DSP Platform Firmware and Software Redesign

Final Report

Team 47 - Spring 2023
Client: Matthew Post
Advisor: Dr. Phillip Jones

Yohan Bopearatchy - Firmware Engineer
Wyatt Duberstein - CLI Engineer
Blake Fisher - GUI Lead
Corbin Kems - CLI Engineer Lead
Cole Langner - Testing Lead
Jens Rasmussen - Project Lead
Long Zeng - Firmware Engineer Lead

Team Email: sdmay23-47@iastate.edu
Team Website: https://sdmay23-47.sd.ece.iastate.edu

Introduction

Problem Statement

In the Signals and Systems I (EE 224) course in Iowa State's Electrical Engineering curriculum, the students have been using the CyDAQ device to enhance their lab experience and deliver real-world results. Currently, the device allows students to measure and record electrical signals being input to the device and outputs those measurements into graph files. Due to the hardware on the device not being properly utilized by inefficient firmware, the data transfer time is very high and requires a lot of waiting to be sent to the user's computer. There are also bugs within the user interface that have been causing issues for the students completing and TAs administering the labs, such as data not displaying correctly on the CyDAQ graphs or the user interface not being user-friendly.

Our proposed solution to these problems is to rework the firmware from the ground up, utilizing the dual-core functionality of the CyDAQ for faster processing power and increasing the efficiency of the firmware code. The CyDAQ will also be implemented with USB communication to the host computer rather than UART, increasing the sampling speed to 10 MSps (internal) and 1 to 2 MSps (external). The User Interface will be completely redone in another language, using PyQt and focusing on intuitive and adaptable design. The design of the User Interface will primarily focus on making it easy for TAs or other senior design groups to add future functionality. This will be done by creating thorough documentation and following common coding conventions.

Intended Users and Uses

Electrical Engineering Students at ISU

Users will participate in EE 224, EE 324, or any other signal processing-related classes at ISU and are interested/passionate about electrical engineering and signal processing. Users will need a quicker CyDAQ for their class's labs, increased reliability of signal processing equipment, and easy navigation through the GUI that is simple to understand and can safely handle errors and misconfigurations. Users will use the CyDAQ in labs and benefit from a new, easy-to-use UI that enables faster learning.

Electrical Engineering TAs and Professors at ISU

Users are instructing classes in the ECpE department that use the CyDAQ, are knowledgeable about signal processing, and know the ins and outs of the CyDAQ to aid in problems that may arise. Users will need a quicker CyDAQ for their class's labs, increased reliability of signal processing equipment, and easy navigation through the GUI that is simple to understand and can safely handle errors and misconfigurations. Users will benefit from better output and error reporting making it easier to help students that run into issues, and an easier-to-use UI will make teaching the software much quicker.

Instructors and Researchers at Other Universities

Users are interested in streamlining signal processing learning activities and looking for open-source signal processing technology. Users need to be able to access signal processing technology in an open-source environment and want to utilize the platform in different ways not necessarily specific to ISU. A benefit for users is that an open-source project eliminates the need to design its product from the ground up, costing time and money.

Requirements

Functional

1. The firmware, CLI, and GUI fully support two unique modes (Basic Operation and Balance Beam).
2. Fully support the configuration of Muxes, Sampling Rates, Generation Rates, Filters, and Balance Beam in the CLI and GUI.
3. Export of data in CSV or MAT file format.
4. Completely decoupled CLI and GUI.
5. 2 Msps transfer speed between firmware on the CyDAQ and the USB driver on Windows.
6. 10 Msps transfer speed communication between the two CPU cores on the CyDAQ.

Aesthetic

1. Considerably more appealing GUI compared to the existing Matlab product.
2. Support both modes (Basic Operation and Balance Beam) in one application.
3. Fully functional command line window to run all commands without the GUI.

User Experience

1. Configuration of CyDAQ in labs with minimal effort or confusion.
2. Logging and handling of errors with the CyDAQ firmware, CLI, and GUI.

Resource

1. Utilize the previous implementation of CyDAQ hardware.
2. Interact with the CyDAQ accessories used for labs (DAD, balance beam, etc.)
3. Deployed on Windows lab computers.

Constraints

1. Run without administrator privileges on the Windows computers in the labs.

Engineering Standards

Universal Serial Bus (USB) 2.0

The CyDAQ should be able to connect to any Windows PC over at least USB 2.0

USB Communications Device Class (USB CDC)

The CyDAQ should support the USB standard that allows for Ethernet frames to be transmitted to and from the device over a USB connection.

USB CDC Abstract Control Model (ACM)

The CyDAQ should support the USB standard that emulates a serial connection between it and the connected PC over USB.

Universal Asynchronous Receiver-Transmitter (UART)

Commands sent to and from the CyDAQ will be over UART, so the CLI tool and communication application running on the CyDAQ must support sending and receiving commands over this standard.

Remote Network Driver Interface Specification (RNDIS)

The CyDAQ should support the USB standard specific to Windows for Ethernet emulation. It is similar to the USB CDC device class but is required on newer Windows systems.

The OpenAMP Framework Specification

The CyDAQ's firmware will utilize the Open Asymmetric Multi-Processing framework standards to communicate between its asynchronously running CPU cores. This includes the remoteproc and RPMsg protocols.

Project Design

Design Evolution

User Interface

Since 491, the application has undergone significant developments with new modes, including balance beam, debug, and plotter modes. These additions have expanded the application's feature set and presented challenges in designing those interfaces. Changes have also been made to the basic operation mode, such as adding a send configuration button and updating labels to be more descriptive. Additionally, the connection indicator for the CyDAQ has been made global rather than duplicated on every page, reducing the code for that component.

Firmware

The firmware design made considerable changes in the second semester of the project. This was because we discovered the bare metal firmware implementation of USB communication provided by Xilinx was riddled with bugs and couldn't hit our required transfer speed. We determined that manually implementing a USB CDC device class in firmware would take too long for how much time we had left in the semester, so we searched for a new solution.

We decided to use Petalinux, a lightweight embedded Linux operating system based on Yocto Linux. The Linux kernel contains USB drivers that are much more stable and well-refined. Our research also showed that the drivers are much more configurable, making future changes much easier. The biggest downside of using Petalinux is that it adds large overhead to I/O operations and has inconsistent timings because of its nondeterministic process scheduler. This will be a problem when trying to sample data using the ADC on the Zybo board, as interrupts need to occur and be processed faster than the highest supported sampling rate. This is only possible with a bare metal application running on at least one of the two CPUs on the Zybo board. After more research, we discovered that this is possible in Petalinux with the Remote Processor Framework, which allows running one (or multiple) CPU core(s) independently of the operating system to enable bare metal applications to be started and stopped in Petalinux runtime.

The biggest challenge of using the Remote Processor Framework is communication. Because each CPU core in our solution will run independently, asynchronous communication is a requirement. Thankfully the OpenAMP, or Open Asynchronous Multiprocessing, library provides an interface for communication between two processes over a shared memory region called RPMsg. In theory, messages can be sent and received from both CPU cores with payloads containing configuration parameters and error/success messages.

The last challenge was how to handle sampling data. Because we need to support 50 kHz sample rates, transferring all that data using RPMsg wouldn't be the best choice as the protocol isn't designed for such high throughput. Instead, we decided to carve out a second shared memory region that the interrupt handler can write to during sampling, then have Petalinux read from that same memory region once sampling is complete. Messages sent over RPMsg will be used to prevent race conditions in the second shared memory region.

We produced a diagram outlining what the two CPU cores will be running and how they will communicate. The implementation section later in this document outlines much of the finer details of how each application works, the size and location of each shared memory region, and the specific configuration of USB Gadgets.

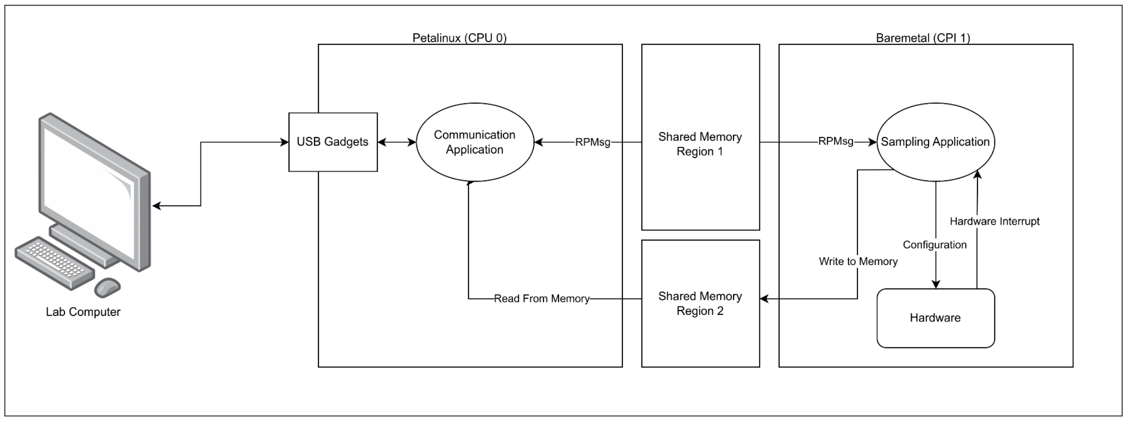


Figure 1 - Firmware Design

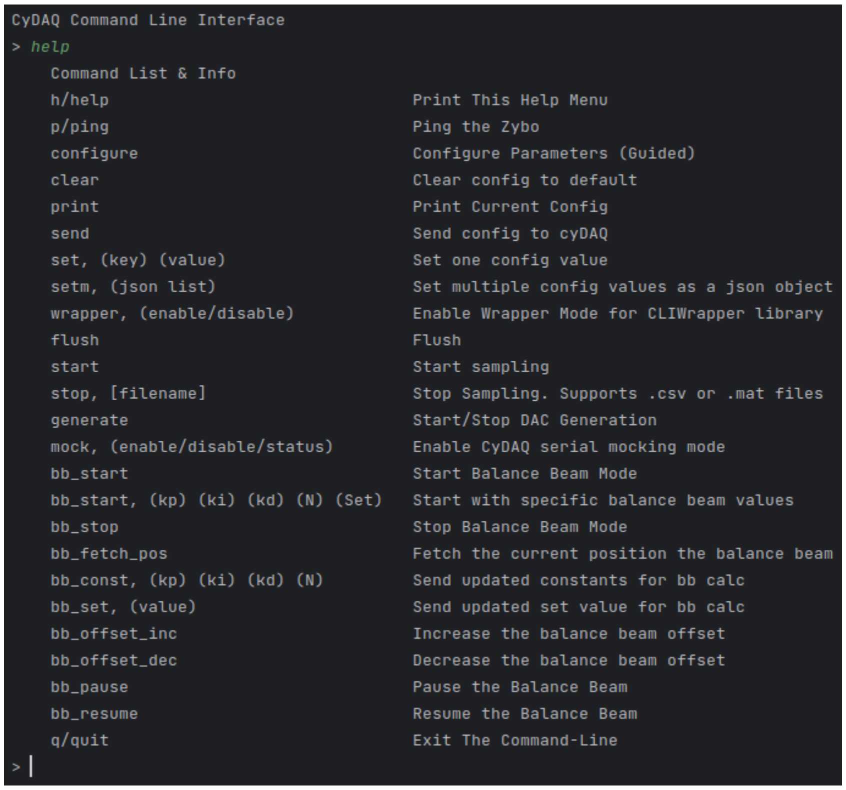
Implementation

User Interface

The GUI was developed using the PyQt5 framework and provided a simple and intuitive way to control the hardware. It allowed us to set various parameters for the CyDAQ, such as sampling rate and signal frequency in the Basic Operation mode and provided real-time feedback on the current state of the hardware through a multi-layered connection system, including a wrapper class for the CLI tool. The GUI also included tools for receiving the sampled data from the CyDAQ and saving it in CSV or MAT format. Overall, our user interface implementation was designed to be user-friendly and intuitive, allowing us to easily control and interact with the hardware.

CLI Tool

The first component for connecting to the CyDAQ is the Command-Line Interface tool, which a previous senior design group developed and we heavily modified to fit our needs. This tool is the lowest layer in the communication between the User Interface and the CyDAQ, as it handles the serial communication between the CyDAQ and the computer. This tool contains a list of commands that can be sent (List of Commands - Figure 2 below) to the CyDAQ for configuring the various modes and receiving data back from the CyDAQ. The CLI tool mainly uses the pyserial library to convert user commands and inputs into buffers sent to the CyDAQ, then returns them to the tool and interprets them. All commands return an “ACK” for a successful command and acknowledgment, or “ERR” if there was an error due to the command sent. Some commands may return other data, such as a few balance beam commands or the “stop_sampling” command, which will fetch sampled data from the CyDAQ and output it to a file. The fetching of sample data is done through the SCP application on the network connection provided over USB. This is made possible because the COMM application running on Petalinux writes sample data to a specific file location when sampling is completed.

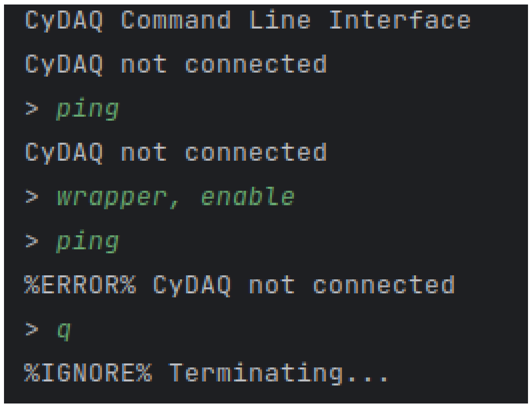


```
CyDAQ Command Line Interface
> help
    Command List & Info
    h/help                              Print This Help Menu
    p/ping                              Ping the Zybo
    configure                           Configure Parameters (Guided)
    clear                               Clear config to default
    print                               Print Current Config
    send                                Send config to cyDAQ
    set, (key) (value)                  Set one config value
    setm, (json list)                   Set multiple config values as a json object
    wrapper, (enable/disable)           Enable Wrapper Mode for CLIWrapper library
    flush                               Flush
    start                               Start sampling
    stop, [filename]                    Stop Sampling. Supports .csv or .mat files
    generate                            Start/Stop DAC Generation
    mock, (enable/disable/status)       Enable CyDAQ serial mocking mode
    bb_start                            Start Balance Beam Mode
    bb_start, (kp) (ki) (kd) (N) (Set)  Start with specific balance beam values
    bb_stop                             Stop Balance Beam Mode
    bb_fetch_pos                        Fetch the current position the balance beam
    bb_const, (kp) (ki) (kd) (N)        Send updated constants for bb calc
    bb_set, (value)                     Send updated set value for bb calc
    bb_offset_inc                       Increase the balance beam offset
    bb_offset_dec                       Decrease the balance beam offset
    bb_pause                            Pause the Balance Beam
    bb_resume                           Resume the Balance Beam
    q/quit                              Exit The Command-Line
> |
```

Figure 2 - List of Commands

Wrapper Mode

When the CLI is in Wrapper Mode, the outputs sent to stdout have special markings so the CLI Wrapper can parse the output type it is (more on the CLI Wrapper below). These differentiate between error responses, normal responses, responses that can be ignored, and live data being spit out from the balance beam mode. Below are some examples of the wrapper message outputs.



```
CyDAQ Command Line Interface
CyDAQ not connected
> ping
CyDAQ not connected
> wrapper, enable
> ping
%ERROR% CyDAQ not connected
> q
%IGNORE% Terminating...
```

Figure 3 - Wrapper Mode Messages

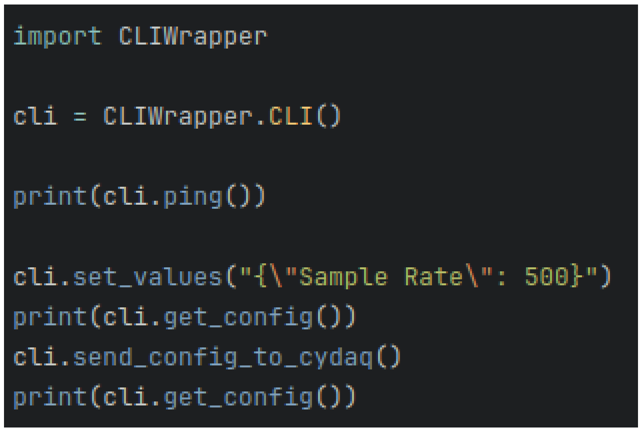
Mock Mode

We wanted a way to develop the CyDAQ user interface without needing a physical CyDAQ connected to the host PC. Doing so would make GUI development faster and easier, as we had limited CyDAQs to work with. This was done with the help of Linux pseudoterminals to mock a serial connection. A command was added to the CLI tool to enable and disable a new "mock mode". When enabled, a serial connection is made that appears the same as a CyDAQ connected physically. A single function was created that parses incoming messages and responds with hard-coded responses just as the CyDAQ firmware would do, except without any actual device configuration. Having these messages mocked allows for GUI developers to develop features based on the mocked function in the event they don't have a physical CyDAQ to test on.

CLI Wrapper

The CLI Wrapper is a tool that was created in Python to wrap the CLI in an internal shell in which the User Interface could send commands to and receive feedback from as long as the application is running. This communicates to the CyDAQ with an existing tool without implementing anything else.

The CLI Wrapper uses a Python package named pexpect to generate internal shell sessions, which run the standard CLI tool. It then expects certain feedback from the CLI tool to know if it is connected properly and running and to receive responses from commands to know if the command was sent successfully and/or if there is data to be received from the CyDAQ. This is how the User Interface sends and receives commands from the CyDAQ.



```
import CLIWrapper

cli = CLIWrapper.CLI()

print(cli.ping())

cli.set_values("{\"Sample Rate\": 500}")
print(cli.get_config())
cli.send_config_to_cydaq()
print(cli.get_config())
```

Figure 4 - CLI Wrapper Code Example

Each method of the CLI Wrapper is blocking, which means it needs to be run asynchronously with the main thread of the UI. This is achieved using the existing PyQT5 QRunnable and QThreadPool classes to push the function calls into another thread. This means that long-running functions, like sampling/retrieving data, won't cause the GUI to freeze when run.

The CLI Wrapper class raises custom exceptions when certain errors occur, for example, a sudden disconnect of the CyDAQ. These errors can be caught by the GUI code and handled appropriately.

PyQt5 Framework

The GUI tool is an interface written in PyQt5. It utilizes an instance of the CLI wrapper class to communicate with the CyDAQ. The main code can be found in the app.py file. The MainWindow class is the app's starting point; all other widgets/elements are added in separate files for organization. It also contains a thread pool for running commands asynchronously and the CLI Wrapper. Each widget is also its own class, which can all be found in the widgets directory in the repository. We chose the PyQt5 Framework because it was the best framework we could find for Python user interfaces, and it has plenty of methods and objects that could serve all of our needs.

The MainWindow class is the actual window of the application, which acts as a container for all widgets to be viewed. On the MainWindow interface, there are two main widgets: the central widget, where all widgets are viewed in a Qt StackedLayout, and the status indicator widget, which can be seen on all pages of the user interface. When using the mode selector widget (the first one loaded upon startup), the buttons will take you to the other pages by switching to that index in the StackedLayout, which will display that widget. See the image below for a more detailed idea of how the view is laid out:

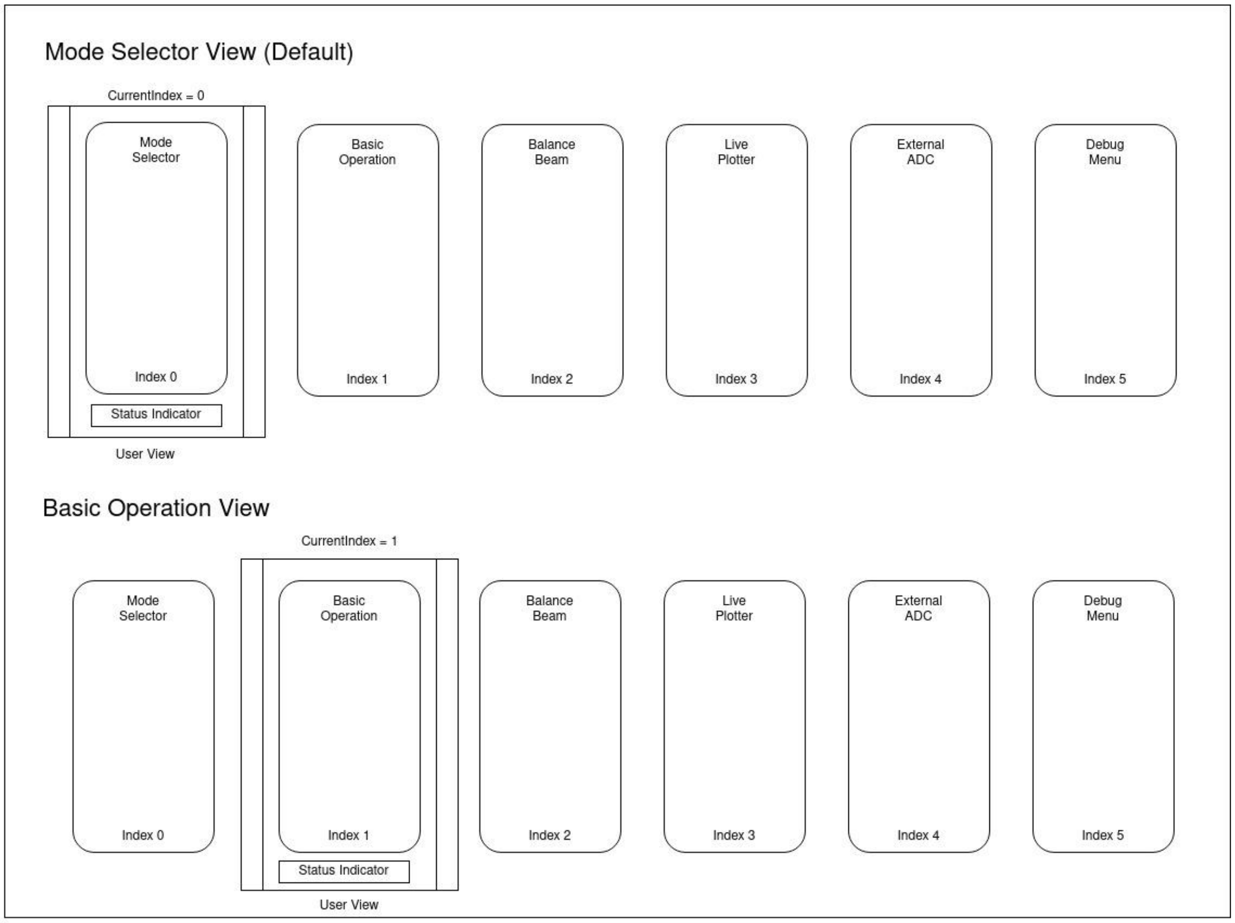


Figure 5 - StackedLayout Design

Because the User Interface uses lots of imported Python libraries and because the CLI Wrapper can take a while to boot up depending on the connection to the CyDAQ, we have implemented a splash screen as well to let the user know that the user interface is booting up.

Pages are designed in the QtDesigner 5 application and then built into Python class files using the build commands provided for the Qt Framework. This allows backend logic to be handled on the module page and various components, layouts, and forms to be added to the design. Once the page is designed and built, it is imported into the widget backend, which includes a class for the page and imports the MainWindow object and the generated UI page. The page is then added to the MainWindow and assigned to a button on QtDesigner and the mode selector widget code.

We use the built-in Python logging library to handle terminal, file, and GUI logging. The logger object is first created in the init function of MainWindow in app.py, then passed to each subsequent widget/window that needs it. The logging object also gets passed to the Wrapper, so all wrapper commands and responses are included in logging outputs.

The Debug page uses the logging object to write logs to the GUI. Check out the LogHandler class in debug.py for its implementation and the Debug section below for more information.

For the connection indicator on the user interface, we have it set to ping the CyDAQ every second to check that the connection is still live. If the ping result returns “CyDAQ not connected," the connection has died, and the connection status indicator turns red. This timer continues after the CyDAQ is disconnected, so if the CyDAQ is reconnected at any time, the user interface will still be pinging to take notice when the connection is alive again and change the status indicator to green.

Mode Selector

The first and most important page on the user interface is the Mode Selector page. This page is shown first on the user interface when started up and has buttons to get to every other page. Currently, two pages are enabled, Basic Operation and Balance Beam, with two other page buttons set up for future pages to be made. Buttons can be enabled and disabled by following the format in the mode_selector.py file.

Basic Operation

This page allows for using the CyDAQ’s signal processing feature and sampling data received through the CyDAQ and saving it to the host computer. First, the user can enter a sample rate from 100 Hz to 50,000 Hz. This impacts how many samples per second the CyDAQ takes. There is then an Input selection type, where the user can choose between a few different input types for the CyDAQ to be expecting. The next part is the Filter. This is a part where, if the CyDAQ is part of a circuit, it can filter out specific signals. When the user switches to a mode requiring a Low and High Corner, the input boxes for each respective corner appear. When the user switches to a filter requiring a mid-corner, the input box for the corner appears, and the low and high corner inputs are removed. Only the necessary inputs for each filter mode appear.

Once the configuration is done, there are two options that the user can now do. If the user only wants the CyDAQ to be a filter for a circuit, then they can send the config to the CyDAQ by pressing the “Send Config” button. This button will send the config from the user interface to the CyDAQ. If there is any error in sending the config or the CyDAQ rejects the config, the button will turn red and let you know there was an error. Otherwise, it will turn green and let the user know it is successfully configured.

If the user wishes to take samples of the input to the CyDAQ, they can use the “Start Sampling'' button. The user interface will start sampling and stop when the user clicks “Stop Sampling”, or the user can set a predetermined length of time in seconds to take samples, in which case they can stop the sampling at any time by pressing the “Stop Sampling” button.

To improve the user experience, the user interface asks for a filename once sampling is stopped. While this happens, the sample data is written to a temporary file in C:\Temp on the lab computers, and when the user selects the target file, it copies the file. This allows samples to be written to file while the user works on the filename, saving time by multitasking.

Balance Beam

This page is a rewrite of the existing MATLAB balance beam GUI for the CyDAQ. This balance beam page has the same layout as the previous one, except for the save step button, as that one was no longer needed for our design.

There are a variety of buttons and configurations for the balance beam mode. First, there are the start and stop buttons, which will only work when the CyDAQ is connected. If the CyDAQ is connected, the user interface will send the start command to the CyDAQ and wait for a response from the CyDAQ. If the balance beam module is not connected, the CLI will return false, and the user interface will let the user know that the balance beam is not connected. It will start the balance beam mode and plot the data it receives if it is connected. While in this mode, it will disable the ping timer used to check for a connection, as it considers the balance beam streaming as a connection.

The Balance Beam on the CyDAQ works by taking measurements of where the ball is on the beam and sending them through serial communication to the CLI tool, one byte at a time. The CLI tool reads this data from the buffer and prints it to stdout. When the CLI tool is in wrapper mode, the data is no longer sent directly to stdout but is stored in a variable that can be called with the bb_fetch_pos command. The output is then wrapped in %BB_LIVE% to tell the wrapper that this output needs to be treated differently.

On the user interface, the library pglive is used for the live plotting of the position of the balance beam. This library is an extension of the pyqtgraph library, with enhanced methods for live plotting that focus on performance and speed. A plotting thread starts in the user interface, which repeatedly sends the “bb_fetch_pos” command mentioned above to the CyDAQ, retrieves the current balance beam position, and plots it. The pglive graphing library adds it to the line graph on the balance beam page.

The other buttons include the stop, send constants, send set point, save data, offset +/-, and pause. These buttons will only work if the CyDAQ is connected and if the balance beam mode is active. The stop button stops the balance beam mode on the CyDAQ and the plotting and resumes the ping timer.

The balance beam uses a few constants to determine how it levels out the ball and where it places it. The constants are kp, ki, kd, and N. These mathematical constants are calculated outside the user interface and then input into the user interface. The user can input the values before starting the balance beam and click “Send Constants” to just send the values or “Start” to send the values and start the program.

The Set point is where the CyDAQ will aim to place the ball on the balance beam. This can be set in cm on the user interface and sent to the CyDAQ using the “Send Set Point” to just send the value or “Start” to send the value and start the program.

The Save Plot Data button is used to store the plot data that has been recorded so far on the CyDAQ. Currently, it stores the values in memory and saves that dict to a MAT or CSV file that the user names. We had implemented a procedure for this that was very similar to the log and basic operation files, where the data is stored in the C:\Temp folder and then copied over when the user requests it, but we could not get that working before the deadline. That will be covered in the future work section of this document.

The Offset +/- buttons calibrate the Balance Beam Sensors. Each time they are clicked, the ball is moved in the positive or negative direction on the balance beam just slightly to ensure that the ball is where it is set in the set point.

The Pause/Resume button is self-explanatory, it pauses or resumes the plotting and balance beam. As you can see in the command list shown above (Figure 1), most of the balance beam buttons/functions are all CyDAQ commands, which means that almost every button on the page (except for the Save Plot Data button) sends a command to the CyDAQ. Because the plotting thread sends a command roughly every millisecond, sending other commands and expecting feedback can get a bit complicated, so plotting is paused when another command is sent just for less than a millisecond to prevent conflicts.

Debug

This page has various debugging tools that we felt necessary to add to the user interface. It has a live debug log that is updated live using the abovementioned logging library. The debug logs can be scrolled through, cleared, and exported to a file at the user’s request. The logging works similarly to the basic operation, as the log information is stored in a temporary file in the C:\Temp directory and then copied to the user’s save location. The log file is overwritten on every startup of the CyDAQ but can be viewed even following a crash of the user interface.

Live Plotter & External ADC

Two more pages exist within the user interface but are currently disabled because we could not finish those modes in time.

The first one is the Live Plotter. Originally, this one was going to be on the same page as the Basic Operation mode and would plot the sample data as it was received by the CyDAQ, but we weren’t able to implement that on the firmware or the user interface. As it stands right now, the live plotter is its own page that can read a csv file and plot the data it reads on there on a separate plotting window using the pglive library for live plotting. Since this feature doesn't serve much of a purpose, it was disabled until future groups could work on it.

The second disabled page is the External ADC. This is another feature that CyDAQ has that we were not able to complete in time on the firmware side of the project, and therefore never designed a GUI page to work with it as well.

PyInstaller

We used the PyInstaller library to compile the program into an executable file. Right now, we have the compile.sh script that runs the compiler using one command to do so. This is where we configure the application icon, the splash screen on startup, and currently, we have it configured to pack everything it needs (libraries, modules) into one executable (.exe) file. There is also the option to have the libraries outside the executable in folders, but we decided that one file would be best for testing and lab rollout this semester.

Firmware

Developing firmware for the Zynq 7000 series Zybo Z7 development board involved multiple development stages. Each stage presented its unique challenges and required a lot of trial and error. Each stage is outlined below.

Development Environment

Xilinx recommends a handful of development environments for their platforms, so choosing one that supports Vitis, Vivado, and Petalinux tools was an important first step. An Ubuntu 18.04 desktop virtual machine running in Virtual Box was chosen as it had the best support for Petalinux tools and provided a desktop environment to install and run Vivado and Vitis. Making a virtual machine also allows multiple team members to work concurrently. We have created a setup walkthrough for creating a virtual machine in Appendix I Section 4.

Building Petalinux

Xilinx provides a starter Petalinux build specifically for the Zybo Z7 board. This allowed us to save a lot of time, as it already comes configured with most of the devices on the Zybo board, including USB and Ethernet. It also comes with an FPGA build, which we modified to support our IO needs. More information about the FPGA configuration is provided in the hardware section below.

Configuring USB Drivers

Although the starter Petalinux build provided by Xilinx contained USB support, we needed to modify it for our needs. Our initial research determined that the ACM and ECM USB gadgets needed to be configured and RNDIS for Windows specifically. All three must be enabled in the Linux kernel configuration with Petalinux tools. Further research of these drivers determined that configuring the drivers at runtime also required the configfs kernel module to be enabled. Once all the required kernel modules were built into Petalinux, they could be enabled through a bash script that runs on boot of Petalinux. Configfs makes this process very easy, as enabling/disabling features and setting configuration values is as simple as creating directories and putting files in them with the correct names and contents. This can easily be done with a bash script that runs directly after Petalinux finishes booting.

Device Tree Customization

The Petalinux operating system utilizes the device tree to define hardware, simplifying the interaction between the kernel and other programs with the hardware. The majority of the device tree is automatically generated by the Petalinux tool when adapting the hardware description. However, as we changed our hardware IP cores, we also had to modify the device tree to incorporate additional information about the modifications. We accomplished this by editing the system-user device tree file, which is an add-on to the main device tree. We specified settings for IP cores, including memory addresses, register spaces, and GPIO directions. There were two major modifications made to the system. The first involved changing the OTG USB port from host mode to peripheral mode, allowing for Ethernet over USB. The second modification was configuring remote process settings, which allocated one core for running Linux and the other for bare-metal operations. Additionally, multiple shared memory locations were reserved for the two cores to exchange data.

Reserving Memory

For the two asynchronous CPU cores to communicate and send sample data to each other, multiple reserved memory locations needed to be defined. In Petalinux, this involved carving out memory locations in the device tree to prevent the operating system from using that memory space. The following are the names of the device tree entries made in the reserved-memory node:

- vdev0vring0, vdev0vring1, vdev0buffer - Required for RPMsg protocol
- rproc_0_reserved - Required for remoteproc to launch the bare-metal app
- shared_memory - The bare metal app writes sample data to this memory region, and Petalinux reads from it when obtaining sample data

A similar process was done on the bare-metal app. Instead of using a device tree, memory locations, and sizes were defined in the sampling app as constants. These memory locations and sizes directly matched the entries in the Petalinux device tree.

Developing Applications

Because each CPU is running independently, they each need a custom-built application that will carry out their specific tasks and communicate with the other application when necessary. The application running on the Petalinux CPU is named CyDAQ_comm, or comm for short, and the other directly running on the other CPU in bare-metal mode is named CyDAQ_sampling, or sampling for short. Both applications were developed and built in Vitis, and their .elf files were incorporated into the Petalinux build. At runtime, Petalinux first runs the comm application, which then launches the sampling application on the second CPU.

The communication application(COMM) is responsible for communicating with the PC connected to the CyDAQ over USB. It listens on the serial interface for commands and processes them accordingly. Some commands, such as ping, only require a basic response. Other commands, however, require sending or requesting data from the other CPU. This communication is done using the OpenAMP RPMsg interfaces. Most commands that the comm application sends to the sampling application over RPMsg require a confirmation response from the sampling application. This confirmation makes diagnosing issues with the RPMsg protocol much easier. The COMM application is also responsible for reading from the shared memory location the other CPU previously wrote sample data to. It outputs this sample data to a serial port and writes it to a file. Our CLI implementation utilizes the file when it reads from the Petalinux filesystem using SCP to copy the data to the host PC.

The Sampling application (SAMP) runs on the remote processor core, specifically on CPU1 in our design. SAMP consists of two primary components: a bare-metal application that utilizes a series of C source codes to manage various design components such as ADC and GPIO modules and a remote process application that oversees RPMsg exchanges and sample data transfers with COMM. The remote process aspect of SAMP receives RPMsg from COMM, executes specific functions in the bare-metal application based on commands and based on the status of the bare-metal function call, and returns acknowledgment or error messages to SAMP.

The sampling application also handles writing sample data to a shared memory region whenever it is interrupted by the ADC hardware. This memory region is also defined in Petalinux, so the other CPU can read it when sampling is stopped. Because reading and writing data always happens sequentially, there is no need for concurrency mechanisms between the sampling memory region. This is a limitation of our implementation of reading and writing sampling data, as live streaming data between the two CPUs would require some concurrent mechanism.

Hardware

The prior Vivado hardware design did not utilize a board BSP, as it was not needed for a bare-metal design. However, incorporating a board BSP can simplify the Petalinux workflow, so we imported the Zybo Z7-10 version 2020.1 board BSP. To update the old Vivado project created in Vivado 2018, we used Vivado 2020.1 and recreated the project with the BSP. We added a GPIO IP for the LEDs and RGB LED on the zybo which lights up after linux is booted, this acts as an indicator for students in the lab to have an understanding of when the CyDAQ is available to be used. We also modified the XADC IP core (points to the onboard analog-to-digital converter) to output AXI-stream data and store it in a DMA (direct memory access). This modification is intended for custom-built applications requiring large-scale data access or streaming capabilities. For the overall design, see Appendix II Section 2.2.

GitLab Wiki

On our GitLab repository, we have a full step-by-step instruction guide on how to run, set up and test the GUI, CLI, and firmware of the CyDAQ project. On top of that, there is also other helpful information such as online guides, official documentation of the libraries used, issues that may be encountered when setting up the project, and how to fix them. The goal of having a detailed wiki is to ensure a smooth transition for the future senior design team that gets this project and catch them up to speed on what we did and our future plans.

Testing Process and Results

Manual Testing

Our testing was mostly done manually because the CyDAQ needed to be set up on a computer with other physical resources connected to be fully tested. To test each release, we ran through some of the labs, such as the Sampling lab, used in EE classes to ensure students could complete them without issues. Testing this way would give us potential bugs users would have when doing their labs. On top of going through labs, we also tested the CyDAQ by doing things that normally wouldn't be done when going through the lab. This would include repeatedly pushing buttons, setting filters and sample rates to values that wouldn't be used in the labs, setting values and sampling things out of order, sampling for periods of time that would never be asked for in labs, and anything else that we wouldn't expect users to do. The benefit of this was finding bugs we wouldn't find just by going through the standard labs. We found many bugs throughout our testing and were able to fix them accordingly to ensure that unexpected behavior from users wouldn't cause issues during students' labs.

Lab Testing

The CyDAQ's purpose is to be used in EE labs as a tool for students, so one of our testing methods was to put our CLI, GUI, and firmware in the labs for students to test and give us feedback. We put our changes in two different labs with eight sections and around 19 teams to test and give feedback. With the feedback we received from the students, we could consider their suggestions and fix the bugs they reported. Of the 18 teams, 17 preferred our new GUI, with the only complaints from the two other teams being bugs they ran into, which we have since fixed. Other feedback we received was about cosmetic and functional improvements we could make. Overall the feedback we received was primarily positive, and we took all their ideas and findings into consideration in our current design.

Petalinux USB Speed Testing

Transferring data between Petalinux and a USB-connected PC can be done most reliably over the emulated ethernet device created by the USB gadget in Petalinux. We verified that the transfer speed was high enough with a networking tool called iperf. The following steps were taken:

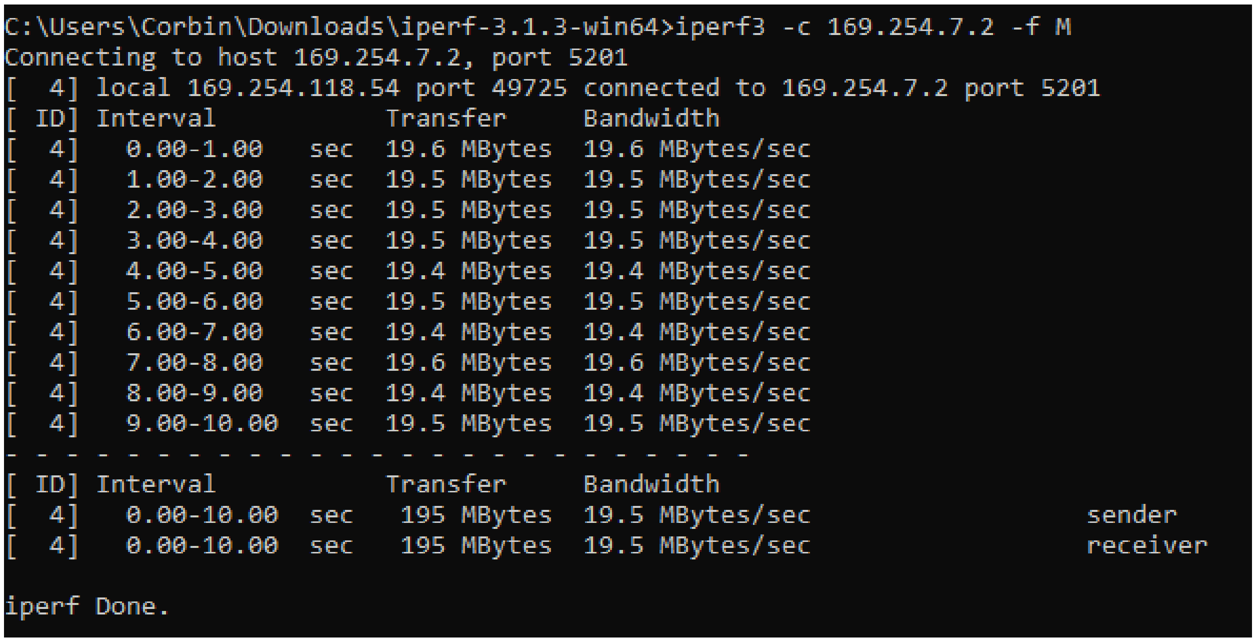
Install iperf on Petalinux

1. Download the iperf3 source code from their GitHub page and transfer it to the SD card with the Petalinux boot files on it
2. Plug the SD card into the CyDAQ and boot it
3. SSH into Petalinux
   ```
   ssh root@169.254.7.2
   ```
4. Transfer the source from the SD card to the home directory, unzip, and install

```
      cp /mnt/sd-mmcblk0p1/iperf-master.zip
/home/root/iperf-master.zip
      unzip /home/root/iperf-master.zip
      cd iperf-master
      ./configure && make && make install
```

5. On Petalinux, start the iperf3 server
   ```
   iperf3 -s
   ```
6. Install iperf3 on your host PC
7. On your host PC, start the test
   ```
   iperf3 -c 169.254.7.2 -f M
   ```

Below is an example output of the iperf3 test results:



```
C:\Users\Corbin\Downloads\iperf-3.1.3-win64>iperf3 -c 169.254.7.2 -f M
Connecting to host 169.254.7.2, port 5201
[  4] local 169.254.118.54 port 49725 connected to 169.254.7.2 port 5201
[ ID] Interval           Transfer     Bandwidth
[  4]   0.00-1.00   sec  19.6 MBytes  19.6 MBytes/sec
[  4]   1.00-2.00   sec  19.5 MBytes  19.5 MBytes/sec
[  4]   2.00-3.00   sec  19.5 MBytes  19.5 MBytes/sec
[  4]   3.00-4.00   sec  19.5 MBytes  19.5 MBytes/sec
[  4]   4.00-5.00   sec  19.4 MBytes  19.4 MBytes/sec
[  4]   5.00-6.00   sec  19.5 MBytes  19.5 MBytes/sec
[  4]   6.00-7.00   sec  19.4 MBytes  19.4 MBytes/sec
[  4]   7.00-8.00   sec  19.6 MBytes  19.6 MBytes/sec
[  4]   8.00-9.00   sec  19.4 MBytes  19.4 MBytes/sec
[  4]   9.00-10.00  sec  19.5 MBytes  19.5 MBytes/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth
[  4]   0.00-10.00  sec   195 MBytes  19.5 MBytes/sec                  sender
[  4]   0.00-10.00  sec   195 MBytes  19.5 MBytes/sec                  receiver

iperf Done.
```

Figure 6 - iperf Performance Output

The transfer speed observed over the emulated ethernet connection was 19.5 megabytes per second. This is well within our minimum requirement of 2 megasamples per second, which correlates to 4 megabytes per second.

Future Work

With the CyDAQ being an ongoing project between past and future senior design teams, more work can be done in the future to further meet any new or existing requirements. Completely changing the firmware to run on Petalinux opened many opportunities for future applications to be built on the CyDAQ. Throughout this project's design and development, we noted these ideas and outlined them below.

User Interface

Socket Communication

The existing GUI uses serial to communicate with the CyDAQ, which has some limitations. Sending and receiving serial messages in a way that can safely handle sudden disconnections, for example, requires a lot of overhead and was the source of many of our GUI bugs. This could be prevented with a socket-based connection established between the host PC and the Petalinux COMM application. Socket connections are designed to handle unstable connections much better than serial connections. On top of that, network connections introduce error correction in the IP layer, which is something that serial can't do.

Module Level Logging

Our current GUI uses the default Python logging library to handle debug messages sent to the console, shown in the debug screen, and written to a crash log. The current logging configuration adds all logs to a single module, which sometimes makes it difficult to filter out log messages for certain sections of the application. This could be improved upon by defining many different modules in the GUI. For example, each individual page in the GUI could have its own module, and then the debug page could have checkboxes that filter the modules based on what the user selects.

Web-Based

The user interface could be completely rewritten as a web app. Petalinux is capable of running any application that can run on other Linux distributions (assuming resource requirements are maintained), so it's safe to assume that it could run a web-based application that a connected PC could simply connect to for a user interface. This would greatly simplify the configuration process, as sending messages between the host PC and CyDAQ could be abstracted away by the web service chosen.

Careful consideration must also be made on which web service is chosen. The CyDAQ only has 1GB of RAM, some of which is reserved already. Petalinux also runs its entire root filesystem in RAM in our current configuration. Some web applications have high RAM usage, and Petalinux will not be able to function well if the usage is exceeded. More research into a lightweight web service would need to be done if this option is to be considered.

Firmware

Speed Up The Boot Time Of The Onboard Linux

The current boot time for the CyDAQ is around 20 seconds, which is longer than the full bare-metal project. The main reason is that Linux is being flashed to the RAM on every bootup and wiped on shutdown. There are a few possible solutions to this problem we came up with. These include changing to a faster SD card, disabling unused Linux features, and pre-loading the Linux system on the SD card. These have a few downsides, including the high cost of a premium SD card, extensive knowledge of Linux required, and the need to format and reload the firmware after every change.

DAC Mode For Audio Signal

There are two design paths for this. Using the Xilinx Quad SPI IP would be easy to implement but would require a fix to the problem of the core constantly resetting. Creating a custom SPI IP is another path that could be taken that would provide customizability but extensive knowledge of VHDL and Xilinx IPs is needed.

Linux Sandbox

The concept of Linux Sandbox involves a Python wrapper for the Vitis software, intending to enable students to utilize Python scripts to manage the CyDAQ. To achieve this, the sandbox must convert Python scripts into C programs compatible with the CyDAQ.

Open-source and DSP experiments

A long-term goal of this project is to open-source the CyDAQ so it can be given to other universities for use and to create different lab experiments on CyDAQ, especially for DSP applications.

Related Products and Literature

The CyDAQ started as a senior design project in 2018, and has since been the focus of multiple other senior design groups. The initial creation of the board was the focus of the first group, and subsequent projects have fixed bugs, implemented labs for the board, and attempted to re-implement the user interface.

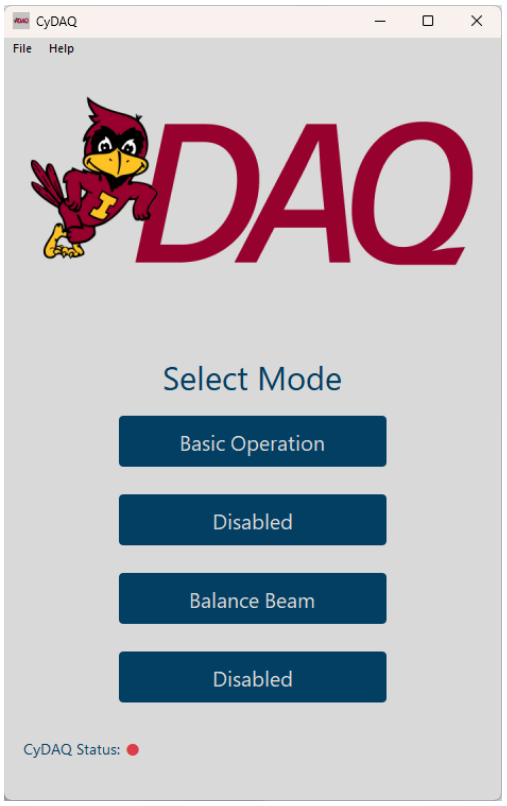
Some of the limitations of this project's context must also be considered. Because it is the product of multiple past senior design projects with little to no interaction between groups, the CyDAQ's software is very inconsistent. Electrical engineering staff have also modified certain code sections between senior design projects to patch bugs and add functionality. Some of these fixes were rushed and required fixing before adding new functionality between them.

One existing product the original senior design group pointed out in 2018 was National Instruments' myDAQ device. They concluded that using it for EE 224 labs instead of the CyDAQ wouldn't work well because it required knowledge of Python scripting and the creation of analog filtering circuits. Those two things didn’t align very well with the teaching goals of EE 224 and would thus require too much overhead for students to use. This was one of the big reasons the CyDAQ had a Windows-based GUI built for it, as a clickable GUI is much easier to use than learning a scripting language.

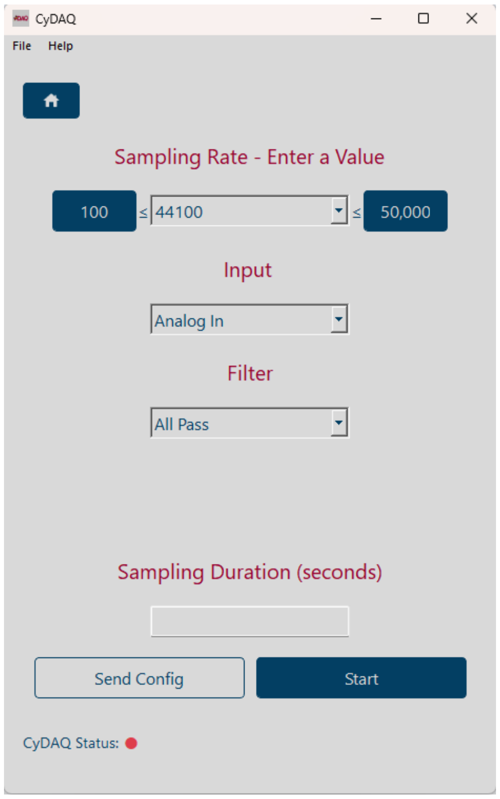
Appendix I - Operation Manual(s)

Section 1 - GUI Pages & Operation Manual

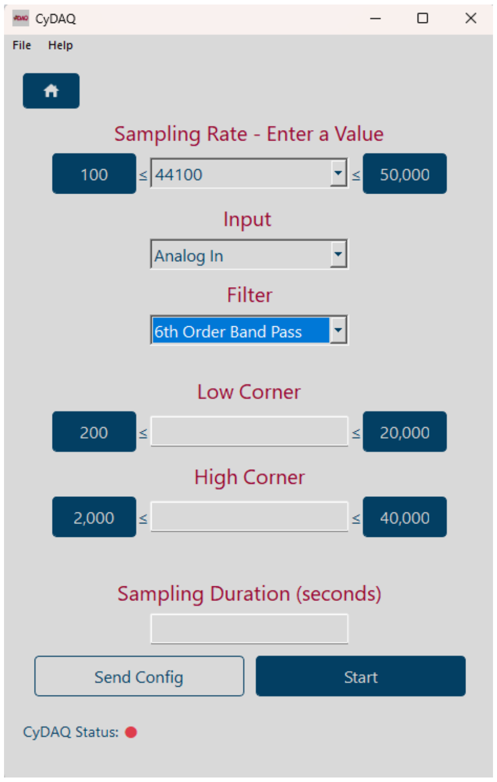
The GUI consists of different pages for each mode of the CyDAQ.



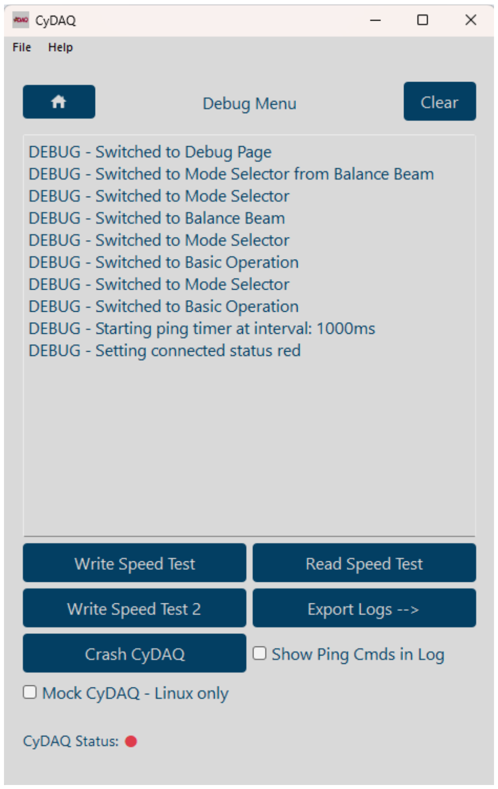
You will be greeted with the main page, the mode selector when running the app. It's pretty simple, just select one of the modes. Only two of them are enabled at the moment. There is a top menu bar where you can access the debug page. None of the other options work at the moment. There is a status indicator on the bottom of the page that is present on every other page in the GUI. This indicator tells the user if the CyDAQ is currently connected to the computer and GUI Wrapper underneath the hood.



This page allows you to collect samples from the CyDAQ with custom settings, including sample rate, input, and filters. You can start/stop sampling manually or set a specified time to sample, and there is a "Send Config" button for sending the configuration without sampling. The user interface prevents the use of invalid options, and a label at the bottom of the page indicates whether the CyDAQ is connected or not.

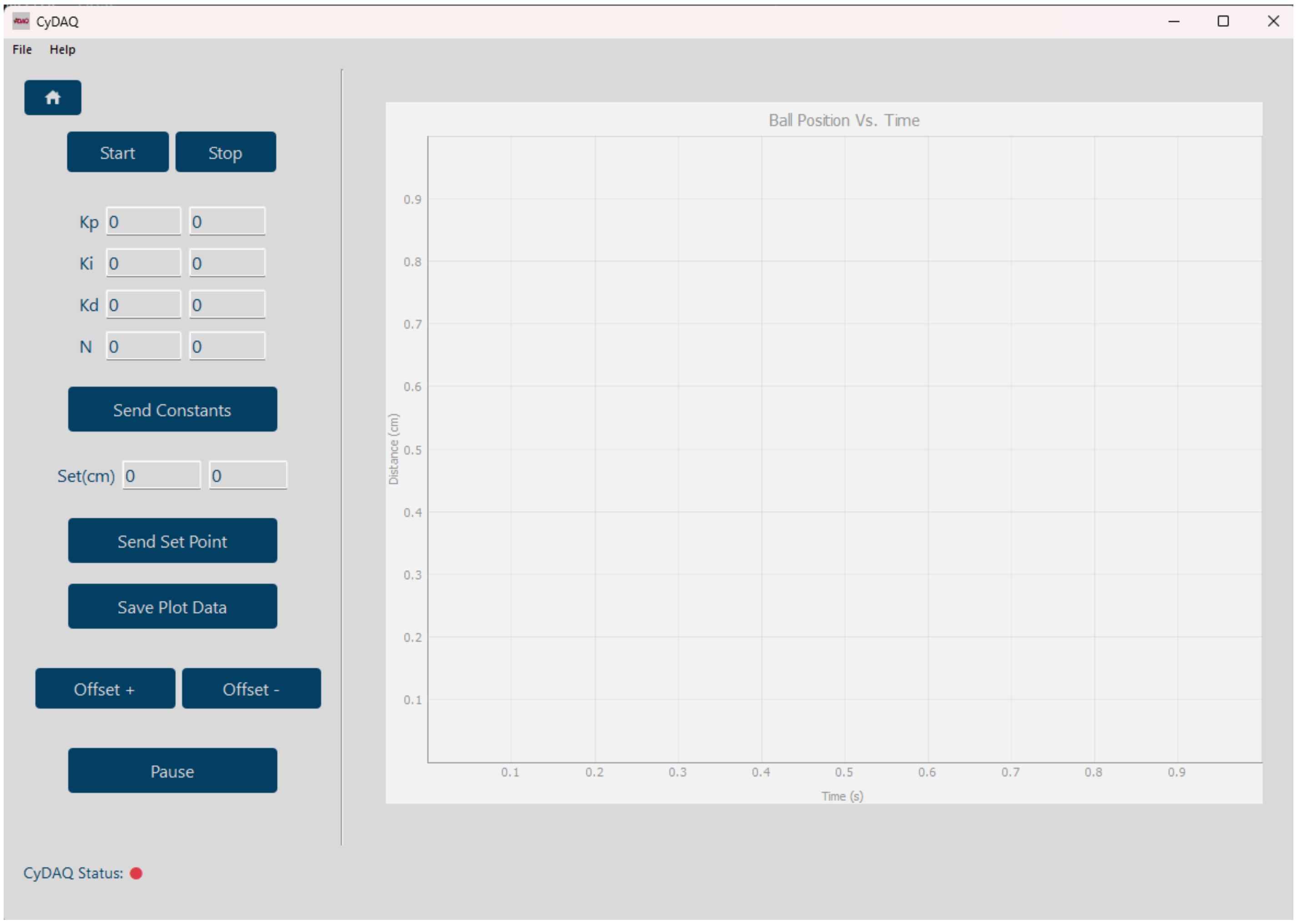


When you switch the filter, the appropriate corner field value shows up on the page where you can set the value. For some of the filters, they only need a Mid Corner and that field pops up on itself as well. This page can change as needed to fit the correct filter.



```
DEBUG - Switched to Debug Page
DEBUG - Switched to Mode Selector from Balance Beam
DEBUG - Switched to Mode Selector
DEBUG - Switched to Balance Beam
DEBUG - Switched to Mode Selector
DEBUG - Switched to Basic Operation
DEBUG - Switched to Mode Selector
DEBUG - Switched to Basic Operation
DEBUG - Starting ping timer at interval: 1000ms
DEBUG - Setting connected status red
```

The debug page in CyDAQ serves as a platform for running tests and obtaining diagnostic information. It contains a Debug Log View that captures debug logs at different levels, a Clear button to erase text view, and an Export Logs button to export current logs. Additionally, there are disabled "Test" buttons to measure python's speed and checkboxes for logging ping commands and enabling Mock Mode for Linux users.



The balance beam GUI is a redesigned interface intended to address issues that the previous version had, such as being unable to hotplug the CyDAQ and requiring frequent reconnection and restarts. The GUI has a button panel on the left side, with buttons that only work when the CyDAQ is connected and the balance beam is active. The start/stop buttons initiate the balance beam mode, while the constants, set point, and offset buttons allow for calibration and configuration of the balance beam. The GUI can plot data on the right side using the CLI wrapper, which sends a command to the CLI tool to get the ball's current position. However, there is no direct live connection to the CyDAQ at present.

Section 2 - Demo/Lab Setup

2.1 CyDAQ

1. Plug in the power cable to an outlet and the USB to the computer.
2. Connect any hardware that will be needed for the demo/lab (Balance Beam, DAD, etc.).
3. Turn on the CyDAQ with the power switch on the side.

2.2 GUI

1. Download the CyDAQ.exe.
2. Run the CyDAQ.exe.

Section 3 - CLI/GUI Development Setup

3.1 CyDAQ

1. Plug in the power cable to an outlet and the USB to the computer.
2. Connect any hardware needed for development (Balance Beam, DAD, etc.).
3. Turn on the CyDAQ with the power switch on the side.

3.2 CLI - Wiki

1. Clone the repo
   ```
   git clone https://git.ece.iastate.edu/sd/sdmay23-47.git
   ```
   - Change the address to whatever repo you're working in
2. Install the requirements from the root directory of the repo
   ```
   python -m pip install -r requirements.txt
   ```
3. Run the CLI for access to the CLI commands
   ```
   python cli/main.py
   ```
   - To get a list of all commands, type ‘h’ or ‘help’

3.3 GUI - Wiki

1. Clone the repo
   ```
   git clone https://git.ece.iastate.edu/sd/sdmay23-47.git
   ```
   - Change the address to whatever repo you're working in
2. Install the requirements from the root directory of the repo
   ```
   python -m pip install -r requirements.txt
   ```
3. Run the GUI application to use the CyDAQ
   ```
   python gui/app.py
   ```

3.3.1 QtDesigner 5

1. Download QtDesigner 5 from this link:
   - https://build-system.fman.io/qt-designer-download

Section 4 - Firmware Development Setup

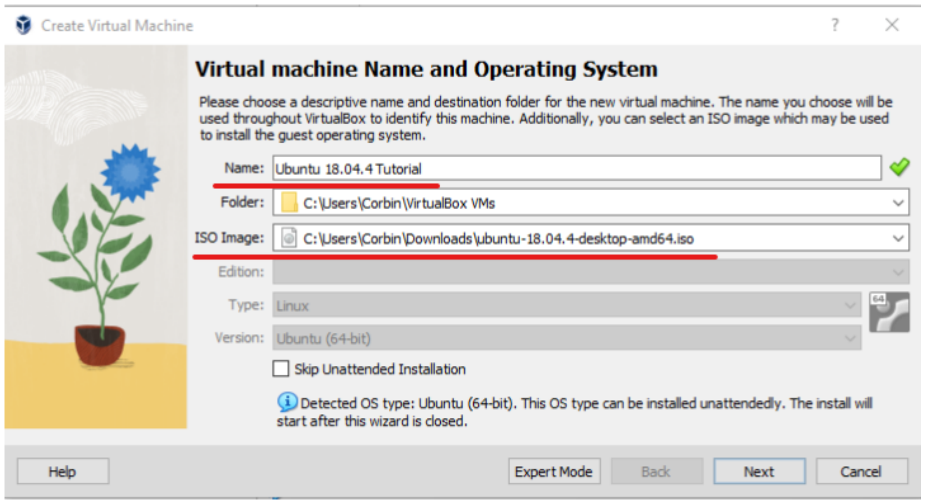
4.1 Petalinux Ubuntu Development Environment - Wiki

4.1.1 Download VirtualBox and .iso installer

- https://www.virtualbox.org/wiki/Downloads
- https://old-releases.ubuntu.com/releases/18.04.4/

4.1.2 Create a New Virtual Machine (VM) in VirtualBox

1. Click New
2. Give it a name and select your downloaded .iso file



Create Virtual Machine

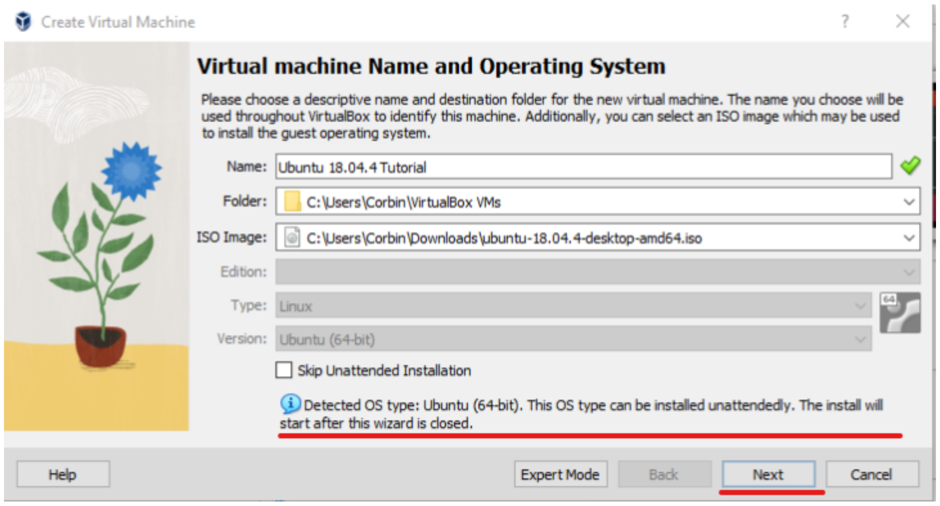
Virtual machine Name and Operating System

Please choose a descriptive name and destination folder for the new virtual machine. The name you choose will be used throughout VirtualBox to identify this machine. Additionally, you can select an ISO image which may be used to install the guest operating system.

Name: Ubuntu 18.04.4 Tutorial
Folder: C:\Users\Corbin\VirtualBox VMs
ISO Image: C:\Users\Corbin\Downloads\ubuntu-18.04.4-desktop-amd64.iso
Edition:
Type: Linux
Version: Ubuntu (64-bit)
Skip Unattended Installation

Detected OS type: Ubuntu (64-bit). This OS type can be installed unattendedly. The install will start after this wizard is closed.

3. Click Next and fill out the unattended guest OS setup



Create Virtual Machine

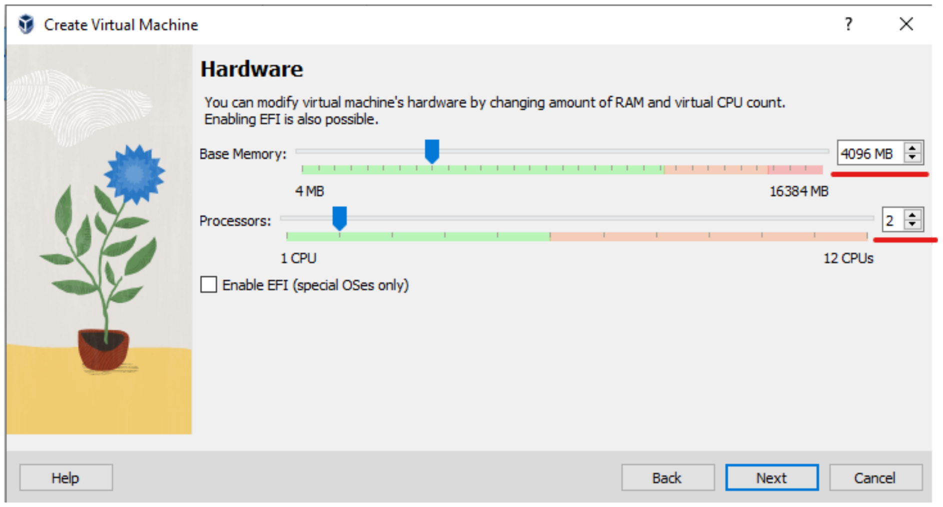
Virtual machine Name and Operating System

Please choose a descriptive name and destination folder for the new virtual machine. The name you choose will be used throughout VirtualBox to identify this machine. Additionally, you can select an ISO image which may be used to install the guest operating system.

Name: Ubuntu 18.04.4 Tutorial
Folder: C:\Users\Corbin\VirtualBox VMs
ISO Image: C:\Users\Corbin\Downloads\ubuntu-18.04.4-desktop-amd64.iso
Edition:
Type: Linux
Version: Ubuntu (64-bit)
Skip Unattended Installation

Detected OS type: Ubuntu (64-bit). This OS type can be installed unattendedly. The install will start after this wizard is closed.

4. Click Next and give it 4096MB (or more) RAM and 2 processors



Create Virtual Machine

Hardware

You can modify virtual machine's hardware by changing amount of RAM and virtual CPU count. Enabling EFI is also possible.

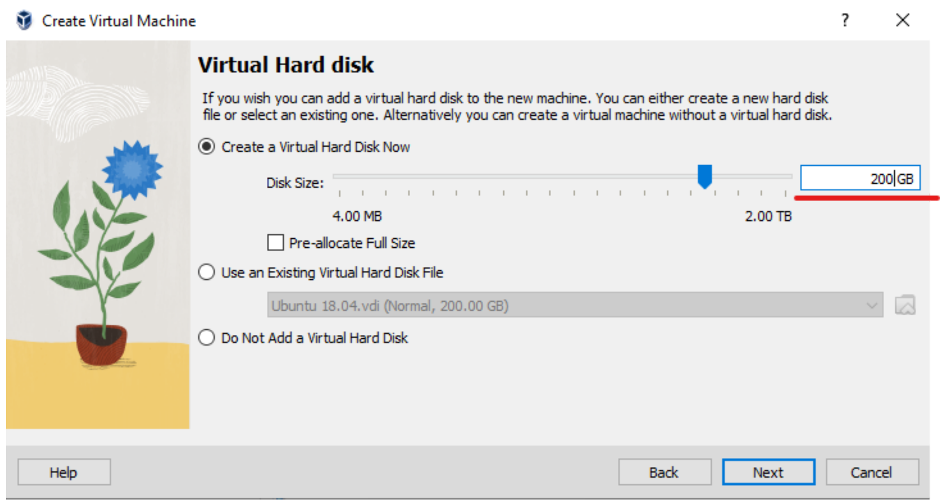
Base Memory: 4096 MB (4 MB – 16384 MB)

Processors: 2 (1 CPU – 12 CPUs)

☐ Enable EFI (special OSes only)

- If you can, we suggest giving it much more RAM. It depends on your system's capabilities.
- We experienced strange stuttering when the processor count wasn't set to 2.

5. Click Next and give it at least 200GB of hard disk space



Create Virtual Machine

Virtual Hard disk

If you wish you can add a virtual hard disk to the new machine. You can either create a new hard disk file or select an existing one. Alternatively you can create a virtual machine without a virtual hard disk.

● Create a Virtual Hard Disk Now

Disk Size: 200GB (4.00 MB – 2.00 TB)

☐ Pre-allocate Full Size

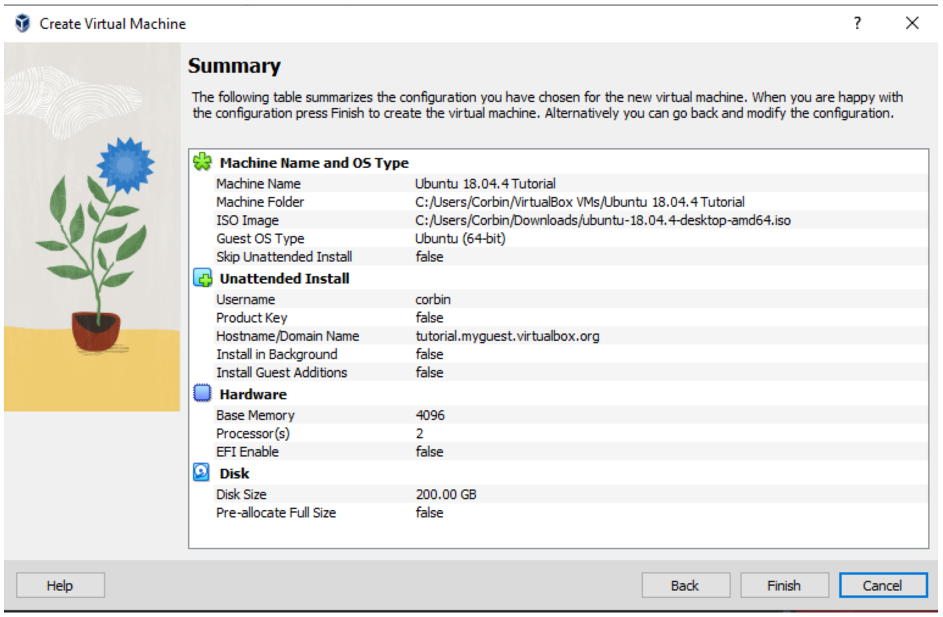
○ Use an Existing Virtual Hard Disk File

Ubuntu 18.04.vdi (Normal, 200.00 GB)

○ Do Not Add a Virtual Hard Disk

- Petalinux and Vivado/Vitis require a LOT of drive space. We don't suggest going below that for your drive.

6. Here is an example of the final configuration screen



Summary

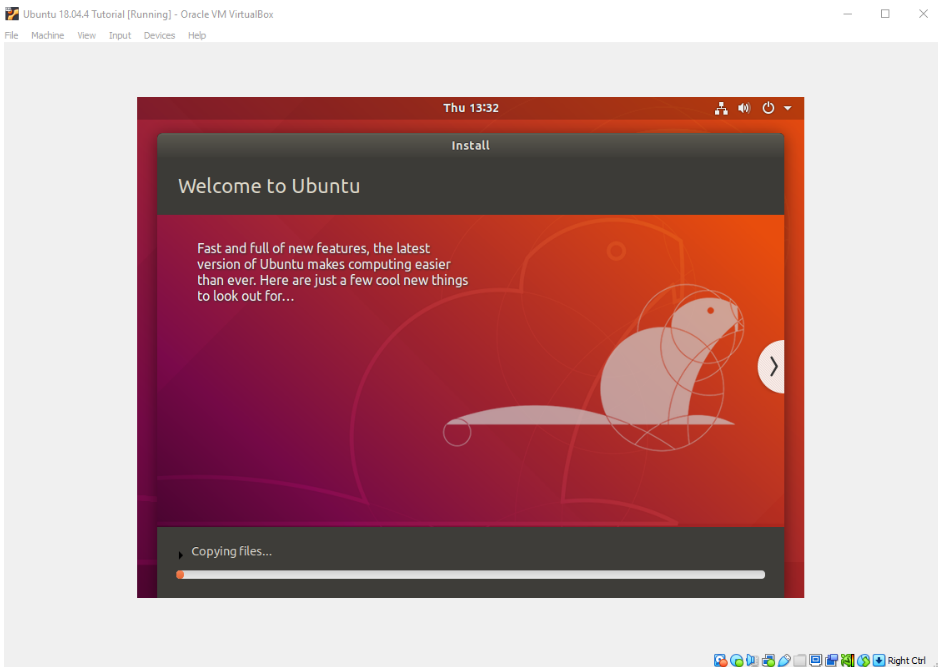
The following table summarizes the configuration you have chosen for the new virtual machine. When you are happy with the configuration press Finish to create the virtual machine. Alternatively you can go back and modify the configuration.

Machine Name and OS Type	
Machine Name	Ubuntu 18.04.4 Tutorial
Machine Folder	C:/Users/Corbin/VirtualBox VMs/Ubuntu 18.04.4 Tutorial
ISO Image	C:/Users/Corbin/Downloads/ubuntu-18.04.4-desktop-amd64.iso
Guest OS Type	Ubuntu (64-bit)
Skip Unattended Install	false
Unattended Install	
Username	corbin
Product Key	false
Hostname/Domain Name	tutorial.myguest.virtualbox.org
Install in Background	false
Install Guest Additions	false
Hardware	
Base Memory	4096
Processor(s)	2
EFI Enable	false
Disk	
Disk Size	200.00 GB
Pre-allocate Full Size	false

7. Click Finish

4.1.3 Set Up The Virtual Machine (VM)

1. Start your VM. You should be greeted with an install screen. It should install and configure your OS for you (unattended).



Welcome to Ubuntu

Fast and full of new features, the latest version of Ubuntu makes computing easier than ever. Here are just a few cool new things to look out for…

Copying files…

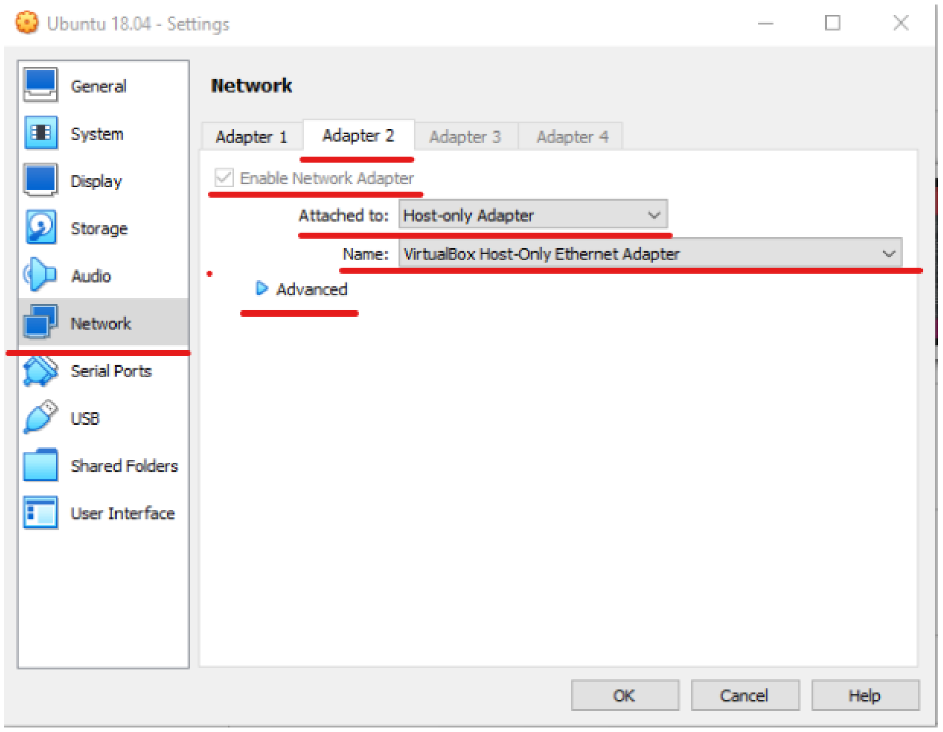
2. Add your user to the sudoers group using the following commands

```
su -
usermod -aG sudo <username>
```

3. Shut down the VM
4. In VirtualBox manager, navigate to the VM settings -> Network tab
5. Select Adapter 2 and enable the following

- Enable Network Adapter

- Attached to: Host-only Adapter
- Name: VirtualBox Host-Only Ethernet Adapter



Network

Enable Network Adapter

Attached to: Host-only Adapter

Name: VirtualBox Host-Only Ethernet Adapter

6. Under Advanced, take note of the MAC address (for a following step)
7. Click OK and power back on the VM
8. Run updates with the following command
   ```
   sudo apt update && sudo apt upgrade -y
   ```
9. Configure the ARP table to resolve the CyDAQs IP address to the correct interface
   ```
   sudo apt install net-tools
   ```
10. Ping the CyDAQ and wait for it to time out (to populate the ARP table)
   ```
   ping 169.254.7.2
   ```
- Ctrl + C to stop once you see a few error messages
   ```
   sudo arp -s 169.254.7.2 <mac address from earlier>
   ```
11. To verify its working correctly, connect the CyDAQ to the PC and ping
   ```
   ping 169.254.7.2
   ```

4.2 Vivado and Vitis Install - Wiki

The following steps assume you are installing this software on Ubuntu 18.04.4 VM

1. Navigate to the Xilinx Downloads Page and download the Xilinx Vitis 2020.1: All OS installer Single-File Download
- You may need to make a free XIlinx/AMD account to download the installer.
2. Extract the installer
   ```
   tar -xf Xilinx_Unified_2020.1_0602_1208.tar.gz
   ```
- This may take a while.
3. Make an install location
   ```
   sudo mkdir /tools && sudo mkdir /tools/Xilinx && sudo chmod 777
   /tools && sudo chmod 777 /tools/Xilinx
   ```
4. Run the installer. It will ask you to select a product. Choose 1. Vitis.

```
cd Xilinx_Unified_2020.1_0602_1208/
./xsetup --agree 3rdPartyEULA,XilinxEULA,WebTalkTerms --batch
Install --edition "Vitis Unified Software Platform" --location
/tools/Xilinx/
```

- This takes about 1 hour to install depending on individual VM performance.

5. You should now see Vitis, Vitis HLS, Vivado, and Documentation Navigation on your desktop.

4.3 Petalinux Tools Install - Wiki

The following steps assume you are installing this software on Ubuntu 18.04.4 VM

1. Download the Petalinux tools installer from the Xilinx Download page. Here is the one we picked.

- You may need to make a free XIlinx/AMD account to download the installer.

2. Enable i386 architecture packages
```
sudo dpkg --add-architecture i386
sudo apt-get update
```
3. Disable dash, select No when prompted
```
sudo dpkg-reconfigure dash
```
4. Install dependencies
```
sudo apt install iproute2 gcc g++ net-tools libncurses5-dev
zlib1g:i386 libssl-dev flex bison libselinux1 xterm autoconf
libtool texinfo zlib1g-dev gcc-multilib build-essential screen
pax gawk python3 python3-pexpect python3-pip python3-git
python3-jinja2 -y
```
5. Make the install directory location
```
sudo mkdir -p /opt/pkg/Petalinux/2020.1
```
6. Make the install directory accessible
```
sudo chmod -R u=rwx,g=rwx,o=rwx /opt/pkg/Petalinux/2020.1/
sudo chown -R <user> /opt/pkg/Petalinux/2020.1/
```
7. Navigate to the installer and add execution privilege
```
chmod +x Petalinux-v2020.1-final-installer.run
```
8. Run the installer, follow the instructions to accept the license agreements
```
./Petalinux-v2020.1-final-installer.run -d
/opt/pkg/Petalinux/2020.1
```

- "WARNING: This is not a supported OS" can be ignored.
- "WARNING: No tftp server found - please refer to "UG1144 Petalinux Tools Documentation Reference Guide" for its impact and solution" can be ignored if you are not using a tftp server.

9. Source the Petalinux tools
```
source /opt/pkg/Petalinux/2020.1/settings.sh
```

- This last command needs to be ran every time you start a console session. It allows you to run multiple Petalinux tools versions and switch between them if you want.

4.4 Firmware Repo Setup

The following steps are assuming you have completed all previous setup steps and are working inside your Ubuntu VM.

4.4.1 Clone the Repo

1. Change into the correct directory.
```
cd ~
git clone https://git.ece.iastate.edu/sd/sdmay23-47.git
```
- Change the address to whatever repo your working in

4.4.2 Build Petalinux

1. Change into the correct directory.
```
cd ~/sdmay23-47/Petalinux/os
```
2. Soure Petalinux if you haven't already.
```
source /opt/pkg/Petalinux/2020.1/settings.sh
```
3. Set Petalinux to build using our custom hardware.
```
Petalinux-config --get-hw-description ../hw
```
4. Build Petalinux.
```
Petalinux-build
```
- This will take 1+ hour to run the first time.
5. Build the bootable image (optional but useful).
```
Petalinux-package --force --boot --fsbl
images/linux/zynq_fsbl.elf --fpga ../hw/CyDAQ_proto_1.bit
--u-boot
```
- The build files are placed in <repo root>/Petalinux/os/images/linux/ and you must move boot.scr, BOOT.BIN, and image.ub to the SD card.

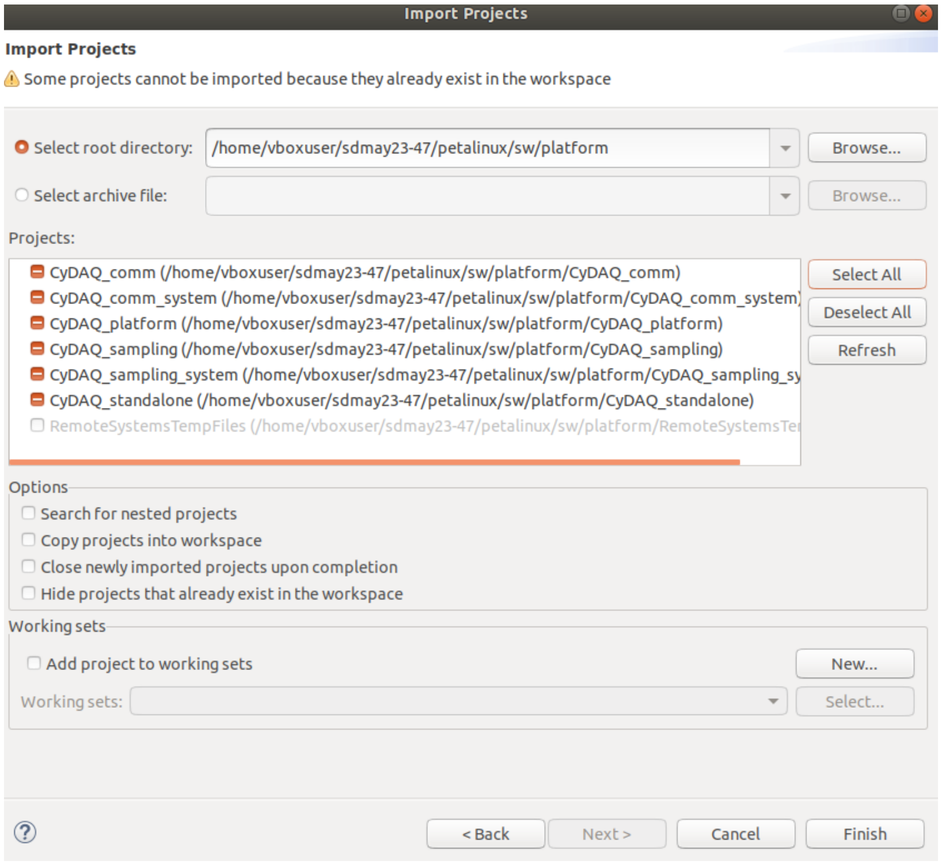
4.4.3 Build Petalinux SDK

1. Change into the correct directory.
```
cd <project root>/Petalinux/os/images/linux
```
2. Build the SDK.
```
Petalinux-build --sdk
./sdk.sh
```
3. This will ask you to enter a target directory. Enter the following:
```
<root project>/Petalinux/sw/linux_files/
```
4. Copy the necessary files from the Petalinux build to the sw directory
```
cp u-boot.elf ../../../sw/linux_files/boot/ && cp zynq_fsbl.elf
../../../sw/linux_files/boot/fsbl.elf && cp boot.scr image.ub
rootfs.cpio ../../../sw/linux_files/image/
```

4.4.4 Vitis Setup

1. Launch Vitis and set the workspace to the following:
```
<root project>/Petalinux/sw/platform/
```
2. You should now be greeted with a welcome screen and... nothing else.

3. Import the projects. Go to file > import
4. Select Eclipse workspace or zip file and click next
5. Click browse next to Select root directory and select ~/sdmay23-47/Petalinux/sw/platform
6. It should already be on that directory
7. Under options, unselect Copy projects into workspace
8. Under projects (on the right) click Select All
9. Your screen should look like the following:



Import Projects

Some projects cannot be imported because they already exist in the workspace

Select root directory: /home/vboxuser/sdmay23-47/petalinux/sw/platform

Select archive file:

Projects:

- [✓] CyDAQ_comm (/home/vboxuser/sdmay23-47/petalinux/sw/platform/CyDAQ_comm)
- [✓] CyDAQ_comm_system (/home/vboxuser/sdmay23-47/petalinux/sw/platform/CyDAQ_comm_system)
- [✓] CyDAQ_platform (/home/vboxuser/sdmay23-47/petalinux/sw/platform/CyDAQ_platform)
- [✓] CyDAQ_sampling (/home/vboxuser/sdmay23-47/petalinux/sw/platform/CyDAQ_sampling)
- [✓] CyDAQ_sampling_system (/home/vboxuser/sdmay23-47/petalinux/sw/platform/CyDAQ_sampling_sy
- [✓] CyDAQ_standalone (/home/vboxuser/sdmay23-47/petalinux/sw/platform/CyDAQ_standalone)
- [ ] RemoteSystemsTempFiles (/home/vboxuser/sdmay23-47/petalinux/sw/platform/RemoteSystemsTe

Options

- [ ] Search for nested projects
- [ ] Copy projects into workspace
- [ ] Close newly imported projects upon completion
- [ ] Hide projects that already exist in the workspace

Working sets

- [ ] Add project to working sets

Working sets:

10. Click Finish
11. You should now see four projects in your project explorer:
- CyDAQ_comm_system
- CyDAQ_platform
- CyDAQ_sampling_system
- CyDAQ_standalone

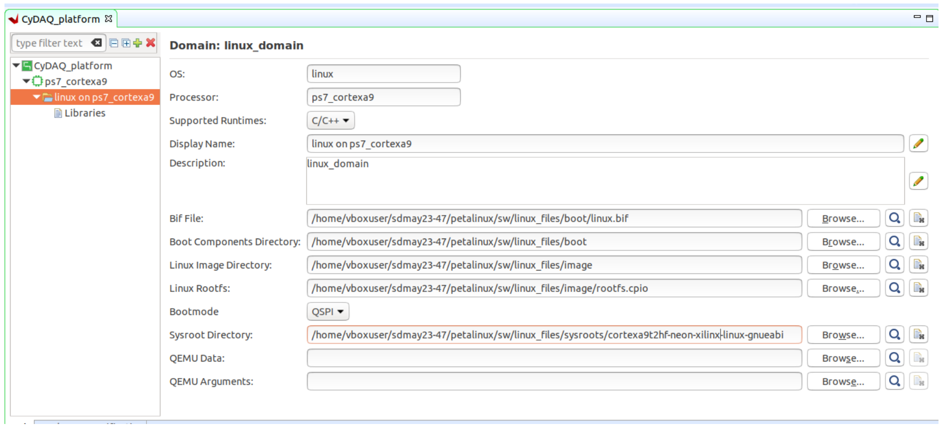
4.5 Issues and Fixes in Vitis

Unfortunately, Xilinx is very bad at making their products work with Git, so the following steps need to be taken to make sure everything in the project is pointing to your home directory, not the last person who committed to the repo.

You may have to do these steps when checking out somebody else's branch or after somebody else commits. Yes, it's quite annoying and time-consuming. Trust us, we know.

4.5.1 Fixes for CyDAQ_platform

1. Expand CyDAQ_platform, open platform.spr
2. Click on Linux on ps7_cortexa9 in the platform viewer.
3. Here, you will see a few directories and paths that need updating to your correct paths. They should all point somewhere inside of ~/sdmay23-47/Petalinux/sw/linux_files, and there will be existing paths there, but just with the wrong home directories.
4. Click browse and select the correct path locations for each.
5. Below is an example of a proper configuration:



Domain: linux_domain

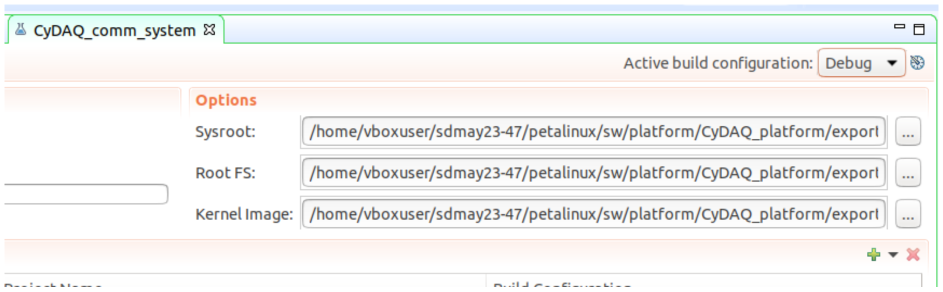
OS:	linux
Processor:	ps7_cortexa9
Supported Runtimes:	C/C++
Display Name:	linux on ps7_cortexa9
Description:	linux_domain
Bif File:	/home/vboxuser/sdmay23-47/petalinux/sw/linux_files/boot/linux.bif
Boot Components Directory:	/home/vboxuser/sdmay23-47/petalinux/sw/linux_files/boot
Linux Image Directory:	/home/vboxuser/sdmay23-47/petalinux/sw/linux_files/image
Linux Rootfs:	/home/vboxuser/sdmay23-47/petalinux/sw/linux_files/image/rootfs.cpio
Bootmode	QSPI
Sysroot Directory:	/home/vboxuser/sdmay23-47/petalinux/sw/linux_files/sysroots/cortexa9t2hf-neon-xilinx-linux-gnueabi
QEMU Data:	
QEMU Arguments:	

- For the Sysroot Directory field, sometimes an error will pop up when you set it. This is a Vitis bug, and setting it a second time to the same path should work.

6. Right-click CyDAQ_platform in the explorer view and select Update Hardware Specification. Click Browse and navigate to ~/sdmay23-47/Petalinux/hw/CyDAQ_proto_1.xsa, then click Ok.
7. You should now be able to build CyDAQ_platform. Simply right-click on CyDAQ_platform in the explorer and click build.

4.5.2 Fixes for CyDAQ_comm_system

1. Expand CyDAQ_comm_system and open CyDAQ_comm_system.sprj
2. On the right, you should see an options section. Sysroot, Root FS, and Kernel Image all need their paths changed to your virtual machine's user home directory. You can manually change the file path in this window.
3. Here is an example:



Active build configuration: Debug

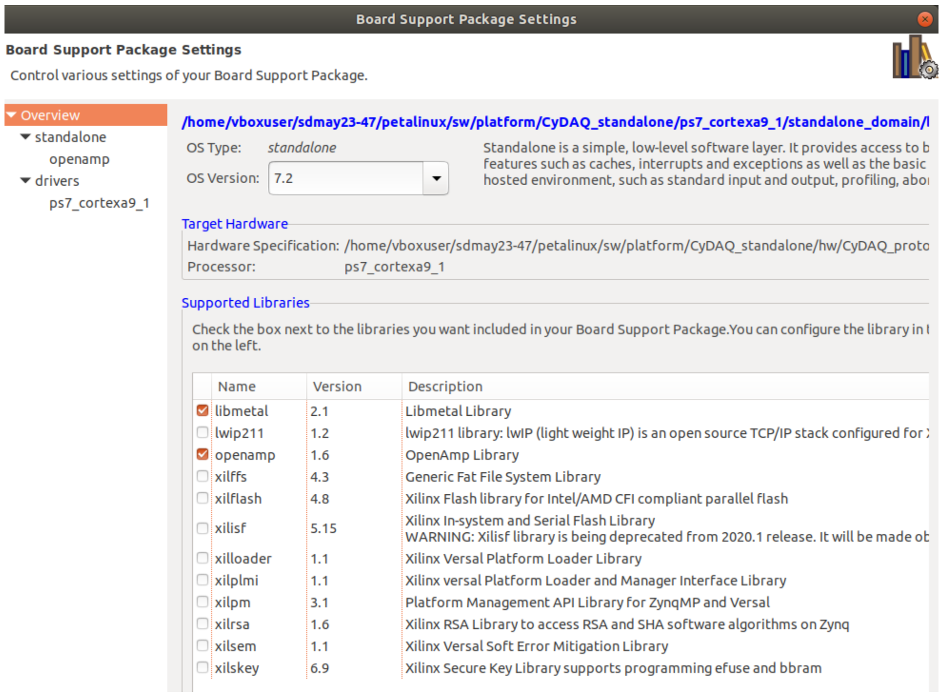
Options

Sysroot:	/home/vboxuser/sdmay23-47/petalinux/sw/platform/CyDAQ_platform/export
Root FS:	/home/vboxuser/sdmay23-47/petalinux/sw/platform/CyDAQ_platform/export
Kernel Image:	/home/vboxuser/sdmay23-47/petalinux/sw/platform/CyDAQ_platform/export

4. Replace vboxuser with your VMs username
5. You can now build the comm app. Right-click on CyDAQ_comm_system in the explorer view and click build.

4.5.3 Fixes for CyDAQ_sampling_system

1. In the explorer view, expand CyDAQ-sampling_system, then expand CyDAQ_sampling.
2. Open the file CyDAQ_sampling.prj
3. In the window that opens, click on Navigate to BSP Settings.
4. You should now be in the settings for CyDAQ_standalone. In that window, click on Modify BSP Settings. You should be greeted with a window that looks like the following:



Board Support Package Settings

Board Support Package Settings

Control various settings of your Board Support Package.

- Overview
  - standalone
    - openamp
  - drivers
    - ps7_cortexa9_1

/home/vboxuser/sdmay23-47/petalinux/sw/platform/CyDAQ_standalone/ps7_cortexa9_1/standalone_domain/

OS Type: standalone

OS Version: 7.2

Standalone is a simple, low-level software layer. It provides access to b features such as caches, interrupts and exceptions as well as the basic hosted environment, such as standard input and output, profiling, abo

Target Hardware

Hardware Specification: /home/vboxuser/sdmay23-47/petalinux/sw/platform/CyDAQ_standalone/hw/CyDAQ_proto

Processor: ps7_cortexa9_1

Supported Libraries

Check the box next to the libraries you want included in your Board Support Package.You can configure the library in t on the left.

	Name	Version	Description
✓	libmetal	2.1	Libmetal Library
☐	lwip211	1.2	lwip211 library: lwIP (light weight IP) is an open source TCP/IP stack configured for
✓	openamp	1.6	OpenAmp Library
☐	xilffs	4.3	Generic Fat File System Library
☐	xilflash	4.8	Xilinx Flash library for Intel/AMD CFI compliant parallel flash
☐	xilisf	5.15	Xilinx In-system and Serial Flash Library WARNING: Xilisf library is being deprecated from 2020.1 release. It will be made ob
☐	xilloader	1.1	Xilinx Versal Platform Loader Library
☐	xilplmi	1.1	Xilinx versal Platform Loader and Manager Interface Library
☐	xilpm	3.1	Platform Management API Library for ZynqMP and Versal
☐	xilrsa	1.6	Xilinx RSA Library to access RSA and SHA software algorithms on Zynq
☐	xilsem	1.1	Xilinx Versal Soft Error Mitigation Library
☐	xilskey	6.9	Xilinx Secure Key Library supports programming efuse and bbram

5. Uncheck libmetal and openamp, then click Ok.
6. Click Modify BSP Settings, re-check libmetal and openamp, and click Ok.
- It does seem strange to uncheck and recheck these libraries, but their paths get messed up between different users. This is the only way we could get things to work consistently.
7. Now, build standalone. Right-click on CyDAQ_standalone in the explorer view, and click Build.
8. It's a good idea to clean the sampling app at this point, as sometimes Vitis creates random errors otherwise. Click on CyDAQ_sampling_system in the explorer and click clean.
9. You can now build the sampling app. Right-click on CyDAQ_sampling_system and click build.

4.5.4 Include Applications in Petalinux build

1. Copy the .elf files generated from comm and sampling projects into the Petalinux botscript recipe folder, so they get included the next time you build Petalinux.
```
cp
~/sdmay23-47/Petalinux/sw/platform/CyDAQ_comm/Debug/CyDAQ_comm.
elf
~/sdmay23-47/Petalinux/os/project-spec/meta-user/recipes-apps/b
ootscript/files/CyDAQ_comm.elf && cp
~/sdmay23-47/Petalinux/sw/platform/CyDAQ_sampling/Debug/CyDAQ_s
ampling.elf
```

```
~/sdmay23-47/Petalinux/os/project-spec/meta-user/recipes-apps/b
ootscript/files/CyDAQ_sampling.elf
```

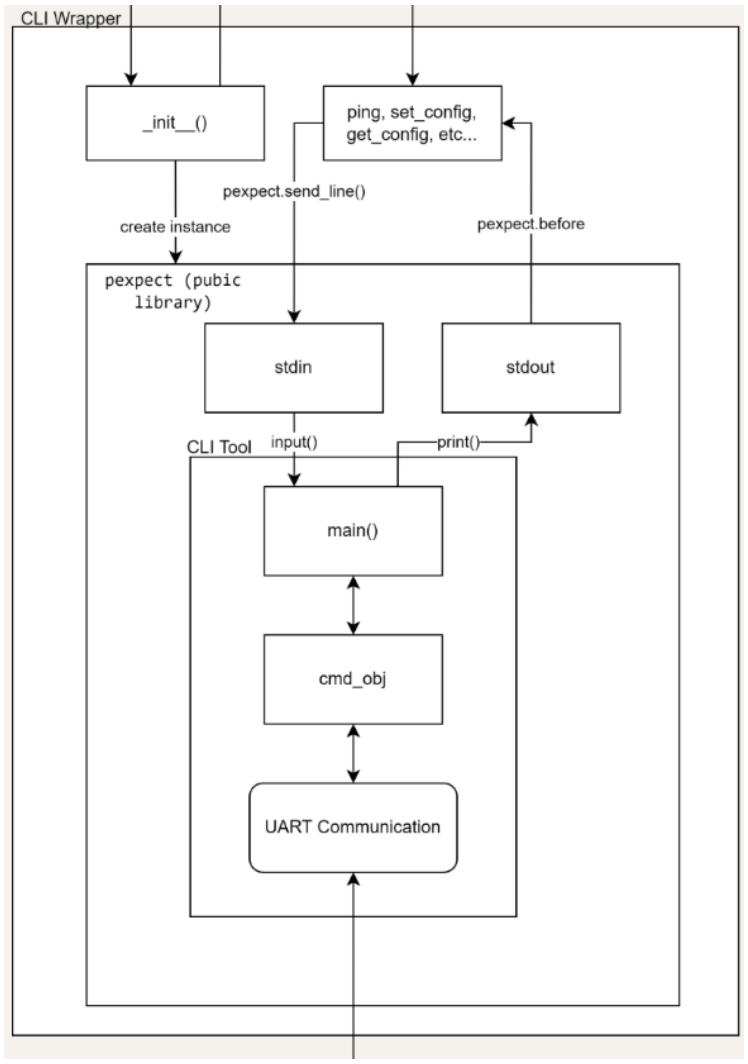
2. Rebuild Petalinux

```
source /opt/pkg/Petalinux/2020.1/settings.sh
cd ~/sdmay23-47/Petalinux/os && Petalinux-build
```

Appendix II - Figures

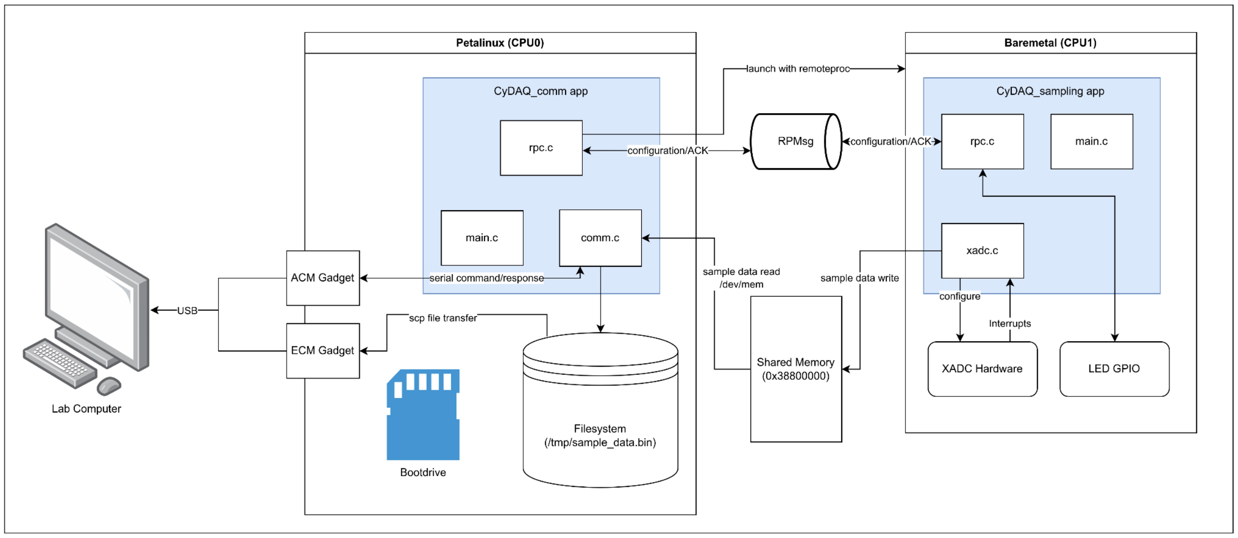
Section 1 - CLI

1.1 CLI Wrapper Graphical Overview

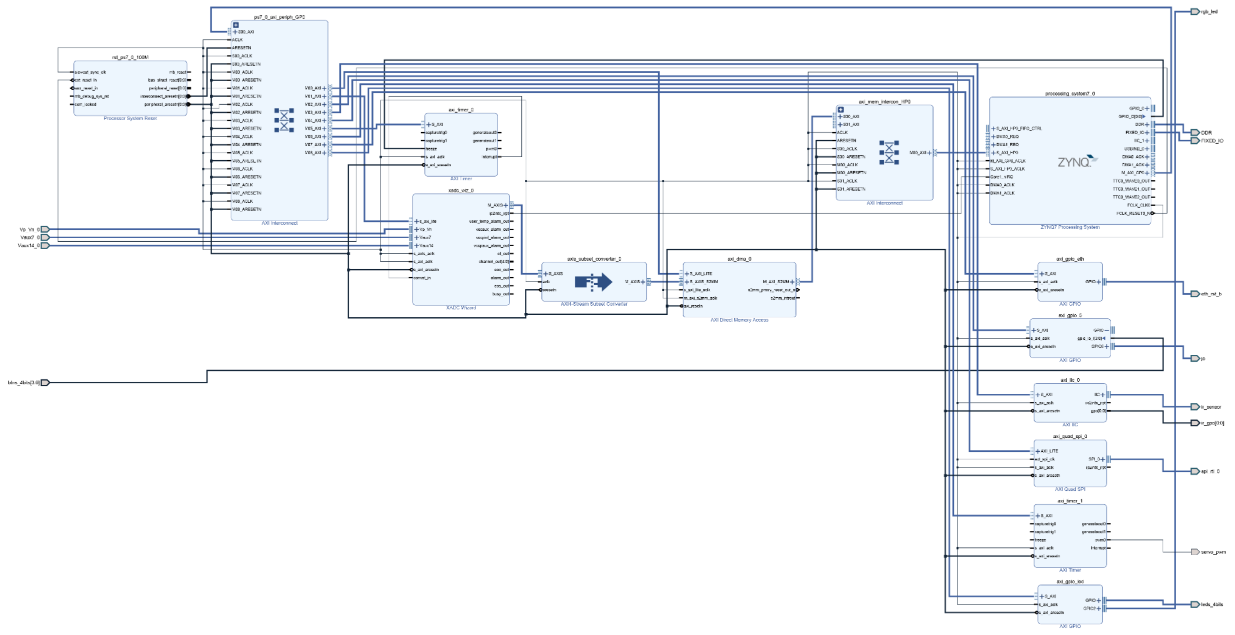


Section 2 - Firmware

2.1 Overall Firmware Design - Wiki

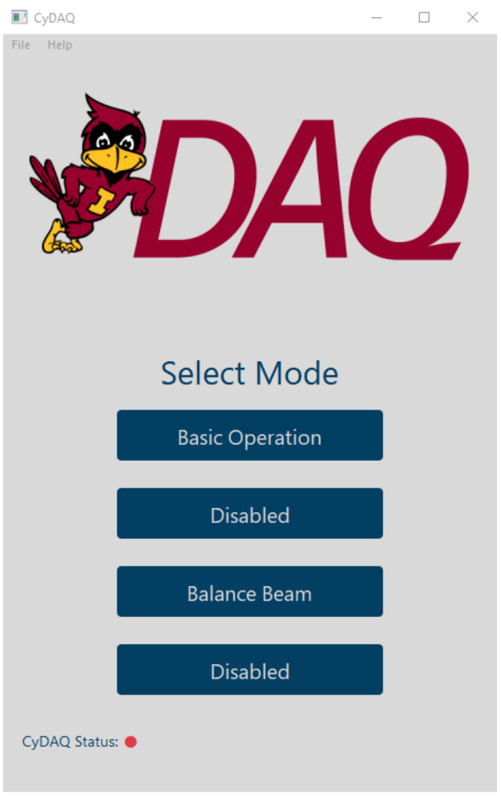


2.2 Vivado Hardware Design - Wiki

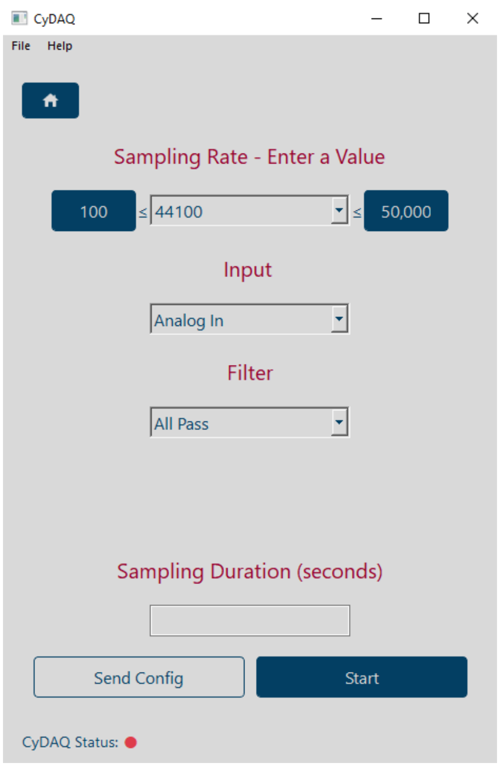


SECTION 3 - GUI - WIKI

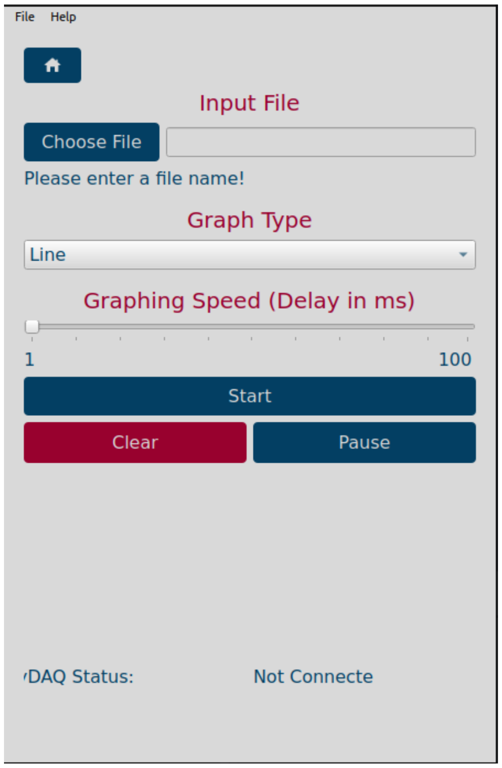
3.1 Mode Selector



3.2 Basic Operation



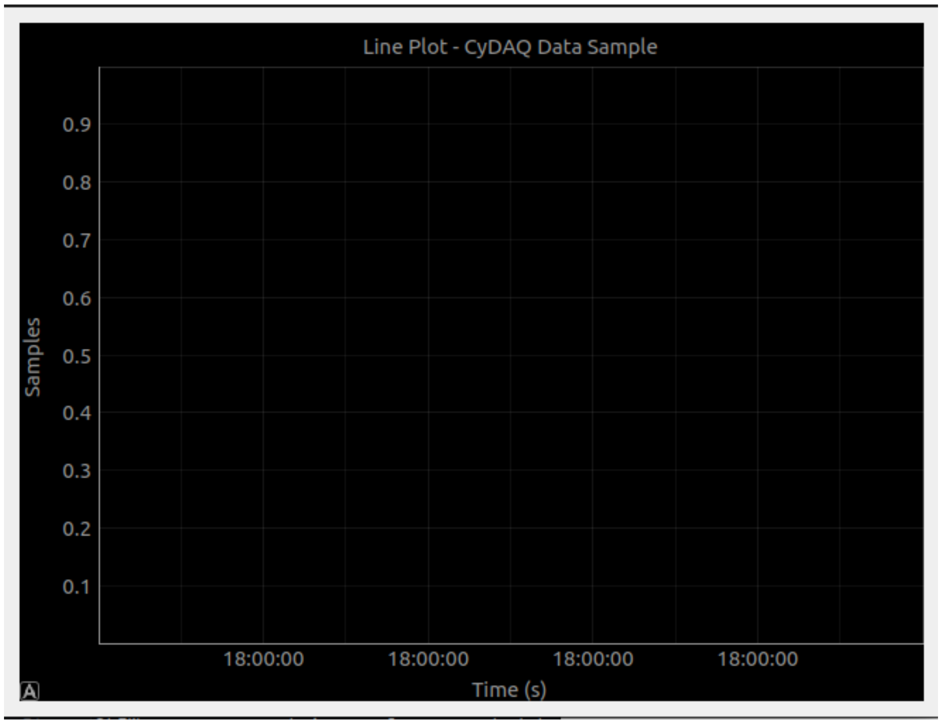
3.3 Plotter

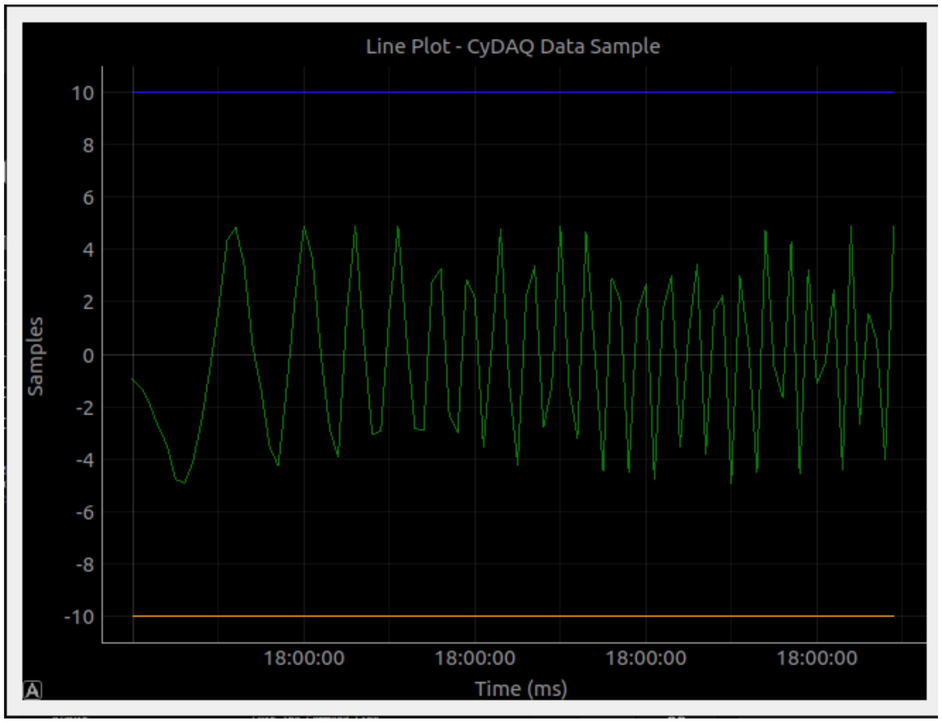


Please enter a file name!

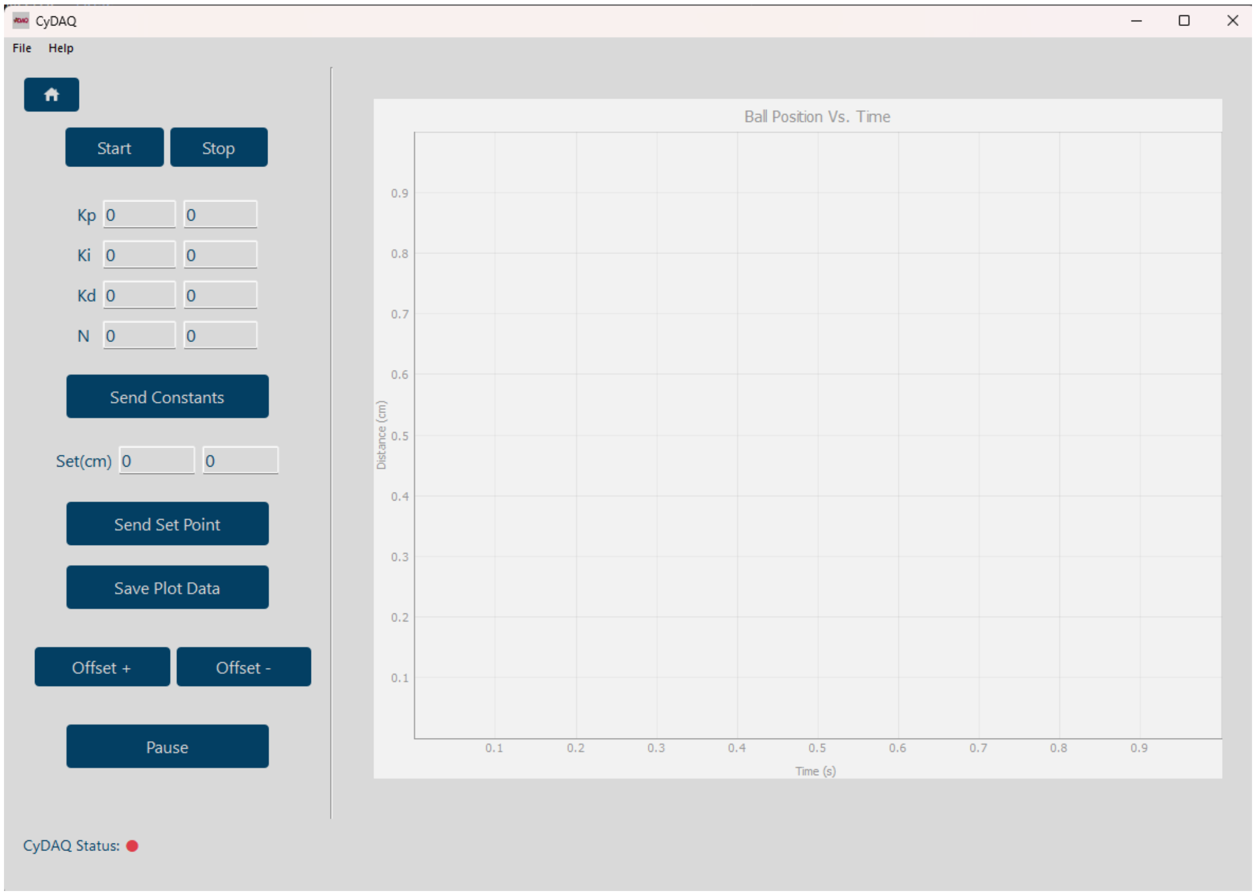
Graph Type: Line

DAQ Status: Not Connecte

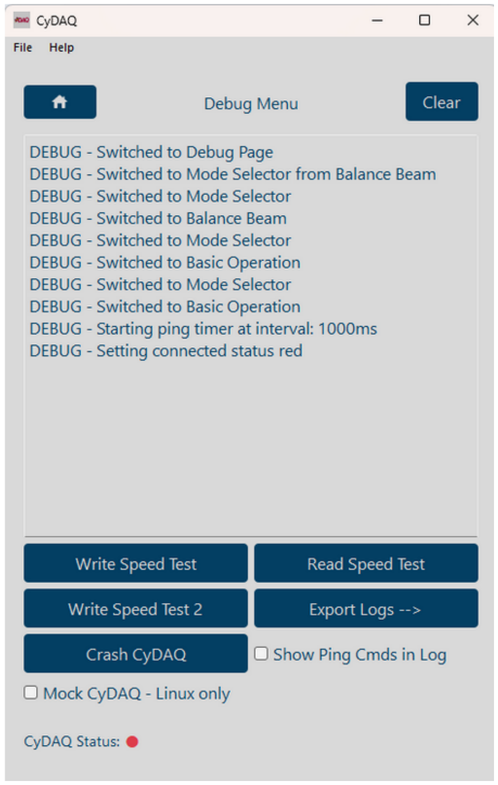




3.4 Balance Beam



3.5 Debug



```
DEBUG - Switched to Debug Page
DEBUG - Switched to Mode Selector from Balance Beam
DEBUG - Switched to Mode Selector
DEBUG - Switched to Balance Beam
DEBUG - Switched to Mode Selector
DEBUG - Switched to Basic Operation
DEBUG - Switched to Mode Selector
DEBUG - Switched to Basic Operation
DEBUG - Starting ping timer at interval: 1000ms
DEBUG - Setting connected status red
```

Appendix III - Alternative Designs

Bare-metal Dual-core Design - Wiki

This is the first version of the design for this project. The plan was to make two standalone bare-metal C projects. Each runs on a CPU core and communicates with shared memory space. The sample data is transferred using a Xilinx CDC USB over the UART library. This design was eventually deprecated because the experimental speed of USB over UART could not meet the requirement of this project.

Appendix IV - Code

Please see our GitLab.