DAQ Control Program for AWAKE BPM

Created: Apr. 24, 2017

Author: Andy Wong

This document serves as a detailed synopsis of the DAQ Control Program (DAQCP) for the AWAKE BPM. It describes instructions for using the program, as well as the underlying motivation and design.

Instructions

Installing Dependencies for the DAQCP

The DAQCP requires Matplotlib (version >= 0.99.1.1) and wxPython (version >= 2.8.12.0 (gtk2-unicode))

The computer that the DAQCP will run on already has all the required dependencies. If needed, simply run the following commands on a clean Scientific Linux 6.7 machine to install the dependencies.

Matplotlib

- sudo yum install python-matplotlib-tk

wxPython

- su -c 'rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm'
- yum install wxPython

Python (version >= 2.6) is also required, but should already be available.

Running the DAQCP

1. In a terminal, change to the project directory
2. Launch the program by entering 'python awake.py'
3. The program will now be running inside that terminal; closing the terminal will also terminate the program

Using the DAQCP

The DAQCP is primarily controlled via the terminal/console. Upon launching the script, you (the user) will be prompted to enter the last portion of the IP address of the connected BPM (e.g. if

the IP address is 192.168.13.**10**, then 10 should be entered. The program then waits for your commands. An example console output is shown below.

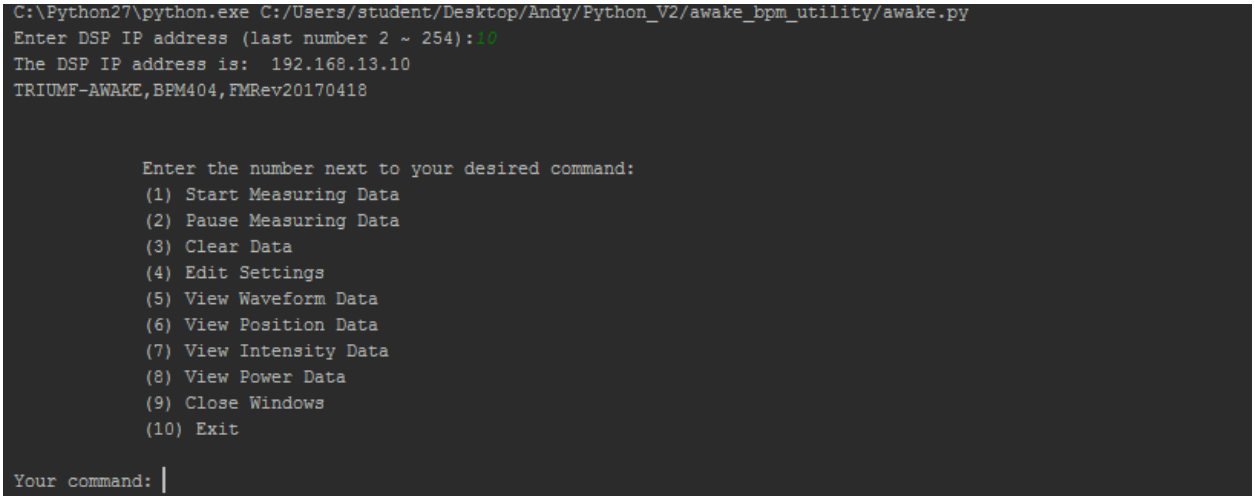

Fig. 1 - Example Console Output After Entering BPM IP Address

Note: if a long timeout error traceback is printed to the console after entering the last portion of the IP address, then the program most likely did not successfully connect to the BPM. In the event this occurs, double check the connection (e.g. by pinging the IP address in a terminal) and then restart the program.

As indicated in Fig. 1, there are 10 available user commands, each of which can be executed by entering the number to the left of the desired command into the console. Below is a description of each command.

Command	**Description**
Start Measuring Data	Changes the operation mode of the program to RUNNING, which means data will be collected from the FPGA as long as there is data available (note: this is different from enabling “Run” in the settings!).
Pause Measuring Data	Changes the operation mode of the program to PAUSED, which pauses the collection of data from the FPGA. The operation mode is already set to PAUSED when the program starts.
Clear Data	Wipes all collected data and resets the internal time counter to 0.
Edit Settings	Opens the Control GUI in a new window, which allows the user to modify and view the settings

View Waveform Data	Displays incoming waveform data (for all four channels) in a plot window. If a plot window was already active prior to executing this command, then the plot simply changes to the waveform data.
View Position Data	Displays all collected position and res. RMS data so far in a plot window. If a plot window was already active prior to executing this command, then the plot simply changes to the position and res. RMS data. If the operation mode is set to RUNNING and the FPGA is providing data, then the plot will automatically update.
View Intensity Data	Displays all collected intensity (S) data so far in a plot window. If a plot window was already active prior to executing this command, then the plot simply changes to the intensity. If the operation mode is set to RUNNING and the FPGA is providing data, then the plot will automatically update.
View Power Data	Displays all collected power data so far (for all four channels) in a plot window. If a plot window was already active prior to executing this command, then the plot simply changes to the power data. If the operation mode is set to RUNNING and the FPGA is providing data, then the plot will automatically update.
Close Windows	Closes the GUI and any active plots. Note: to close windows individually, simply click on the 'X' button at the top of the desired window to close.
Exit	Exits the program. Any active windows (Plots, GUI) are closed. All collected data is lost.

Viewing Plots

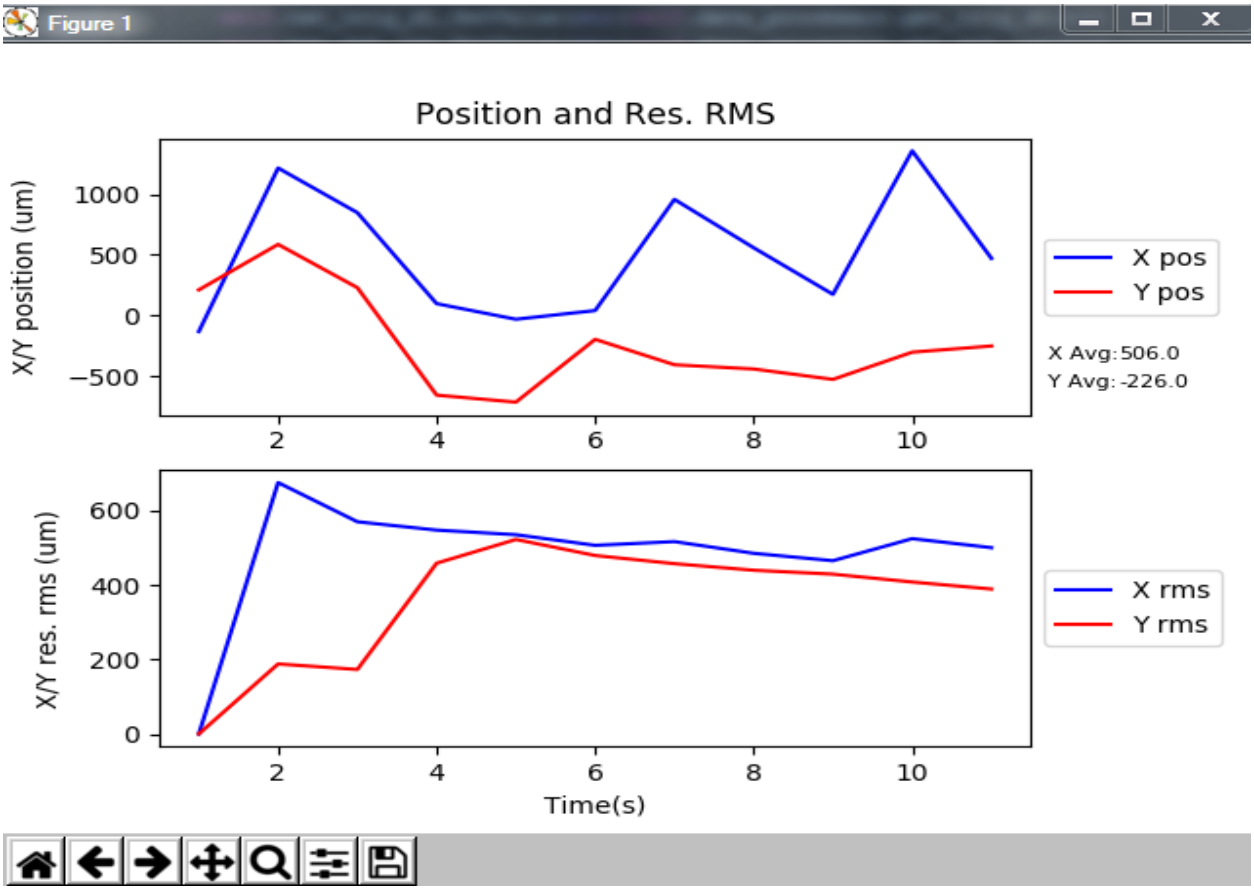

Fig. 2 – Position and Res. RMS Data Plot

Plots can be viewed by entering any one of the 'View ___' commands (Commands #5-8). They include a number of built-in features, such as zooming into the plots for a closer view, panning around the axes, and saving a screenshot of the plots.

Note: By default, all collected data will be displayed on the plots, with no panning required (the x and y axis will automatically adjust themselves as data is collected). However, if the user zooms into a plot and/or pans around, then the axis adjustments will stop and the user will need to manually pan around to examine the latest data. To return to the default functionality, simply close and reopen the plot window.

Using the GUI

Using the GUI is very similar to editing the settings in the LabVIEW program. A setting can be changed by simply entering the desired value into (or checking/unchecking) the box next to the

corresponding label, and then clicking the appropriate 'Update' button. If successful, all values associated with that 'Update' button (i.e. those in the same 'box' as the button) are written to the FPGA and a dialog appears indicating that the action was successful (the 'Update' button for 'Status' will *read* from the FPGA).

For settings that allow the user to type in their input, there are certain restrictions to the types of inputs that the user can provide, depending on the setting. Below is a list of these restrictions.

Panel	Restrictions
AddressPanel	MAC Address • Only hexadecimal digits (0-9, a-f/A-F) • Maximum of 2 digits per box • Final entered value will be interpreted as a hex value (e.g. 26 will be interpreted as 0x26 or 38 in decimal) • Values less than 16 in decimal can entered with an optional appending 0 (e.g. 5 can be entered as '5' or '05') IP Address • Only numerical digits (0-9) • Maximum of 3 digits per box • Final entered value of each box must be an integer from 0-255
EventParamPanel	All Settings: • Only numerical digits • Final entered value must be non-negative
ChGainPanel	All Settings: • Only numerical digits and a single period (.) for floating point values • Final entered value must be greater than -5.0 and less than 5.0
CalGainPanel	All Settings: • Only numerical digits and a single period (.) for floating point values • Final entered value must be greater than -5.0 and less than 5.0
OtherParamPanel	All Settings: • Only numerical digits • Final entered value must be non-negative

Note that if a setting is not given a value (i.e. the text box is empty) or is given a value that does not meet the appropriate restrictions described above, then any attempts to update other settings in the associated panel will fail.

Managing the Flash

There are some important things to keep in mind when reading from or writing to the flash memory in the FPGA.

Writing to Flash

When the corresponding button is clicked in the GUI, all values currently present in the FPGA registers are copied to the MicroBlaze flash buffer, and then copied from the flash buffer to the flash memory. Thus, in order to write a specific setting to the flash memory, it must first be loaded to the appropriate FPGA register (by clicking the relevant 'Update' button), and then written to flash memory by clicking 'Write to Flash'.

Reading from Flash

When the corresponding button is clicked in the GUI, data from the flash memory is copied to the flash buffer, and then copied from the flash buffer to the appropriate FPGA registers.

Note: at the time of writing this document, the flash memory does not hold any data regarding the Mode Register, which means read/write operations will not involve the Mode Register.

Updating the Addresses

In order to properly update the MAC and IP address of the BPM, the following steps must be done:

1. Enter the desired addresses, and then click the associated 'Update' button to write the addresses to flash buffer.
2. Click 'Write to Flash' to copy data from the flash buffer to flash memory
3. Reset the BPM
4. The BPM should now be configured with the new addresses

Motivation

The DAQCP serves as a replacement for the LabVIEW program used to interface with the AWAKE BPM. While the LabVIEW program provides the same functions as the DAQCP and is easier to develop/make changes to, it does come with some caveats:

1. A license is required to use the program
2. The program is more resource intensive and may not perform ideally when used to interface with a BPM remotely

The aim of the DAQCP is to accomplish the same tasks as the LabVIEW program, but be free to use at all times and perform better in remote operations.

Design

The DAQCP's design consists of the following major components:

Console/Terminal

This is where the user types in desired commands to control the flow of the DAQCP. All commands are entered as numbers ranging from 1 to 10 that each correspond to a specific action. Once a command is entered, the console will immediately prompt the user for their next command (until the user enters the exit command).

Plot Window

This is where plots are displayed. The plot window is capable of displaying waveform, position and res. RMS, intensity, and power data, though only one plot window can be active at a given time. By default, the plots will update automatically as new data is collected. Note that collected data will continue being updated regardless of whether the plot window is open or not.

Control GUI

This is where the user can modify and view the current BPM settings, in addition to accessing the FPGA's flash memory. The GUI can be opened or closed at any time without affecting the collection of data.

<u>Development Decisions</u>

Python

This programming language was selected for development on the basis of simple syntax (making it easier to read and write programs in the language) and availability of related libraries (described below).

Matplotlib

Matplotlib is a popular Python plotting library that offers a variety of ways to create and edit plots with relatively few lines of code. It has an extensive API with decent documentation. Its capabilities proved very suitable for displaying plots for the DAQCP.

wxPython

wxPython is a Python GUI toolkit that allows programmers to easily create robust graphical user interfaces. This particular toolkit was selected for creating the Control GUI due to the availability of excellent documentation and demos/example code.

System Architecture

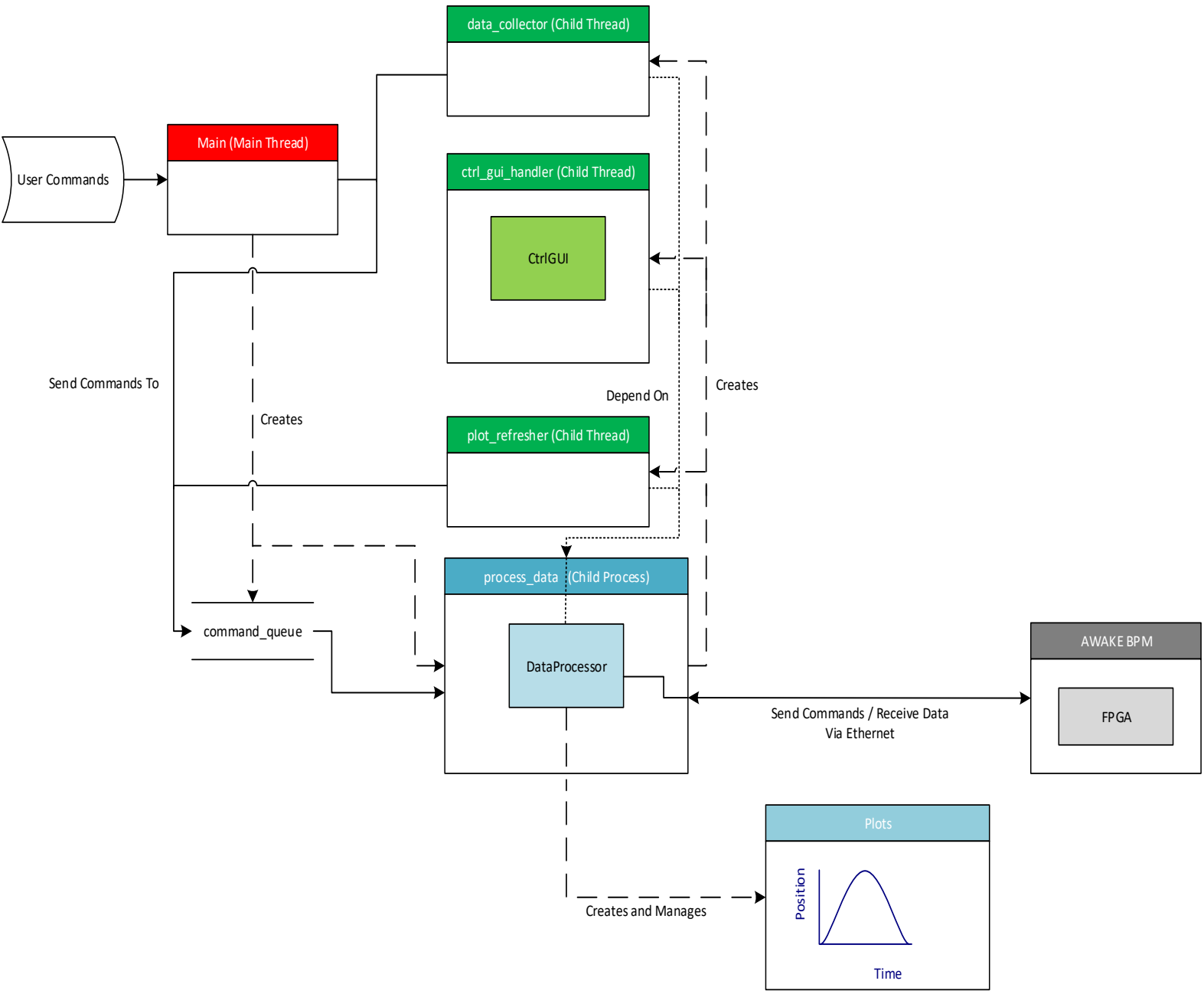

Fig. 3 – System Diagram of the DAQCP

System Overview

On start-up, the system enters the main function of awake.py, which runs in a single thread called the main thread. After retrieving the IP address of the BPM, it creates a thread/process-safe blocking queue called *command_queue* and a child process referred to as *process_data*. All external commands (from the user) and internal commands (from child threads, described below) are added to this queue and eventually handled by *process_data*. Once the queue and child process have been created, the main thread begins to continuously ask the user for commands, up until the user sends an exit command.

When created, *process_data* begins by constructing a *DataProcessor* object. This object is responsible for managing all plots and communicating with the BPM via Ethernet; it accomplishes these tasks via its methods, which are executed based on commands retrieved from *command_queue*. The child process continuously waits for new commands from *command_queue*, up until an exit command is received (behaving similarly to the main thread).

Depending on user commands and the current state of *DataProcessor*, *process_data* can spawn any one of three child threads (note: all three can be alive at once, but no two threads that are alive can be of the same type):

1. *data_collector*

   This thread is spawned when the user sends the command to start collecting data from the BPM. While alive, it continuously reads in new data from the FPGA and updates any active plots (via internal commands to update the plots), following a similar algorithm to the LabVIEW program.

   The thread is killed when the user sends a command to pause data collection.

2. *ctrl_gui_handler*

   This thread simply creates and runs the GUI.
   The thread is killed when either the following occurs:
   - the user clicks on the 'X' button to close the GUI window individually
   - the user sends the command to close all active windows

3. *plot_refresher*

   This thread is spawned any time the user attempts to view a plot, but data is currently not being collected. The thread merely "refreshes" the plot (via an internal command to

refresh the plot), allowing the user to interact with the plot window (note: it does nothing if no plot is active).

The thread is killed when data starts being collected.

Due to the concurrent nature of the system, several locks are used to prevent race conditions. These locks are all associated with *DataProcessor*, since the object is shared between multiple threads.

The three locks are as follows:

1. eth_lock: this lock is acquired by any method that needs to conduct Ethernet communication. Without this lock, multiple threads attempting to communicate via Ethernet at the same time will result in strange, inconsistent crashes.
2. data_lock: this lock is acquired by any method that needs to read from or write to the data buffers (i.e. the buffers that hold data collected from the BPM). This lock is mainly used by the plot-related methods. Without this lock, it is possible for the data buffers to become 'out of sync' when plot window needs to update, such that (for example) the number of collected X position values differ from the number of collected Y position values. This will cause the plot window to crash abruptly.
3. gui_lock: this lock is used by threads that are concerned with the current state of the GUI (i.e. is it open or closed). This was introduced due to the inclusion of worker threads that update the FIFO occupancy and event #, which need to terminate properly when the GUI is closed.

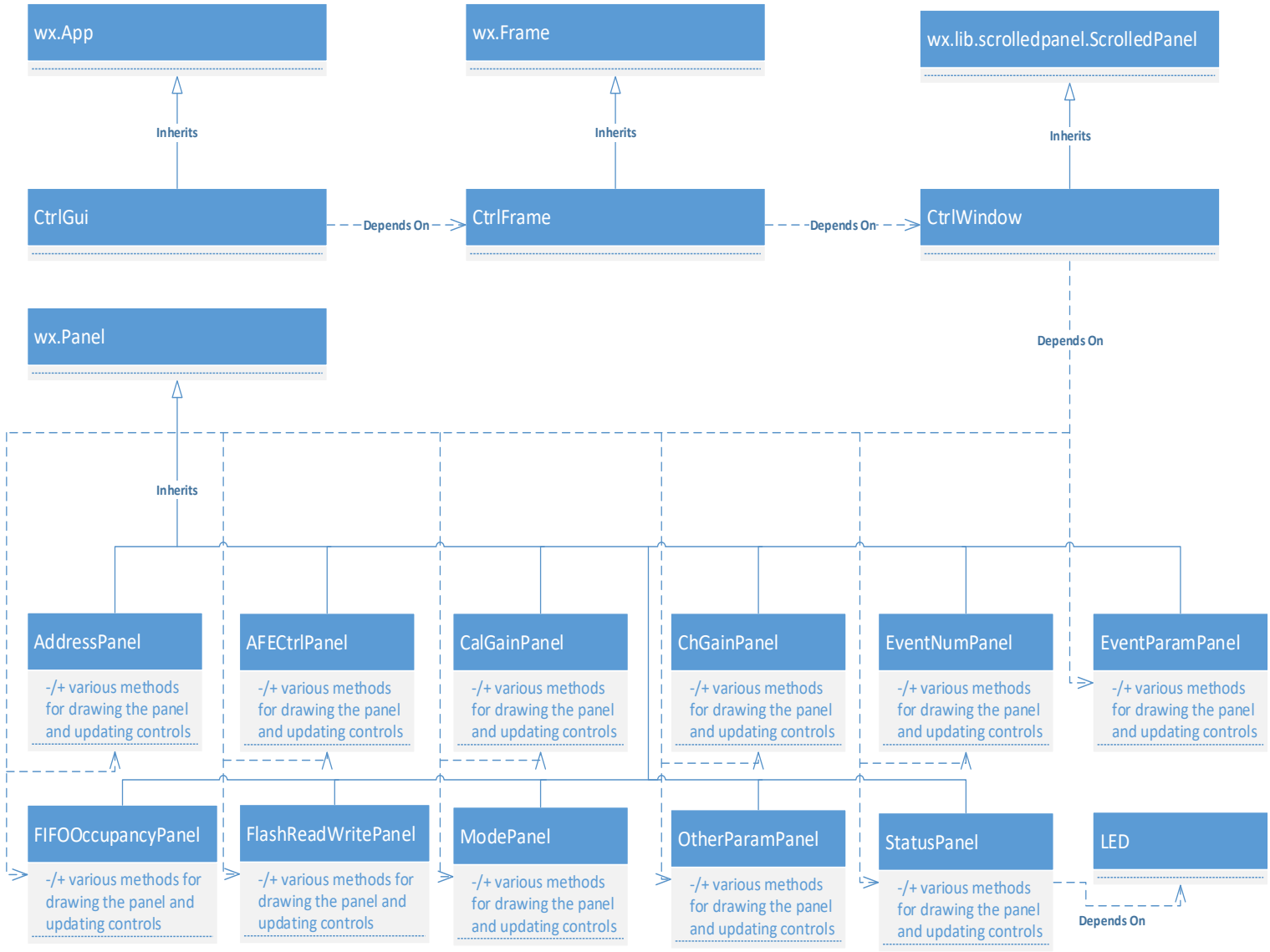

Fig. 4 – GUI Class Diagram for the DAQCP Control GUI

Note: The Validator class and its subclasses as well as the methods for each class shown above have been omitted for clarity.

GUI Overview

As described in the System Overview, the GUI is created every time the *ctrl_gui_handler* thread spawns, which occurs whenever the user sends the command to edit settings. Because the GUI is built using wxPython, the top-level app must either be an instance of *wx.App* (a built-in class of the wxPython library), or an instance of a class that inherits from *wx.App*. The latter option is more suitable (the subclass being called *CtrlGUI)* due to the complexity of the overall system.

During the initialization of *CtrlGUI*, an instance of *CtrlFrame* is created. This object is essentially the top-level window of the GUI (i.e. the box which the user can move around, minimize, maximize, or close). Inside *CtrlFrame* is a single instance of *CtrlWindow*, which is a special type

of Window in wxPython that is capable of scrolling when the frame is too small to show all of its contents. The *CtrlWindow* instance houses all of the various panels (see above) relevant to controlling and viewing the BPM settings.

GUI Layout

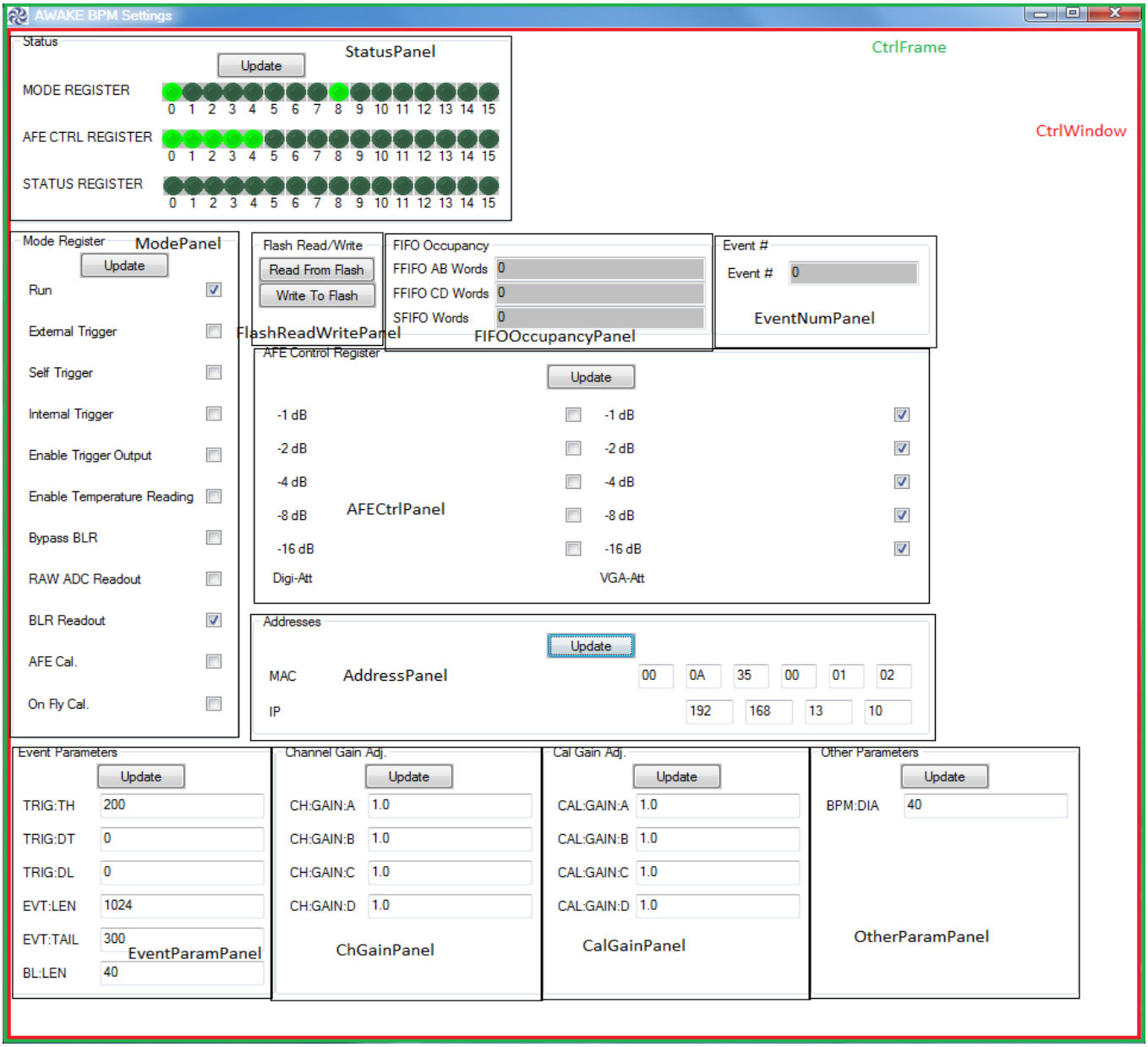

Fig. 5 - Control GUI Layout

Note: Panel borders shown above (in black) may not reflect the borders of the actual Panel objects exactly.

The layout of the GUI is accomplished using multiple nested sizers (that is, instances of *wx.BoxSizer*). These are essentially objects that handle the visual arrangement of elements in a panel, window, or frame. For example, the four panels at the bottom of the GUI (*EventParamPanel*, *ChGainPanel*¸*CalGainPanel*, and *OtherParamPanel*) are all grouped together into one sizer is that is oriented horizontally.

The contents of each panel consist primarily of buttons, text boxes, check boxes, and labels. All of these objects are part of the wxPython library. The LED objects in *StatusPanel* are created using code from Daniel Eloff (see: http://code.activestate.com/recipes/533125-wxpython-led-control/).

GUI Behaviour/Functionality

The core behaviour of the GUI is very similar to its counterpart in the LabVIEW program. There are various text and check boxes for users to provide input. Event handlers are attached to all of the 'Update' buttons shown in Fig. 5. When a button is clicked, the appropriate values within the panel that the button is part of are written to the corresponding registers in the FPGA. The one exception to this is *StatusPanel*, where the 'Update' button involves reading from the appropriate registers in the FPGA and displaying their binary values on rows of LEDs. FIFO occupancy and event # are also displayed and refreshed at regular intervals. See *Managing the Flash* in the **Instructions** section for the behaviour of the 'Write to Flash'/'Read from Flash' buttons.

Every time the GUI is opened, all panels (other than FlashReadWritePanel, FIFOOccupancyPanel, and EventNumPanel) are initialized with values read from the FPGA. Thus, any new settings that are written to the FPGA will be reflected in the GUI every time it opens.

Panels that allow for user input in a text field (such as *AddressPanel* or *EventParamPanel*) also make use of a Validator subclass, which is responsible for validating inputs (e.g. ensuring that the user does not enter letters in a numerical field).