Developing a Python Library to Support the CC-USB Control Module

And its Implementation at the Fermilab Test Beam Facility

Karen Lipa, SIST Intern

College of Engineering

University of Illinois at Urbana-Champaign

Urbana, IL

8/9/2012

Table of Contents

I. Abstract

The project aimed to develop a Python software library on Linux to communicate with the CC-USB CAMAC control module manufactured by Wiener Co. The software library consists of Python wrappers around low-level C functions, provided by the manufacturer. The low-level C functions interact with the Linux USB subsystem. To simplify the user interface for CAMAC communication, the Python wrappers were used to develop Python classes which make the CAMAC interface object oriented. The software was optimized to readout wire chambers which are part of a cosmic ray test stand at the Fermilab Test Beam Facility (FTBF).

II. Introduction

Fermi National Accelerator Laboratory, also known as Fermilab, is a United States Department of Energy National Laboratory dedicated to "advancing the understanding of the fundamental nature of matter and energy" through high energy physics. Many of the experiments at Fermilab involve observing certain events or occurrences many, many times in order to find the one time when a new discovery can be revealed. This requires the intake and storage of lots of data as these searches and measurements are being made. One of the methods utilized at Fermilab for data storage is the CAMAC crate. CAMAC crates hold up to 24 plug-in modules and one crate control module, which can interface with a computer. They employ a parallel dataway bus that enables a user to address modules by slot number, subaddress number, and function number in order to read data from the modules into the controller and write data from the controller to the individual modules. This technology has been around and in use since the early 1970s since it is still one of the most efficient and cost-effective methods for storing and accessing the large amounts of data handled at Fermilab. One more recent improvement to the system that has been developed though, is a change in the type of control module used. Fermilab has been using SCSI bus control modules for many years now, but it has now become desirable to implement the use of more standard and compact USB control modules. I have had two main general goals for this summer: implementing the CC-USB control modules manufactured by the WEINER company for use at Fermilab, and creating a Python-based software system to make the use of the CC-USB (and other control modules) easier and more efficient. I also worked specifically to tailor the software to one particular experiment. I used time-to-digital converter CAMAC modules to read out data generated by a wire chamber within a cosmic ray telescope in order to determine the locations of particles traveling through the scope.

III. Motivation

Implementing the use of the CC-USB control modules involved familiarizing myself with the data acquisition system used here at Fermilab and the CAMAC standard system. One of the big goals throughout this project was to make the code as simple and easy to understand and modify as possible for the future users. As I was only a summer intern, the software needed to be well-documented enough and sufficiently easy to understand so that physicists at Fermilab will be able to modify it in whatever ways necessary for their future projects without relying on my help to make the changes to the base code. For this reason, I worked to wrap the base code, written in C, in Python. Python is a high-level programming language designed for easy readability and modification. Wrapping the C-code enables all the functions written in C to be called and accessed from Python, which gives the user all the speed and functionality that C provides accompanied with the easy modification and user-friendliness of Python.

IV. Methods

I came into my internship at Fermilab with a basic knowledge of C programming, having just completed a college C programming course during the spring semester, but virtually no knowledge of the Python programming language, CAMAC crates, or the data acquisition systems used at Fermilab. I started off toying around with a Windows GUI that was connected to one mini CAMAC crate with a CC-USB control module, just to get a feel for how the computer communicates with the control module and the other modules within a crate. During that time period, I also spent time reading documentation on the CC-USB control module, as well as information on the CAMAC system in general. The next step was to review and gain a thorough understanding of the “example” C-code provided by the WEINER company. I spent several weeks running tests and ensuring I had a full grasp of what each function did, not only within the computer but within the CAMAC crate itself. After that, I went through the tricky process of creating wrapper functions in C that would wrap the C functions code in a way so that it would be accessible from Python. This is a moderately well-documented process, but took quite a bit of trial-and-error and searching the web to complete successfully. Following the wrapping of the C-code came the bulk of the work: writing the Python classes and methods to take the wrapped C functions and make them as simple and easy-to understand for the end user. This involved several weeks of reading up on Python programming itself and practicing with simple examples. Once I had the hang of it, I set out to develop a specific software system for the CC-USB, CAMAC devices in general, and finally for general data acquisition. Once these had all been developed, I moved my work station from Fermilab's D0 Assembly Building over to the Test Beam Facility so that I could use the system to read out data generated from the cosmic ray stand. First, I had to set up my NIM and CAMAC crates and run several standardized tests that I had developed in order to ensure that all the hardware (such as individual modules and the pulse generator) was working properly. This took quite a bit of time as many of the modules are old and have been sitting around in the Test Beam Facility for some time. During this time, I was also confronted with certain aspects of my software that were not ideal and worked to fix those, in addition to constantly improving my documentation and debugging methods.

V. The Software

CAMAC

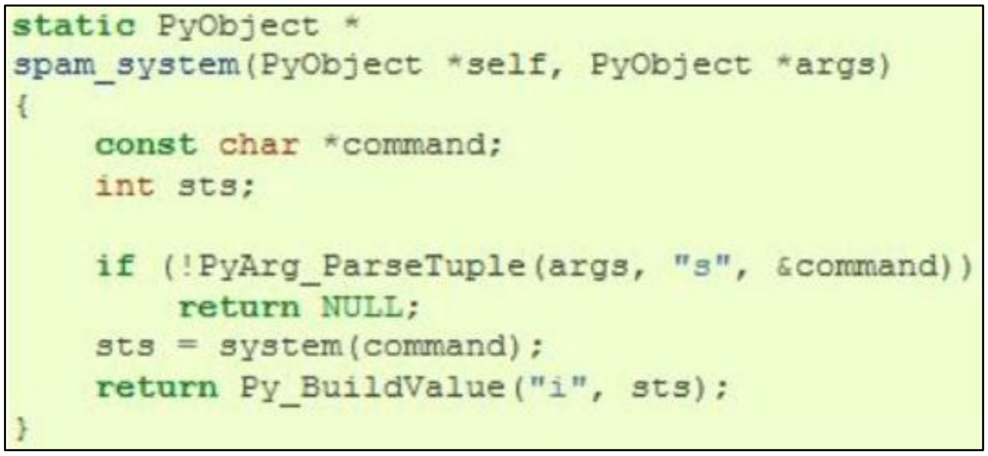
CAMAC, which stands for Computer Automated Measurement and Control, is a system which has been widely used at Fermilab for many years as a means of data acquisition and storage. As was stated in the introduction, CAMAC crates allow for communications between a computer and the individual modules within the crate through the use of the read and write busses in the back of the crate. This is an advancement from NIM crates, an older crate style that required physical adjustments, such as the insertion of pins or turning of small dials with a jewelry screw on each module in order to adjust their individual settings. While CAMAC crates have been in use for quite some time and are utilized and understood by many people at Fermilab, the CC-USB control module is a much newer technology and, as such, has not yet been properly implemented for use at the lab. The particular CC-USB control module that was used is one manufactured by WEINER Co., and came accompanied with some low-level C code to be used for accomplishing most basic functions with the module. For example, the code provided functions to find all the CC-USB devices attached to the computer, open them, and perform read, write, clear, inhibit, and initialize functions. While there were many other functions included in this example code, many of them were not necessary, or were called from other functions, and did not need to be wrapped in order to be accessed from Python.

Wrapping in Python

The purpose of wrapping C code in Python, more formally referred to as "extending Python with C," is to provide access to the C functions from Python. This is desirable because Python is a high-level, object-oriented language, with an easily understandable syntax that provides for easy comprehension and modification of the code. C code, on the other hand, has a much more complex syntax which makes it take much longer to write. By wrapping the C functions provided by the WEINER

Co. in Python, all the functions specific to the CC-USB and to USB devices in general can still be utilized, but in a much more user-friendly way.

The first step in creating a Python wrapper is to write the main wrapping function. Python and C differ in that Python is an object-oriented language (meaning that all integers, strings, etc are objects on the heap), while C is not. The wrapper function acts to mitigate this disconnect by converting between Python objects and C types within this wrapping function. There are two functions, PyArg_ParseTuple and Py_BuildValue, that act towards this purpose. PyArg_ParseTuple takes in a Python object and converts it to a C variable of a type specified by the user in the form of a format string (e.g. as shown in **Image 5.1** if a user wanted to convert the Python object “command” to a C string, they would use the format string “’s’”). This newly converted C string can now be used as an input in the C function that is being wrapped (in this example, system()). The return value is then converted back into a Python Object through the use of the Py_BuildValue function. Through this process, it would appear from the Python code that you are simply passing an object into a function, and then getting an object back out and the user does not have to worry about dealing with the underlying C function.

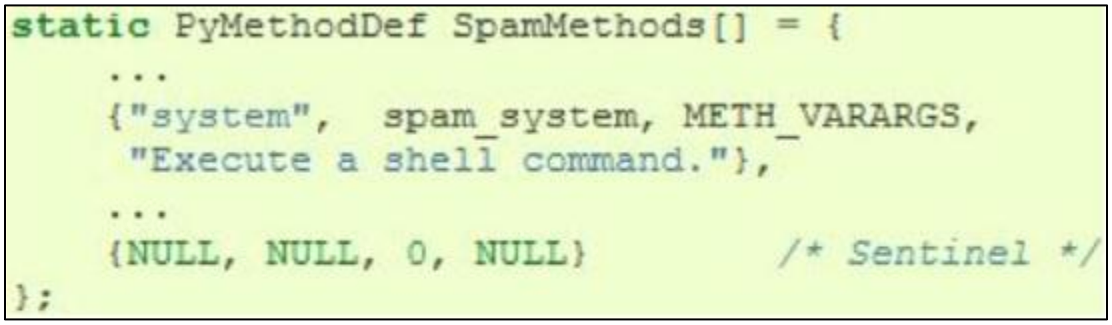


```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}
```

Photo: http://docs.Python.org/extending/extending.html#a-simple-example
Image 5.1 Example of a simple wrapper function

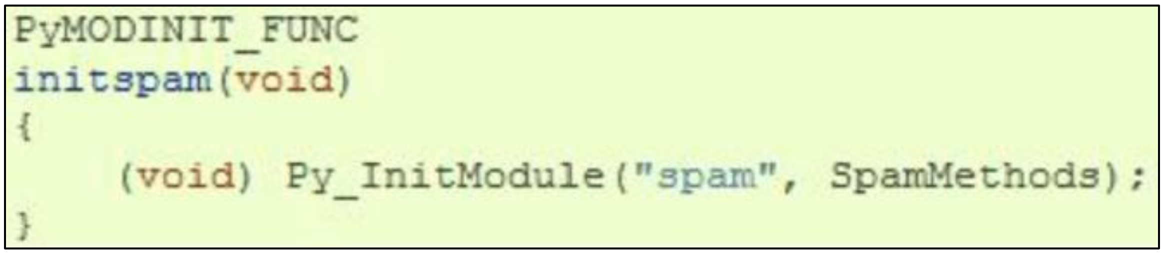
The next step in wrapping a function is creating a methods table, which basically just informs the system (and the user) of what names the user will use to call each function. This can be seen in **Image 5.2**, the methods table corresponding to the above given example of the wrapper function.



```
static PyMethodDef SpamMethods[] = {
    ...
    {"system",  spam_system, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL}        /* Sentinel */
};
```

Photo: http://docs.Python.org/extending/extending.html#a-simple-example
Image 5.2 Example of a methods table

The final part of the wrapper function file is the initialization function. This function defines the name of the module to be called from Python when the wrapper function is run. (**Image 5.3**)



```
PyMODINIT_FUNC
initspam(void)
{
    (void) Py_InitModule("spam", SpamMethods);
}
```

Photo: http://docs.Python.org/extending/extending.html#a-simple-example
Image 5.3 Example of an initialization function

Python Classes

A class in Python is basically a blueprint for the different attributes and functions that can be performed on something. For example, a class for a CAMAC crate would have attributes such as an ID number, and would include functions to open the crate and to perform the read, write, clear, initialize, and inhibit functions.

Within a class are things called methods, which are functions that apply only to instances of the class. As was mentioned, the class simply defines a blueprint, and an instance of a class must be created in order for it to be put to use. Creating an instance of a class endows that instance with all the attributes that are defined within the class and also enables all the class methods to be performed on that specific instance. The first step in setting up the Python library was to develop the base class. This was a CC-USB crate class, with methods to perform open crate, read, write, initialize, clear, and inhibit functions. From this class, a CAMAC module class was constructed, which inherited all the methods contained in the parent class, while also creating methods of its own. Certain specific modules, such as the Jorway 85A and the Lecroy 3377 were also given their own classes, with specific methods relevant to those modules. For example, within the class for the Jorway 85A (a scaler counter module) is a readscaler method, which calls the read method, with specific inputs relevant to the Jorway 85A.

Testing

After several Python classes were created, the next step was to create test scripts, which include all the necessary steps in testing a certain class to ensure that all the software was working correctly and that the proper data was being returned. These test scripts were also later used once the software was completed as a quick means of testing out pieces of hardware to make sure everything was working correctly. At this point, a test script was able to be run and data was read out for the first time from a Lecroy 3377 TDC module. The TDC works by logging a hit and then keeping track of the time from that hit until it receives a “stop” signal. The data in this testing phase came from a pulse generator, which produced both a simulated "hit" and the "stop" signal.

Multi-module (and multi-crate) readout

Once it became possible to read out the data generated by a pulse counter, the next goal was to enable multiple modules to be read out at once. In many cases (like with the cosmic ray telescope project) it is necessary to be able to read out a whole crateful, or even multiple crates full of TDCs. This involved extending the CAMAC crate class to include methods such as TDCreadout, which would loop through all the references to TDC module instances within the crate, and read each one. This process required a solid knowledge of the workings of Python and the class and inheritance systems. After the multi-module readout script had been written and successfully tested, work began on a system class, which would be able to find and open all CAMAC crates in the system, not just CC-USBs, and read them out at once. This took the specific CC-USB software and extended it to be used for all CAMAC devices since, while the CC-USB is the latest technology, there are still many SCSI control modules lying around at Fermilab, and it would be good for the software to be able to be applied to them too. At the conclusion of the summer internship, this "system readout" class and script were still not completed, as the SCSI software was not yet brought up to the same level as the software for the CC-USB.

VI. The Experiment

Fermilab Test Beam Facility describes its goal as "providing flexible, equal and open access to test beams for all detector tests, with relatively low bureaucratic overhead and a guarantee of safety, coordination and oversight." The particular project that the CC-USB control module and data acquisition system was optimized for this summer was a cosmic ray test stand. The project aims to use the test stand as a means of locating high-energy particles traveling through a telescope in order to provide an optimal system for testing new detectors. Particles are detected and located within the telescope through the use of wire chambers and scintillators that are attached to photo multiplier tubes (PMTs). These pairs of scintillators and PMTs are spaced out in a vertical fashion with several wire chambers in between in order to track the timing and locations of particles that pass through the telescope.

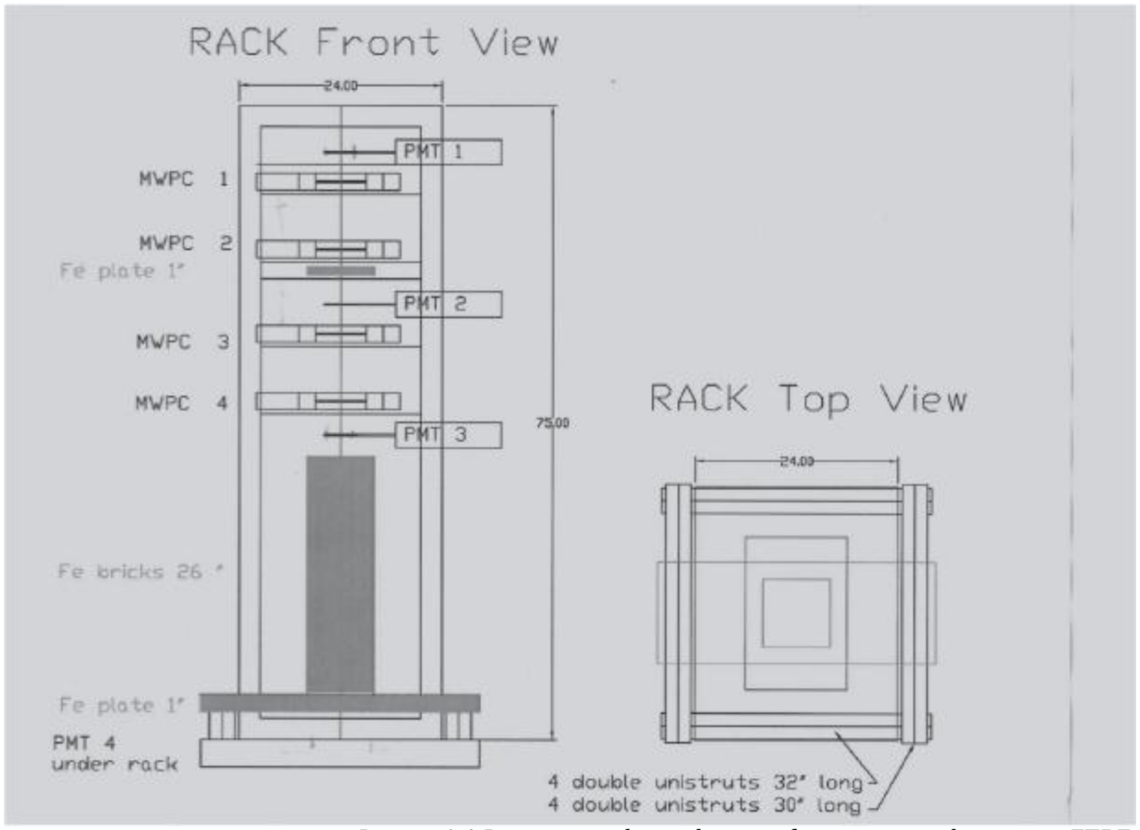


Image 6.1 Diagram outlining layout of cosmic ray telescope at FTBF

Lecroy 3377 time-to-digital converter CAMAC modules (which in this case are set to common stop, single word mode) keep track of the time between when a hit is recorded and when the stop signal is received. There are numerous different options for settings, such as the resolution of the data value, maximum number of data words per channel, etc. that can all be set using a write function command. The Python library enables the user to set the "debug level," which is a value that determines how much information is being shown upon readout. A user may choose to see the register values and what they are set to, the header word and meaning, and the data word, or may choose not to see anything as the data is simply written to a file.

A scintillator is anything that emits photons when it is struck by a high-energy particle. The attached PMT produces an electrical signal when a photon is released. As is visible from **Image 6.1**, scintillators and wire chambers are placed in line with each other within the telescope. The purpose of the scintillators is to generate the "stop" signal for the TDCs, which tells them to stop looking for hits from the wire chamber. With multiple scintillators all lined up and interspersed with the wire chambers, the hope is that if several of the scintillators were hit with a high-energy particle around the same time, that means that the particle also traveled through the wire chamber in between them.

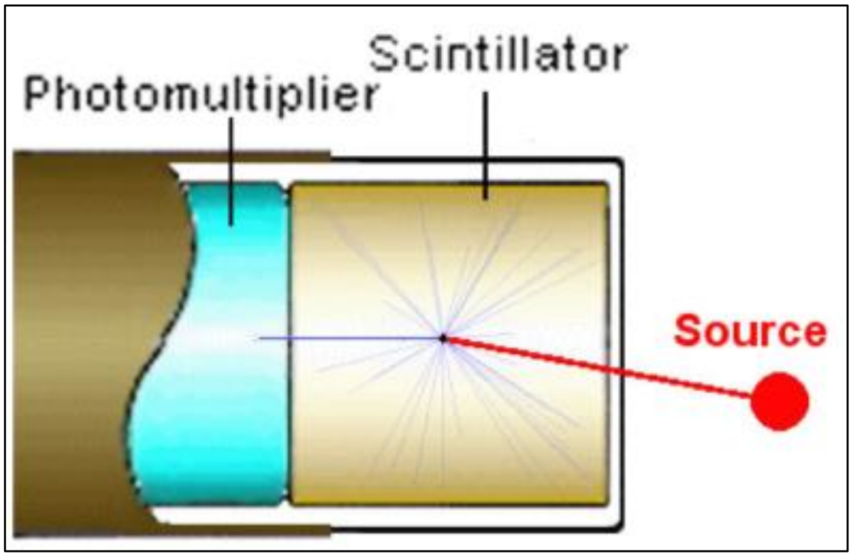


Photo: http://williamson-labs.com/ltoc/cbr-tech.htm
Image 6.2 Diagram of PMT and scintillator

A NIM module called a discriminator was used to determine the number of scintillators that needed to be in coincidence before the stop signal was transmitted to the TDCs. Requiring more scintillators to be in coincidence decreases the number of stop signals and increases the reliability of those signals in that they have an increased probably of indicating the passage of a particle through the wire chambers. The discriminator sends a stop signal when the conditions have been met, that is to say when enough scintillators had a coinciding hit. This signal is then sent via a Lemo cable to a Lecroy 4301 module within the CAMAC crate. This module enables that one Lemo signal to be converted into ECL and sent out to the many Lecroy 3377 TDCs that are in the crate, as can be seen from **Image 6.3**. The right-most blue module is a 4301. You can see that it has one green Lemo cable going in, and then an ECL output that is then distributed out to all 8 TDCs that are currently in the crate. In total, there are 16 cards of the wire chamber that need to be read-out, which requires 8 TDCs (since each TDC has two input channels). With all 4 wire chambers in the telescope in place, 24 TDCs will be needed. This will necessitate the ability to readout two CAMAC crates at a time, since each crate can only hold 22 Lecroy 3377 TDCs at a time.

Image 6.3 Stop signal sent out to TDCs via 4301 module

VII. What's Next?

As with anything in life or in science, there is always room for improvements in any project. As the cosmic ray test stand is not yet set-up, that is obviously the next clear step that needs to be taken in order for that project to move forward. When it is completed, it will contain 4 wire chambers and 4 scintillator/PMT combos, all meticulously lined up and calibrated for maximal efficiency in the readout. Software-wise, as was previously mentioned, the system class is not yet completely bug-free. It will also be necessary for the SCSI software system (which was worked on by another intern this summer) to be brought up to functionality so that it may be used in conjunction with the data acquisition and system readout systems developed for this project.

VIII. Acknowledgements

I would personally like to thank my Supervisor, Geoff Savage, for providing me with such a challenging and worthwhile project to work on this summer. I would also like to thank Dianne Engram, Sandra Charles, Jamieson Olsen, and Elliott McCrory, who supported me throughout this process. In addition, I would like to thank the entire SIST committee and Fermi National Accelerator Laboratory for selecting me into this prestigious program and facilitating the great amount of learning that took place for me this summer.