Abstract

The University of South Dakota has been working with techniques and methods to generate high purity germanium crystals to be used in detectors. There is a high demand for high purity germanium detectors, but a very limited supply. We would like have a portable detector system to demonstrate our capability to have a practical (not just for lab testing) detector system to address the increasing demand for HPGe detectors in the physics research field**.** Currently, the University of South Dakota is able to test their own crystals, but only on a desktop setup in the lab; we aim to create a small, portable cryostat that can test high purity germanium crystals and complete the fabrication chain. To accomplish this, we developed a small system that is capable of monitoring and setting the voltage that is supplied to the germanium detector. We then extended this control through a webui that would allow devices on the same network to access, monitor, and set the voltage with an intuitive interface.

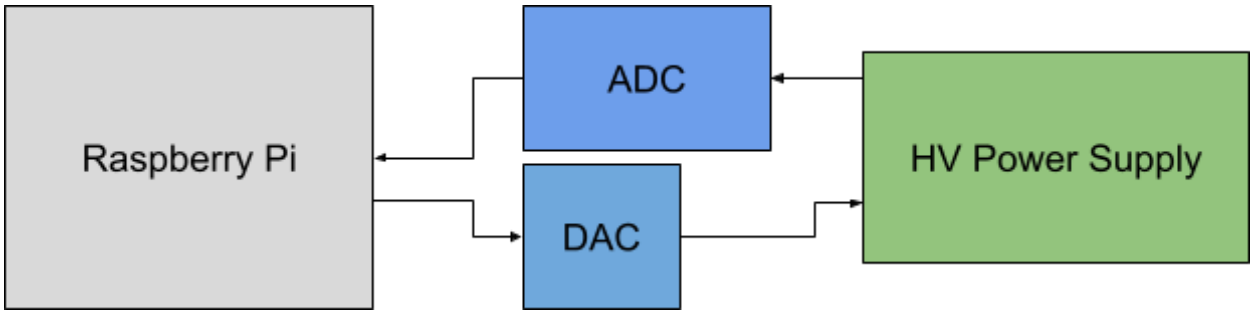
**** Notes **********
Control and monitor HV power supplied to the cryostat.

Hardware: 1 Adafruit ADC ADS1115 - Monitors Voltage, 1 Adafruit DAC MCP4725 - Sets Voltage, Raspberry Pi (Computer)
Software: Python (Flask Framework, Adafruit Libraries for the ADC and DAC, matplotlib for graphs), HTML5, CSS (Bootstrap), JS (Web Interface), C/C++ Integration

Control and Monitor Voltage via a basic web interface that is served by the Pi at it's IP address, which can change.
****

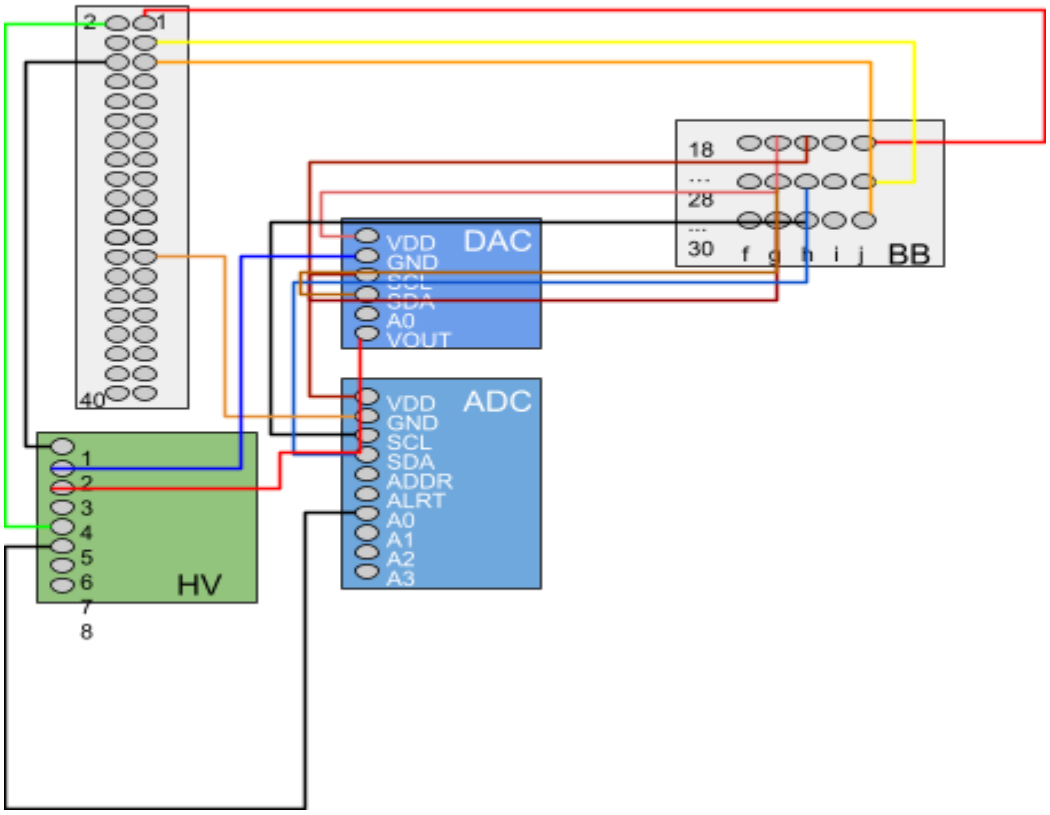
Diagrams

High Voltage and Monitoring System Diagram



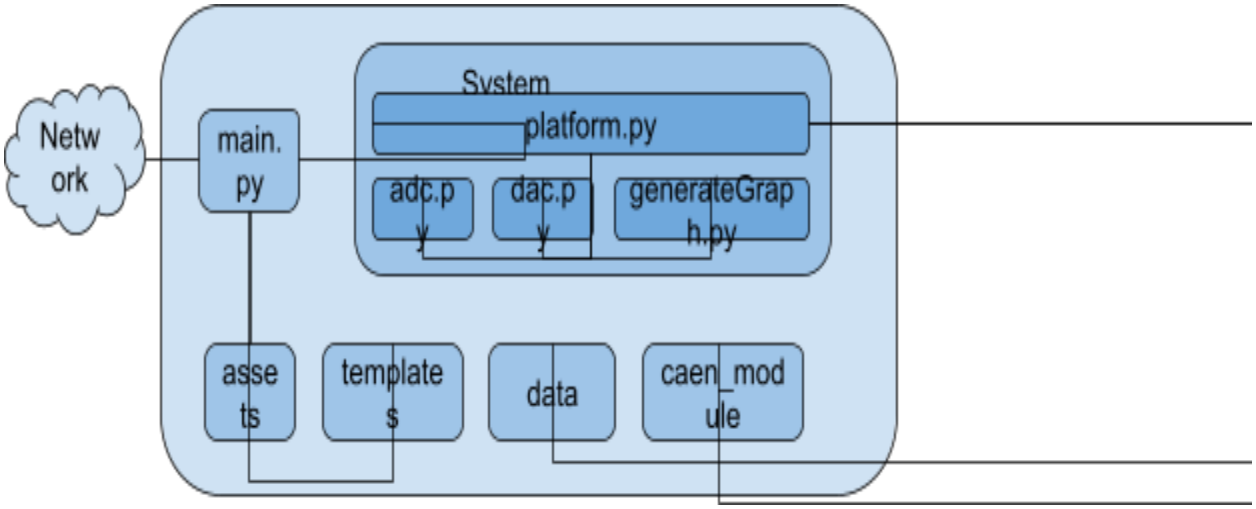
This is a simplified diagram of the basic system layout. Our control and management software runs on the Raspberry Pi and utilizes an ADC (Analog to Digital Converter) to monitor the voltage across the High-Voltage Power Supply and a DAC (Digital to Analog Converter) to control the High-Voltage Power Supply.

Wiring Diagram:



This diagram provides an overview of how the hardware devices above are wired. The left-most pin board is connected to the Raspberry Pi. Because of how the component is wired, the pins are reflected across the y-axis from how the Raspberry Pi pins are typically formatted. Please refer to the numbers on this diagram to see how they line up with the typical Raspberry Pi pinboard. The DAC is the Adafruit MCP4725 model (https://learn.adafruit.com/mcp4725-12-bit-dac-with-raspberry-pi/hooking-it-up) and the ADC is the Adafruit ADS1115 model (https://learn.adafruit.com/raspberry-pi-analog-to-digital-converters/ads1015-slash-ads1115). The HV is the EMCO CA20 High Voltage Power supply, which supplies up to 2000V. Connections between the devices are shown in the diagram above. For more information concerning how the individual components work, please refer yourself to the provided links.

Software Diagram



There were several goals in developing the control software. We attempted to make the code as modular as possible so reduce digging through large confusing files and to make adding other modules easy and fast.

Loop Identification and Code Explanation

System Implementation:

Control Interface (Webapp - main.py, assets, templates):

Written in Python 3 we utilized the flask framework to speed up and ease development. It uses a simple HTML5 based web page with a touch of javascript. To style it we used Bootstrap 4.0. In this model, that means there is a client and the server. The user's web browser is the client and the Pi acts as the server. To handle the communication between them, we developed a RESTful api (http://www.restapitutorial.com/). The basic summation is that we make HTTP requests (GET, POST, PUT, etc) on specific URLs to handle communication between the client and server. The actual information is transferred via JSON objects (refer to main.py's comments to better understand what is going on).

System Backend (System Directory, data, caen_module):

The system backend is written in exclusively python 3, with the exception of one component (caen_module) which is further documented below. The key component is platform.py, which acts as the interface to the whole system. All subsequent

components and devices are written as modules and used by platform.py. There are several advantages to using this format. For example, each individual component can be directly run to use test changes and extend functionality. Furthermore, platform.py also handles data collection and the generation of our voltage graph via generateGraph.py. The module generateGraph.py utilizes the matplotlib python module to generate graphs based on a file of data. The usage of the system should feel quite familiar if the user has experience with Matlab.

The caen_module is different from the other modules, as it is written in C/C++. We were able to get permission to utilize a key component of the source code as a reference to developing software that we can use to talk to the CAEN DT5770. In fact, the code is really just modified to be capable of compiling on linux, too, utilizing the c preprocessor (i.e. #ifdef _WIN32). Note that there is one file that hasn't been integrated yet, DPP.cpp, and that is because winthreads needs to be converted to POSIX threads. This file doesn't appear contain anything necessary for communicating with the device and is really just an abstract layer for the developers to talk to the device. However, what it has that would be helpful is information on how to communicate with the device along with macros that make useful aliases. The code utilizes the ftd2xx.so driver provided by ftdi which they have released for a variety of platforms and architectures. The project utilizes cmake to build, two of its most important files are libftd2xx-<arch>.a and libDT5770.a (libDT5770 could potentially be used with more devices but it is the one we tested it with). In the future, these libraries should be used to accomplish something useful. Controlling the device is like most devices: you write specific data to specific parts of memory, the devices reads those and writes data to other pieces of memory. You'll need to figure out which registers you need to write to and where to read from. They have headers with definitions that will most likely make this easier. Unfortunately I haven't been able to create documentation related to the interfaces but the headers do contain useful info and documentation.

Example with plot

(bits/precision/digitization-range of the input/output 2^x in ADC and DAC conversion)
In our case: 2V-2620V
So 12 Bits of Precision: (2620-2)/(2^12) ~ 0.64V/step