the user to locate and select an smt_ped_tools ROOT file. The resultant filename string is used to initialise a RefRun object and if an SDAQ object is already loaded then a THistoAnalysis object is also created.

If the 'Input Thresholds' button is clicked then the program checks to see whether fSelectedHDInumber is a sensible value, implying that an HDI has been selected, and if so InputThresholds() is called. InputThresholds() clears the TRootEmbeddedCanvas contained in the 'Analysis' frame and calls the appropriate function for drawing the calibration histograms for the selected HDI. A new ThresholdInputDialog is then created.

If the 'Commit Non-Problematic HDIs' button is clicked then CommitAllNonProblematics() is called. CommitAllNonProblematics() first checks whether an SDAQ object exists in memory and, if so, it displays a TGMsgBox message box to confirm whether the function should proceed by presenting 'OK' and 'Cancel' buttons. If 'Cancel' is clicked, the function exits. If 'OK' is clicked, the function loops over all HDIs in the loaded SDAQ and checks whether each one is problematic. If not then the HDI is instructed to commit its thresholds to the database.

The 'Compare Histograms' check box is used to change the value of a bool which is consulted every time calibration histograms are plotted to see whether reference plots should also be drawn.

A few functions are used by ThresholdInputDialog to access the current GUI configuration and make changes. PlotCalibrationHistos() is a function that accepts an HDI list position number and draws the calibration plots for the corresponding HDI. GetHdiListType() returns the HDI list ID number, GetSelectedHdiNumber() returns the list position of the current selected HDI and GetRunNumber() returns the current run number.

The function CallAnotherThresholdInputDialog() is called whenever a ThresholdInputDialog is closed to decide whether another one should be created.

4.1.2 ThresholdInputDialog

When a ThresholdInputDialog is launched, it uses a pointer to the main TSdaqBrowser to store locally some information about the current selected HDI, such as the position in the current list and what list is displayed. This information is useful in the event that the configuration in the main window changes.

The constructor populates the dialog window with graphical objects by explicitly adding some TGTextButtons and TGLabels and by calling AddValues() and AddTextEntries().

AddValues() deals with either three or five types of threshold depending on whether a reference run is being considered - 'Ped + 3 sigma (new)', 'Ped + 6 (new)', 'Threshold (database)' and (with reference run) 'Ped + 3 sigma (reference)' and 'Ped + 6 (reference)'. For each type of threshold, AddValues() loops over all SVX chips in the HDI being considered and creates a TGTextButton displaying the relevant threshold value. ThresholdInputDialog stores an array of threshold values and an array of TGTextButton pointers for each threshold type and these are also filled in the SVX loop. The ThresholdInputDialog uses a TGMatrixLayout layout manager to position the TGTextButtons in rows according to threshold type and columns according to chip number.

AddValues() also creates a TGCheckButton and a TGRadioButton at the end of each row.

AddTextEntries() creates a row of TGTextEntrys below the rows of TGTextButtons described above. These are initially filled with values from the array of 'Ped + 6 (new)' thresholds. ThresholdInputDialog stores an array of values corresponding to those displayed in the TGTextEntrys (the 'edited thresholds') and also an array of pointers to the TGTextEntrys.

After calling AddValues() and AddTextEntries(), the constructor sets up the initial configuration of the TGCheckButtons by setting them as 'checked.' These buttons are used for adding and removing lines from the calibration plots in the TSdaqBrowser 'Analysis' tab frame so the constructor also calls functions to plot any lines needed to match this original configuration.

As with TSdaqBrowser, ThresholdInputDialog has a ProcessMessage() function for handling events. If any of the TGTextButtons displaying thresholds are clicked then that threshold value is displayed in the TGTextEntry for the corresponding chip. If the TGRadioButton at the end of a row is clicked then a function is called that transfers all of the threshold values in that row into the TGTextEntrys. Whenever a TGTextEntry is updated, the new number is also saved to the corresponding array of values. If one of the TGCheckButtons is clicked then a function is called which either plots or unplots the associated threshold values on the calibration histograms.

If the 'Replot' button is clicked then Replot() is called. Replot() compares the local HDI information that was saved when the ThresholdInputDialog was created to the current HDI information in the TSdaqBrowser window. if the run numbers are different, then Exit() is called and the dialog closes. If the run number is still the same but the list type has changed then the function restores the original ThresholdInputDialog button configuration, calls TSdaqBrowser::MakeListDecision() to display the original HDI list, plots the calibration histograms for the featured HDI and draws the database and edited thresholds. If the list type has not changed then the same process is performed without calling TSdaqBrowser::MakeListDecision().

If the 'Commit' button is clicked then the array of edited thresholds is passed to the HDI function commitThresholdsToDatabase(). If the 'Next' button is clicked then the bool 'fShowNextInputDialog', which is shared between the TSdaqBrowser and the ThresholdInputDialog and has a default value of false, is set to true and Exit() is called. Similarly, if the 'Previous' button is clicked then the shared bool 'fShowPreviousInputDialog' with default value false is set to true and then Exit() is called.

The Exit() function calls TSdaqBrowser::CallAnotherThresholdInputDialog() which uses the bools fShowNextInputDialog and fShowPreviousInputDialog to decide whether or not to launch another ThresholdInputDialog.

4.1.3 Rules For ROOT Classes

There are a few rules for classes that are added to the ROOT system. All class definitions must have as the very last line before the final '};' a line matching the description

```
ClassDef(<class_ name>,<class_ version>)
```

where <class_ name> is the name of the class and <class_ version> is an integer version number. The implementation file must contain a line matching the description

```
ClassImp(<class_ name>)
```

Every ROOT class must have a default constructor (ie.. either has no arguments or all arguments have been assigned default values). These requirements make sure that each class is taken into account when the rootcint preprocessor creates default functions for class persistence and a CINT dictionary. The dictionary makes the class accessible from the CINT interpreter.

In turn, to run the rootcint preprocessor within the DØframework the line

```
include SRT_D0/arch_spec_root.mk
```

must be included before

```
include SRT_D0/standard.mk
```

in the source code directory GNUmakefile. Every new ROOT class needs to be named in a 'linkdef' file so that rootcint will develop appropriate linkage code in the dictionary. Each linkdef file should have the name format *<file_ name>_linkdef.h* where *<file_ name>* is the name of an implementation file and header file pair containing the code for the relevant classes. This file should also be included in the source code directory. In smt_ped_tools, all of the new ROOT classes are included in *sdaqbrowser.cpp* and *sdaqbrowser.hpp* so the linkdef file is called *sdaqbrowser_linkdef.h*. The layout of the linkdef file should look something like the example shown below

```
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;

#pragma link C++ class <class_ name>+;

#endif
```

Please see [3] for more information about creating linkdef files in ROOT.

4.2 DØOM Classes

The DØObject Model (DØOM)[5] is an object persistence system for use in C++ programs in the DØframework and dØomORACLE is a package that provides an interface between DØOM and ORACLE databases. smt_ped_tools uses dØomORACLE to access the online production database. The following classes are DØOM classes because they inherit either directly or indirectly from the DØOM base class, d0_Object. Python dictionaries, created using the 'oracleParser.py' script, are used to map these classes to tables and variables in the database[7]. Each column in a table corresponds to one data member of a class and each row, or entry, in the table corresponds to one instance of the class.

CalibrationRuns Corresponds to the database table CALIBRATION_RUNS which contains run information from calibration runs.

CalibrationTypes Corresponds to the database table CALIBRATION_TYPES which maps the calibration run type to an id number.

Channels Corresponds to the database table CHANNELS which groups silicon channel id number according to SVX chip.

ComparisonSets Corresponds to the database table COMPARISON_SETS which assigns an ID number to a pair of pedestal-specific and gain-specific calibration runs. This table has not been filled.

Gains Corresponds to the database table GAINS which contains channel- level data from gain-specific calibration runs. This is no longer filled.

Pedestals Corresponds to the database table PEDESTALS which contains channel-level data from pedestal-specific calibration runs.

Statuses Corresponds to the database table STATUSES which contains the dead and noisy status of each silicon channel. This has not been filled.

SmtDaqSvxThreshold Corresponds to the database table SMT_DAQ_SVX_THRESHOLD which contains a history of all chip thresholds that have been committed to the database.

SpecialString Corresponds to the database variable DUAL.SYSDATE which contains the current time and date.

RunNumber Contains the variables RUN_NUMBER and START_TIME from the database table RUNS.RUNS.

SmtDaqHdi Corresponds to the database table SMT_DAQ_HDI which contains information specific to every HDI.

SmtDaqSvx Corresponds to the database table SMT_DAQ_SVX which contains information specific to every SVX.

SmtDaqSvxSimon Corresponds to the database table SMT_DAQ_SVX_SIMON which is a copy of SMT_DAQ_SVX that is used during code development.

SmtDaqSvxRun Corresponds to the database table SMT_DAQ_SVX_RUN which contains run-summary information for each SVX chip.

SmtDaqSimon Corresponds to the database table SMT_DAQ_SIMON which is a copy of SMT_DAQ_SVX_THRESHOLD that is used during code development.

dØomORACLE works by opening a stream to the database[6]. Selection criteria are submitted to the stream and a C++ standard library *list* is returned filled with references to d0_Objects that correspond to table entries matching the criteria. The dØomORACLE methods are made transparent in the 'sdaqbrowser' program by wrapping them in the class DatabasePortal. Every other class places function calls to DatabasePortal when they need to access the database. DatabasePortal does not inherit from d0_Object so it is described in section 4.3.

The specifics of DØOM classes are well documented in [5], [7] and [6] although it is worth mentioning here that all DØOM classes should be listed in a file called D0OM_COMPONENTS in the source code directory.

4.3 Non-Persistent C++ Classes

These classes do not inherit from ROOT's TObject or DØOM's d0_Object.

DatabasePortal DatabasePortal contains all of the dØomORACLE code used in the 'sdaqbrowser' program. The constructor initialises the DØOM system and creates an input and output stream to the online production database. Almost all of the methods involve querying the database for a list of one or more of the objects listed in section 4.2, looping over them and then returning some of the data they contain.

CommitSvxChip() and CommitHdi() are particularly interesting because after retrieving objects from the database, they make modifications and use the d0StreamDB::write() function to write the objects back into the database. This has the effect of committing changes to existing entries in the database, rather than adding new ones, because every object has at least one unique key[7] that identifies an exact table entry. The object must be passed into the database using the same d0StreamDB that was used to extract it.

Cratemap Cratemap provides a mapping between the name of an HDI and the corresponding VRB crate name. It achieves this by reading the information in from the file .cratemap, which is saved in the user's home directory during setup, and creating a C++ standard library *map*. The function GetCrateName() accepts an HDI name and returns a VRB crate name.

THistoAnalysis THistoAnalysis is used for comparing two SDAQ runs. The constructor accepts an SDAQ object and a RefRun object and calls the member function reduceRefRun(). This function creates a new SDAQ object and fills it with the HDIs from the RefRun that also appear in the original SDAQ object. This process of reducing the reference run saves memory by discarding HDIs that would not feature in the comparison anyway.

The method CanCompare() accepts an HDI name and returns true if that HDI exists in both SDAQ objects or false if not. The remaining methods of THistoAnalysis are graphical methods that compare the data from the two SDAQ objects. drawCalibrationHistos() is the function that produces the six reference calibration plots mentioned in section 3.4. The rest of the graphical methods are not used.

SVX The SVX class contains pedestal and pedestal error data specific to one SVX chip in the silicon detector. As well as methods for accessing the calibration data and the chip identification data there are methods for calculating and accessing the two previously described thresholds 'Ped + 3 σ ' and 'Ped + 6'.

4.4 smt_ped_tools Executables

smt_ped_tools builds three executables as follows.

4.4.1 create_ntuple

create_ntuple prompts the user for a desired number of runs to enter into a ROOT file. The DØOM system is then initialised before creating a dØomORACLE database stream and fetching a list of CalibrationRuns objects from the database corresponding to all available SMT calibration runs. A new ROOT file is created using the HepTuple package. The program then begins a loop over the calibration runs, starting with the most recent and working backwards until reaching the requesting total number of runs.

For every run number, create_ntuple queries the database for a list containing all available Pedestals objects in that run. This list is then looped over and the pedestal information for each channel is unpacked and saved into the ROOT file.

When the program finishes running through these loops, the ROOT file is saved and closed and the database stream is closed and deleted.

The pedestal data for one run takes up quite a lot of memory so it is recommended to use rootFromRun to create ROOT files one run at a time rather than using create_ntuple for accessing more than one run and using up vast amounts of memory on the online cluster.

4.4.2 rootFromRun

rootFromRun is similar in nature to create_ntuple. A run number is taken as the main argument and then the program initialises the DØOM system as well as creating a dØomORACLE database stream. The HepTuple package is used to create a new ROOT file and the program submits database queries to fetch two lists containing all Pedestals and Gains objects corresponding to the relevant run.

The program then loops over all channels, unpacking the associated Pedestals and Gains object for each one and saving the information to the ROOT file. Some diagnostic output is generated for the first ten channels and some summary information is outputted after the loop has finished. The ROOT file is saved and closed and the database stream is closed and deleted.

4.4.3 sdaqbrowser

sdaqbrowser simply runs a ROOT TApplication in which a TSdaqBrowser is created.

References

[1] Brun, Rademakers, Panacek, Buskulic, Adamczewski and Hemberger, *The ROOT User's Guide 3.02c,*
ftp://root.cern.ch/root/Users_Guide_3_2c.pdf

[2] DØ Collaboration, *"The DØ upgrade. The detector and its physics,"* Fermilab Pub-96/357-E, 1996

[3] *Using ROOTCINT,*
http://root.cern.ch/root/RootCintMan.html

[4] *ROOT Class Categories,*
http://root.cern.ch/root/Categories.html

[5] H. Greenlee, J. Hobbs, S. Snyder and V. White, *DØOM User Guide,*
http://www-d0.fnal.gov//d0dist/dist/releases/development/d0om/doc/d0om_user_guide.ps

[6] E. Barberis, H. Greenlee, M. Vranicar, S. White and V. White, *Database Access Using DØOM,*
http://www-d0.fnal.gov//d0dist/dist/releases/development/d0omORACLE/doc/database_access.ps

[7] E. Barberis, H. Greenlee, M. Vranicar, S. White and V. White, *Python C++ to Database Mapping Dictionary,*
http://www-d0.fnal.gov//d0dist/dist/releases/development/d0omORACLE/doc/database_dictionary

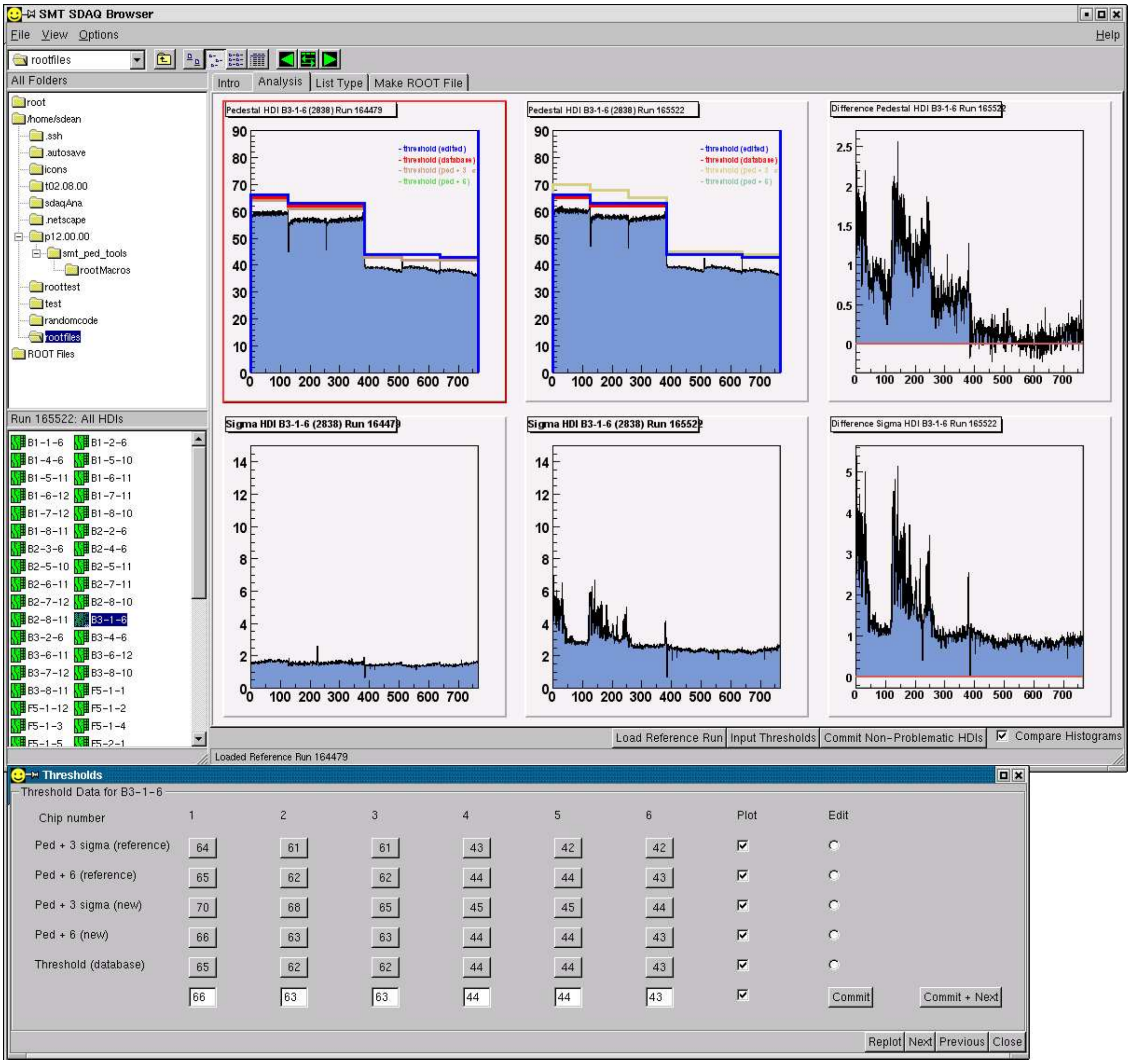

Figure 4: The 'Input Thresholds' dialog