Sensor & GUI Manual

Amir Hadi Hosseinabadi, David Black

Table of Contents:

Introduction

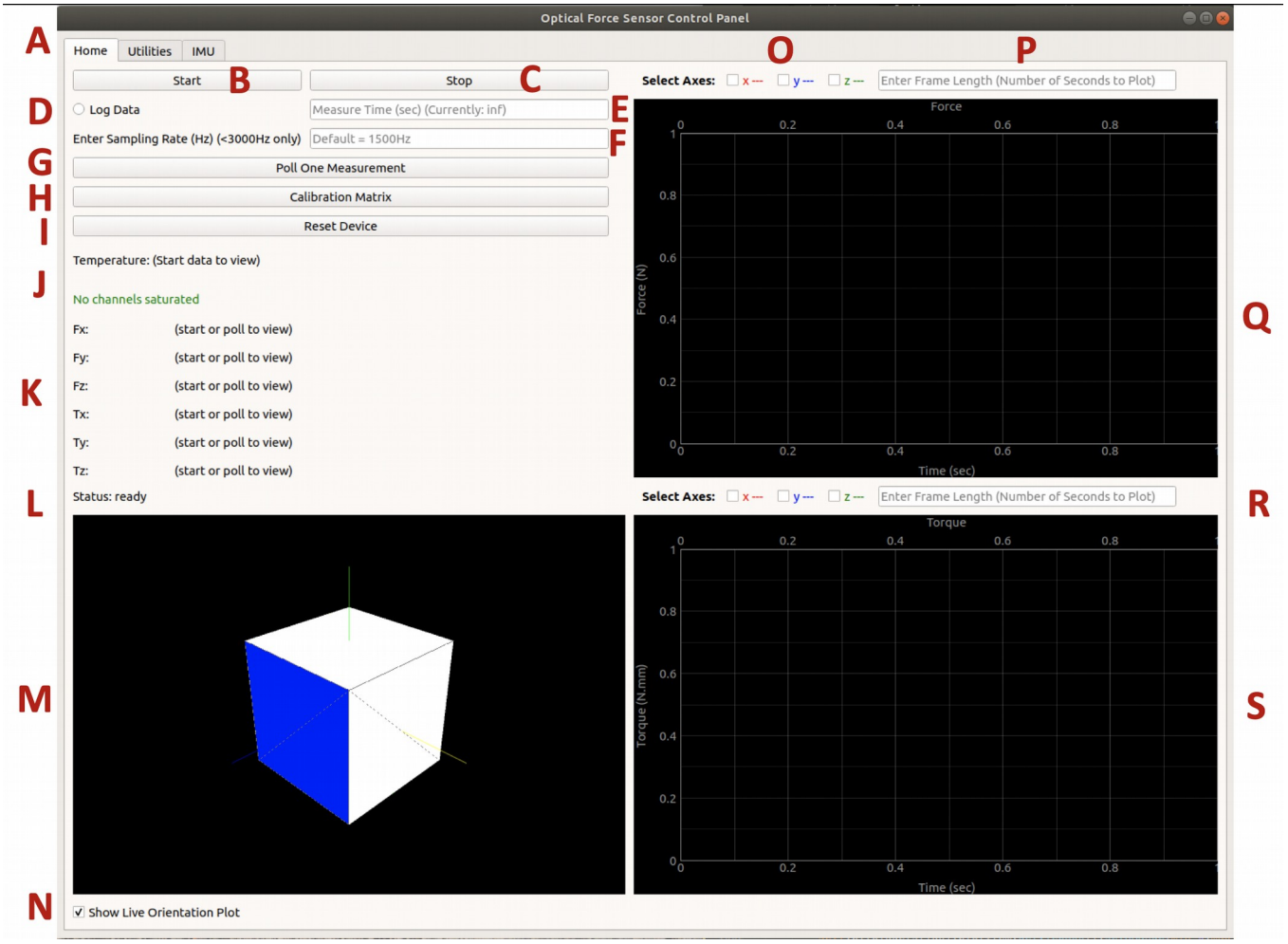
This sensor was developed for use in telerobotic haptic interfaces. Specifically, it was designed to be mounted on the shaft of the Patient Side Manipulator (PSM) of the da Vinci surgical robot and to measure the forces being applied at the tissue. The ability to sense forces at the PSM without having to dispose of the sensor every 10 uses is a major factor holding back the implementation of 4-channel teleoperation on the Intuitive da Vinci robot. With the novel method of optical force sensing used in this device, it can be mounted on the PSM and provide highly precise, 6-axis forces and torques in real time. The sensor is sterilizable, and when the PSM end-effector is replaced, the sensor can simply be removed and remounted onto the new end-effector. Furthermore, it can provide temperature, accelerometer and rotation vector readings for temperature correction, 4-channel impedance teleoperation, and orientation feedback respectively.

The sensor operates using 6 ADS1257 transducers. Each has an infrared LED in a different orientation, irradiating a bicell photodiode through a thin, collimating slit. Deflections in the shaft on which the sensor is mounted lead to relative displacements of the LED and sensors, so the amount of light on each changes. This can be used to calculate the deflection of the shaft, and thus the forces and torques being applied to it in all 6 axes. Due to the averaging effects of the large number of incident photons, as well as onboard signal conditioning electronics, the sensor is resistant to noise and can provide extremely precise measurements of down to nanometer-scale deflections.

The transducers sample their photodiodes at a high rate (up to 30kSPS) and using signal conditioning like moving average filtering to eliminate noise. The resultant voltage signals are in the form of a difference signal and a sum signal, which are converted to bytes and sent to the computer through an RS-485 to USB interface at up to 3kHz. In addition, the IMU reading and temperature are sent. The communication is controlled through the use of unique start sequences for each data packet, and Cyclic Redundancy Check error-detecting codes. The low-level communication is carried out on the host Linux PC by pylibftdi, a fast, light Python wrapper for the popular libftdi device driver C library. A software package is built on top of this to handle and parse the messages, and send commands to the sensor. Finally, a graphical user interface (GUI) was developed to provide a user-friendly interface that gives access to all the sensor’s capabilities.

Further information on the sensor and its components can be found in the appendix.

Base Tab



Overview

This is the main tab where essential high-level functions can be carried out, and where force and torque data is displayed.

A

Tab selection. Navigate between the base functions, hardware-level control, and IMU control.

B

Continuous data transmission start button. This tells the sensor to start sending data packets at the specified rate (see F). The sensor parses the packets into difference and sum signals (plotted on the Utilities Tab) which are transformed to force and torque data (plotted on the base tab), as well as temperature and IMU readings (plotted on the IMU tab).
Once start is pressed, the plot options can be changed, but no other functionality that involves communicating with the sensor will work. To configure the sensor in any way, press stop first.

C

Continuous data transmission stop button. This stops the sensor sending packets and allows the user to configure it. Closing the GUI window has the same effect.

D

Data logging button. Check this before starting data transmission if you wish to log the data and save it to a .csv file. The first time it is selected in one session, it will open a dialog in which the user can select where to save the resultant data file.

When data logging is activated, it is essential that no packets are missed, and there is no delay in the data. Hence, any computationally heavy functions are deactivated, including all plotting.

E

Data logging time input. Enter how long to log data for (in seconds). If you wish to measure forever until you press stop, enter -1. Pressing the stop button (C) will stop logging and save the data. Note, closing the GUI will stop transmission but will not save the data.

F

Sampling rate input. Enter the desired sampling rate for continuous data transmission. This can be any value less than 3000Hz. The next time the start data transmission button is pressed, it will proceed at the entered value.

G

Poll button. Poll a single data packet. This returns difference, sum, force, torque, IMU and temperature data. The single data point will be plotted on the active axes (see O), and will be displayed in the value display (J and K). Polling is a useful test to see if everything is working. If polling fails, the sensor is not responding as expected. If the result of polling is that every value is 0.0, the sensor did not initialize properly. It should be unplugged from power and plugged in again. This usually fixes it. If the values that are polled are weirdly high and/or the saturated warning comes on (J), it is likely that the OFC and FSC registers were not read into the software properly upon start-up. This causes the conversion from bits to voltage of the difference signal to be thrown off. To remedy this, the easiest thing to do is to close and reopen the GUI. Otherwise, one can go to the Utilities tab, select ‘all’ under channels, and select On/Reset, thus resetting and redoing the self-calibration on all the ADS’s.

H

Calibration matrix selection. Click here before doing any measurement to load the calibration file. The calibration (.cal) file is an xml file that contains the voltage-force calibration matrix. Without this, the measured force and torque values will be meaningless.

I

Reset button. This performs a full reset of the device. The various components that are reset (successfully or otherwise) are displayed in the status display (L).

J

Temperature and saturation display. The temperature is measured with every force measurement and is reported live here. It is also plotted on the IMU tab. In addition, a few saturation checks are performed with each measurement. The result of these is displayed here. Saturation of the forces and torques is commonly because of incorrectly reported FSC and OFC registers at start-up. To fix this, simply restart the GUI or reset all the ADS's (on the Utilities tab).

K

Force and torque display. This is updated with every measurement. If the values say '(failed)', the sensor is not responding to requests for data, or communication is otherwise faulty. If the values are all exactly 0.0, try turning off and on the sensor (by unplugging it from power).

L

Status display. The sensor returns some responses to requests from the host PC. These responses are printed here. The responses are usually in the format:

<Component>-<Action>-<Result>

Where <Component> can be DAC, IMU, ADS1 through 6, etc. <Action> is what the component was asked to do. Common actions are CNFG (configuration), RST (reset), PWUP (power-up), PWDN (power-down). <Result> is either S for successful or F for failed.

Some other common messages are:

CONT-DTX-S (continuous data transfer stopped successfully)

IMU-$$DA-S (IMU $$ disabled successfully, where $$ can be GR=Game Rotation Vector, AC=Acceleration, LA=Linear Acceleration, GY=Gyroscope, MG=Magnetometer, RV=Rotation Vector)

M

Sensor orientation display. This visualizes the orientation of the sensor based on the IMU game rotation or rotation vector readings. If the IMU is not in one of these modes, the orientation plot is not updated.

N

Orientation display toggle. Turn on or off live display of the sensor orientation. This can affect other plotting speed and performance.

O

Axes selection. Select which components of force to plot. It is possible to plot one or more at a time, and the selection can be changed during live data transmission.

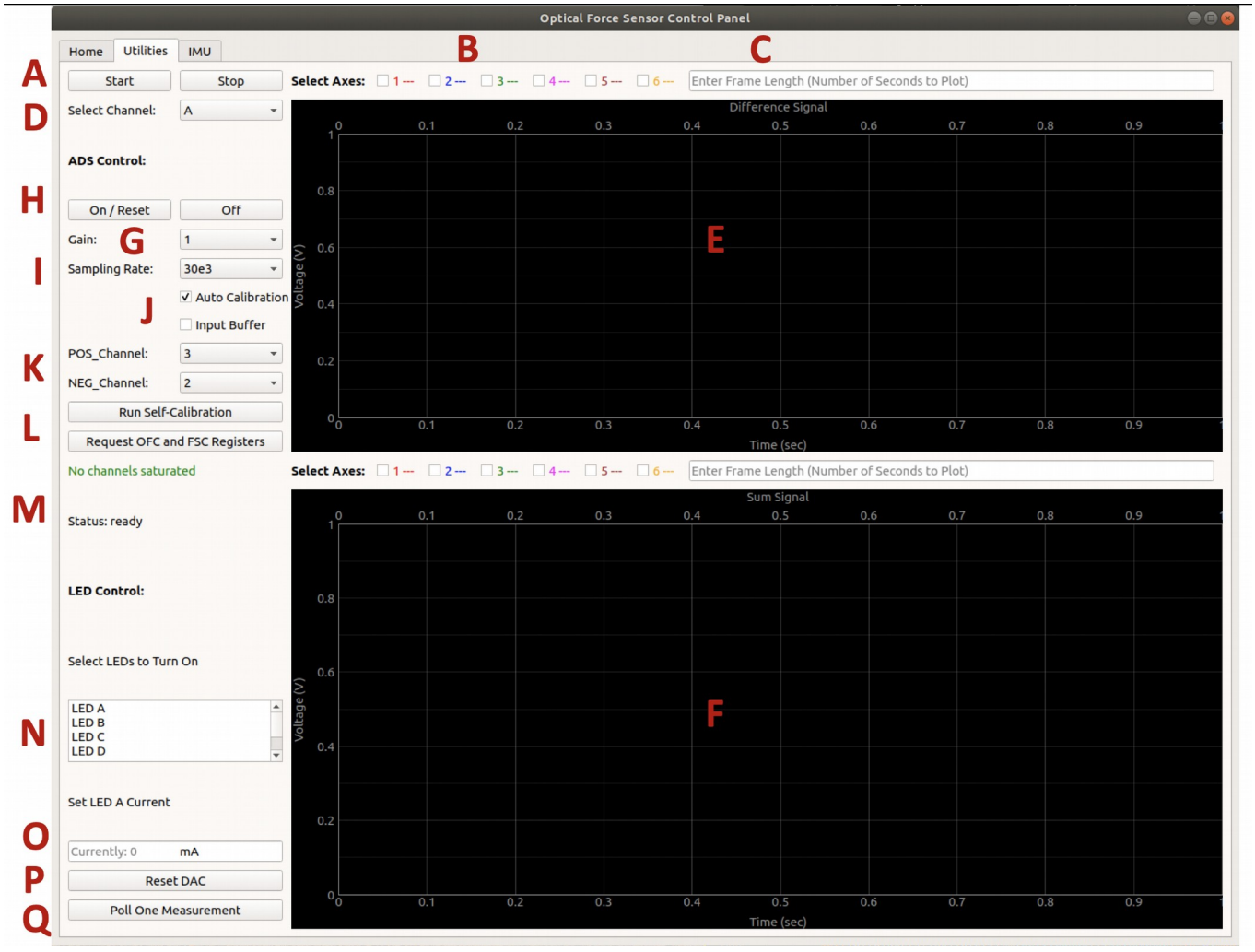
P
Plot range input. Enter how many seconds of data should be plotted. This can be changed during live data transmission. The more data being plotted, the more computationally intense and potentially slow it will be.

Q
Force plot. The x, y, z components of force can be plotted here.

R
Axis options for torque plot. This is the same as O and P for the force plot, but applies to the torque plot.

S
Torque plot. The x, y, z axes of torque can be plotted here.

Utilities Tab



Overview
Directly control some low-level hardware configurations and functions, and view the raw output of the sensor in the form of difference and sum signals of every transducer.

A

Start/stop buttons for continuous data transmission (as explained in B and C of the Base Tab)

B

Axis selection for difference signal plot. Select which (1 or more) of the transducer outputs to plot. This can be changed during live data transmission.

C

Plot range input. Enter how many seconds of data should be plotted. This can be changed during live data transmission. The more data being plotted, the more computationally intense and potentially slow it will be.

D

ADS channel selection. Select which channel to control with the configuration buttons below. It is possible to choose a single channel between A and F, or to select ‘all’, in which case any change is applied to all the channels.
If the sensor’s measurements are saturated, select all and click On/Reset (H) to redo the self-calibration. Or just restart the GUI.

E

Difference signal plot. The difference and sum signals are combined to obtain the forces and torques. Any of the 6 difference signals can be monitored here.

F

Sum signal plot. The difference and sum signals are combined to obtain the forces and torques. Any of the 6 difference signals can be monitored here.

G

Gain selection. Choose the ADS gain.

H

Deactivate, activate, reset ADS. To turn off a channel, select which channel in D, and click off. To turn the channel on again, or reset and re-self-calibrate a channel that is already on, click On/Reset.

I

ADS sampling rate selection.

J

ADS register settings. Toggle auto calibration and the input buffer. More information on these can be found in the ADS datasheet (see apendix)

K

Positive/Negative channel selection. Choose which channel to make the positive and/or negative channel.

L

OFC and FSC settings. Run self-calibration to determine and set the current OFC and FSC register values. Also request the current values and print them. Note, this does not save the reported value in the software to be used for parsing of data packets. To actually update the values, restart the GUI, click On/Reset, or click ‘Run Self-Calibration’.

M

Status and saturation state display. Like J and L in the Base tab.

N

Turn on/off LEDs shortcut. Select an LED to turn it on, deselect it to turn it off. It is generally recommended to use the LED current input below instead as it is sometimes not clear if an LED is actually on or not using this selection widget. In addition, this does not allow one to select the desired current.

O

LED current input. Set the current to the currently selected LED (channel selection) in milliamps. The recommended value for sensor operation is 7mA. This places the sum voltages at their optimal 1.8V.

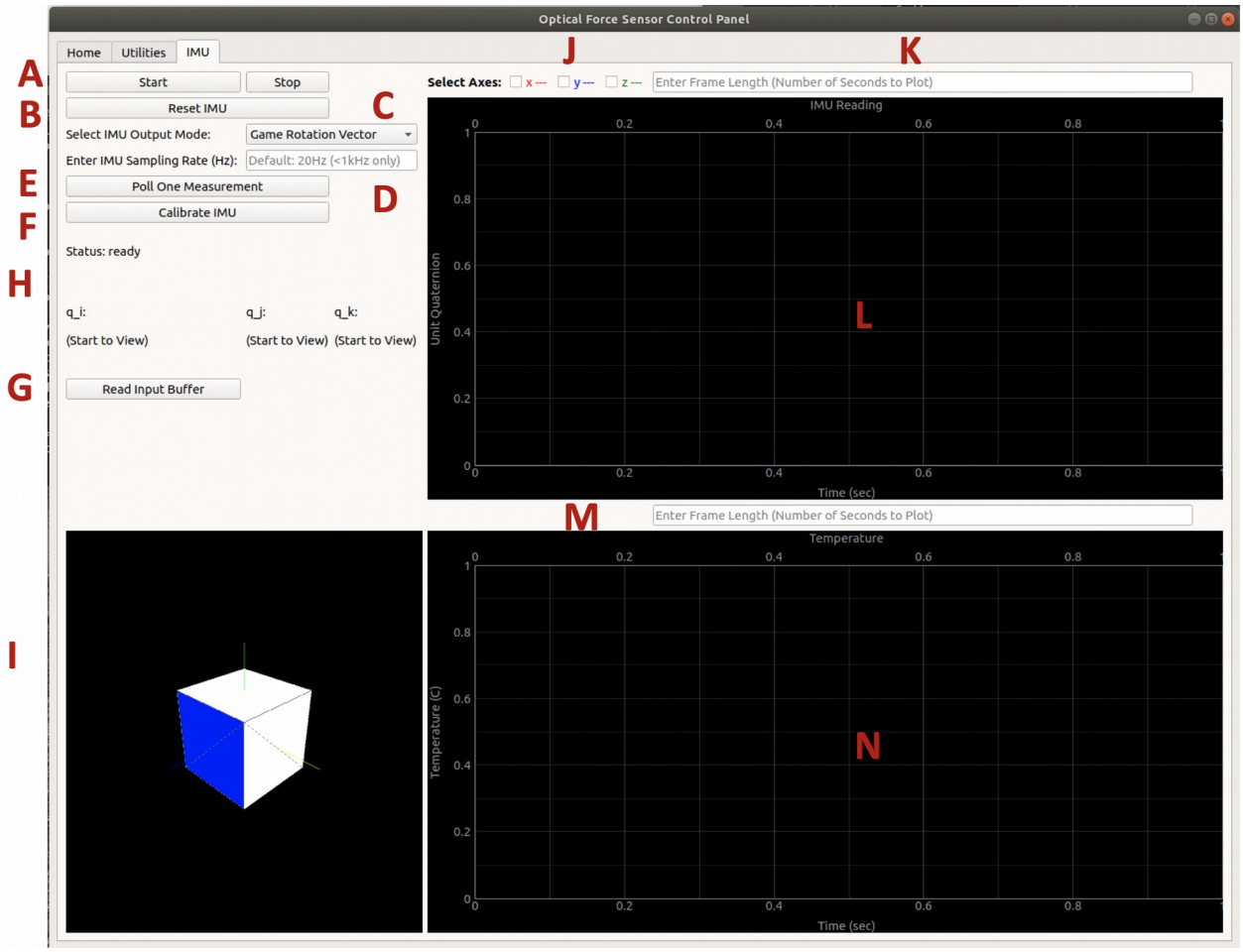
P

Reset DAC. Reset the DAC, setting all LED currents to 0mA.

Q

Poll one measurement. Same as G in the Base tab.

IMU Tab



Overview

Control IMU functions, and view its outputs, as well as a live plot of the temperature.

A

Continuous data transmission start/stop button. Like B and C of the Base tab.

B

Reset IMU. This is useful if the IMU output is acting strange. It is susceptible to failing, so if IMU-RST-F or the like is returned, simply reset again a few times until it succeeds.

C

Select what output mode to put the IMU into. This can be Game Rotation Vector, Rotation Vector, Acceleration, Linear Acceleration, or Gyroscope. The sensor actually alternatingly reports each one at a certain rate. To activate a single mode and deactivate the others, the IMU is simply setting the rate for all the unselected modes to 0Hz.

D

IMU mode sampling rate input. Enter the sampling rate of the selected IMU mode. All but the selected mode will have sampling rate set to 0.

E

Poll button. See G of the Base tab.

F

Calibrate IMU. Start IMU calibration window. See the documentation for that there.

G

Read input buffer. Empty the input buffer and print its contents to the terminal.

H

IMU data display. Prints the live values being measured by the IMU. This can be a variety of values, from the x, y, z components of linear or angular acceleration to the imaginary components of unit quaternions for the rotation vectors.

I

Orientation display. See M of the Base tab.

J

Axis selection for the IMU plot. See O of the Base tab.

K

Plot range input for IMU plot. See P of the Base tab.

L

Plot of the IMU reading. This can be a variety of values, from the x, y, z components of linear or angular acceleration to the imaginary components of unit quaternions for the rotation vectors.

M

Plot range input for temperature plot. See P of the Base tab.

N

Temperature plot. The temperature is plotted here.

Further Features

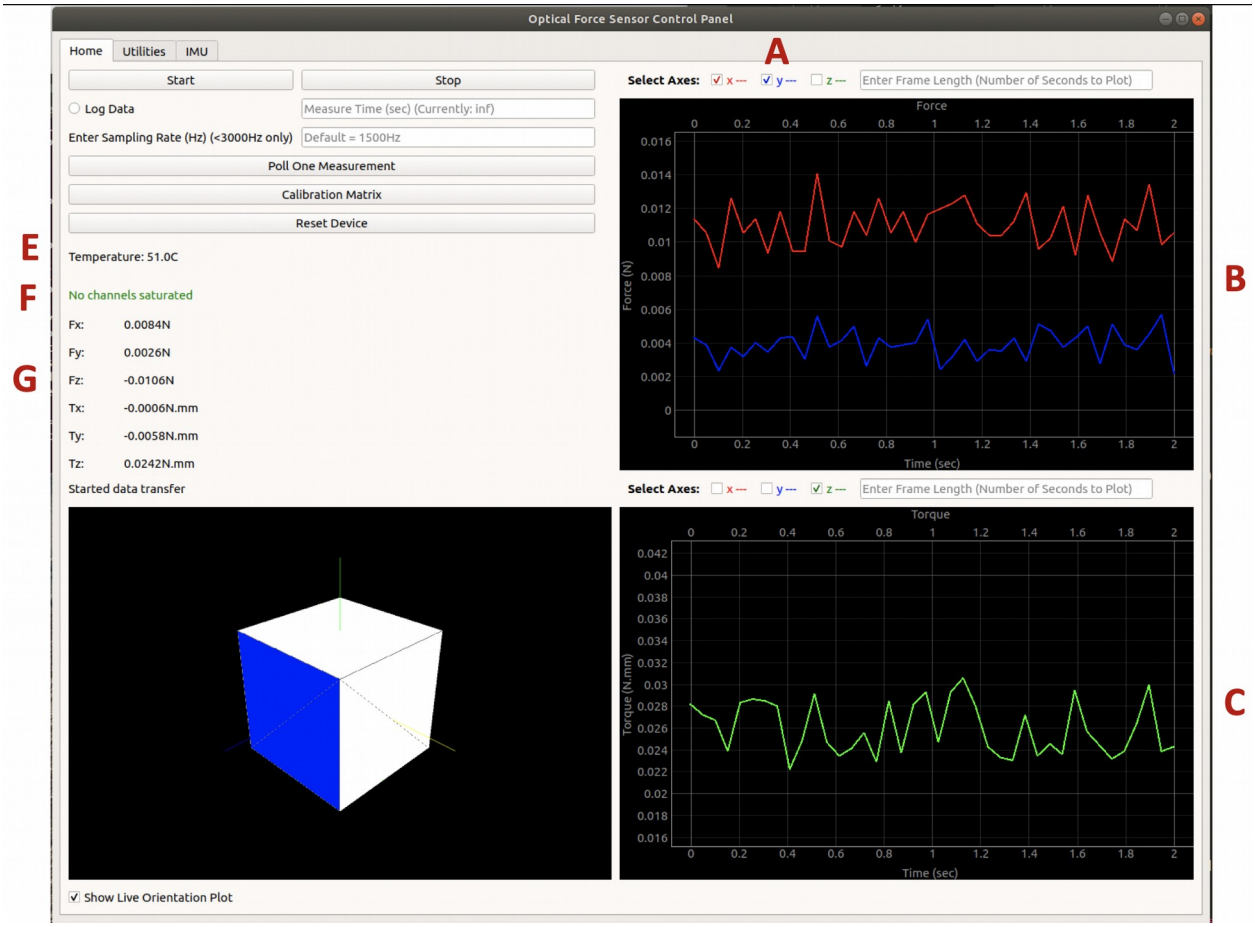


Figure 1: Plot Data Selection

A

One or more axes can be selected on each plot. These will display in the same color as the label and checkbox. For example, Fx is red and Fy is blue. The selection can be changed while it is plotting live.

B

Here we see the x and y axis plotted, as selected in A.

C

Here we see the z-axis plotted, as selected in the torque axis selection menu.

E

During continuous data transmission, the temperature value is updated live here. It is also plotted on the IMU tab.

F

The saturation indicator shows whether any of the difference or sum axes or wrenches are saturated.

G

The values of the forces and torques are also displayed live here. Label G in the Base tab section describes what to do if these values are incorrect.

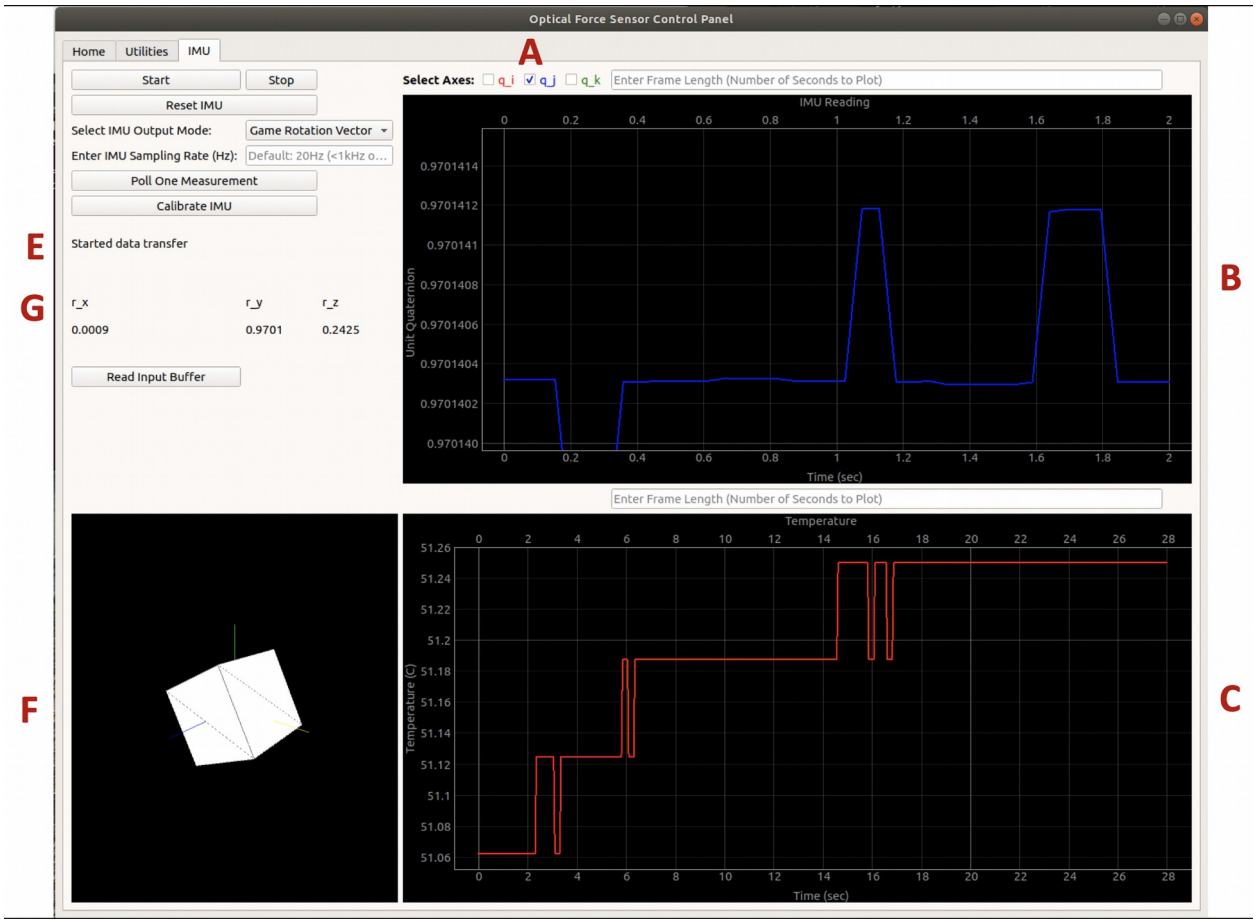


Figure 2: IMU Plotting

A

Axis selection. In the IMU tab, the label of the axis selection boxes will change according to what is being plotted. In this case, it is plotting the i, j, and k components of the unit quaternion reported by the sensor.

B

The IMU plot, with one axis shown

C

The temperature plot. Since this has only one axis and is good to monitor constantly, it has no axis selection buttons, but rather constantly plots the temperature if continuous data transmission mode is active.

E

The status box shows when continuous data transfer has started or stopped.

F

The orientation display shows the sensor’s orientation. It is possible to pan, rotate, and zoom the view. The base axes are shown in green, blue, and

yellow. When the sensor it aligned and mounted vertically, the green, blue, and yellow faces of the cube line up with the corresponding axes.

G

The first three elements of the IMU report are displayed here. This can be the x, y, and z components of acceleration or linear acceleration, or the imaginary components of a unit quaternion in the case of rotation or game rotation vector outputs.

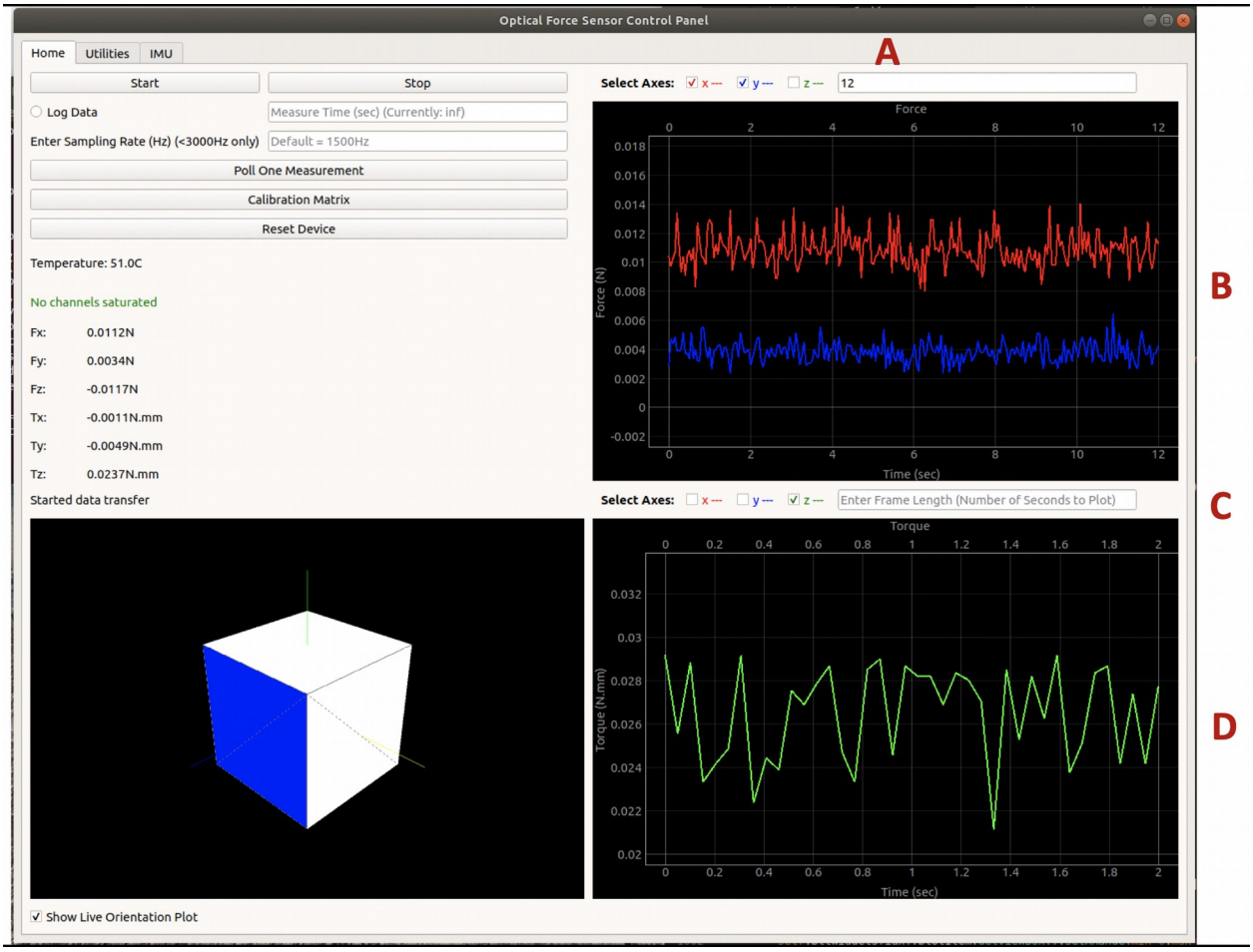


Figure 3: Plot Data Range

A

Plot range input. This box is used to change the plot's range Here it is changed to 12 seconds, and the force plot is displayed accordingly, as seen in B.

B

The force plot with a 12-second range instead of the default 2 seconds. This was set in A.

C

Each plot's range is controlled independently. The torque plot plot range was not changed and thus remains at its default of 2 seconds, as seen in D.

D

The default plot range is 2 seconds. This ensures real-time display of the data as there is little computational intensity.

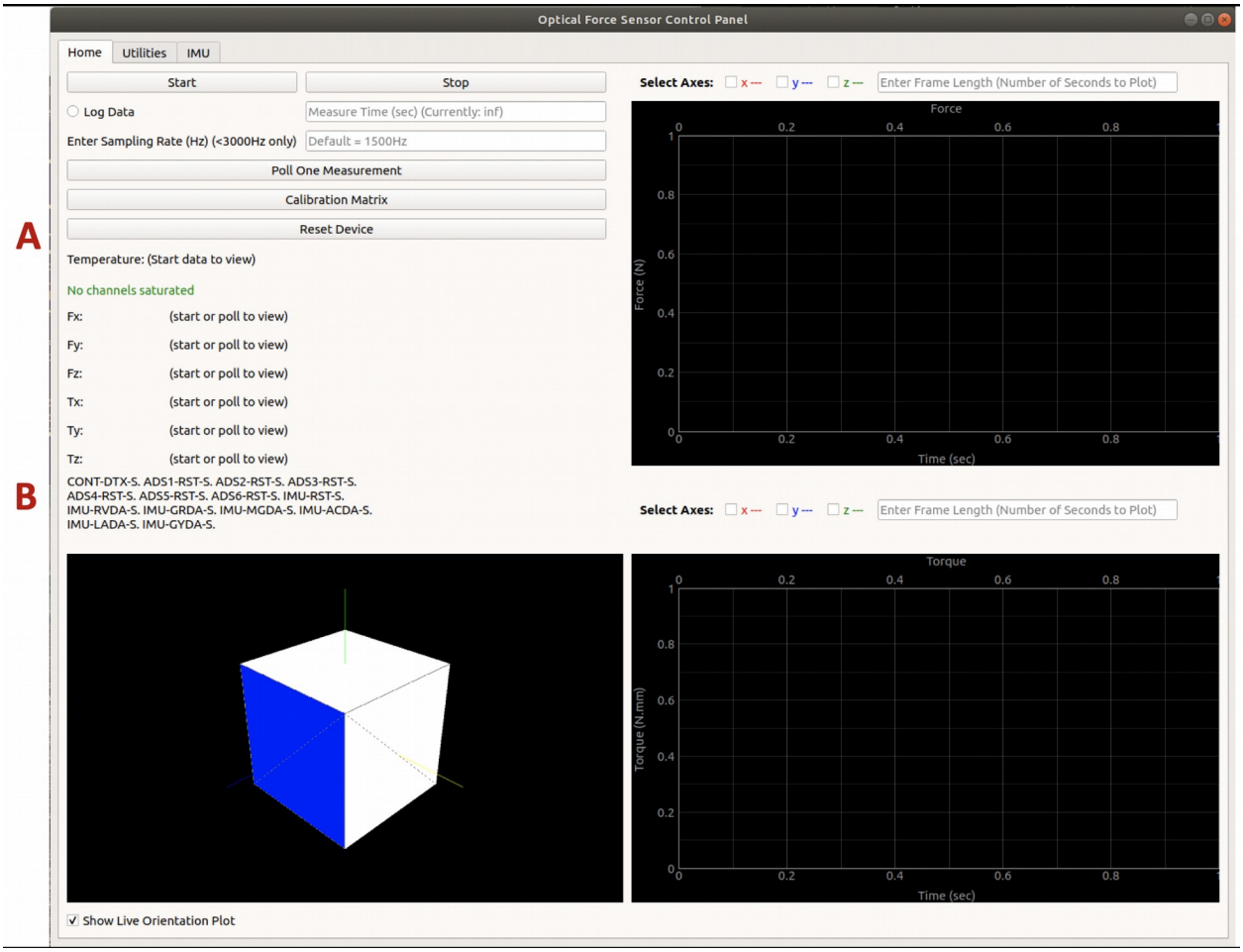


Figure 4: Sensor Responses

A

When a button is pressed, the response sent by the sensor is displayed in the status display. In this case, reset was pressed.

B

When reset was pressed, the sensor successfully stopped data transmission, reset all 6 ADSs, reset the IMU, and disabled all IMU outputs. This is displayed in the status display.

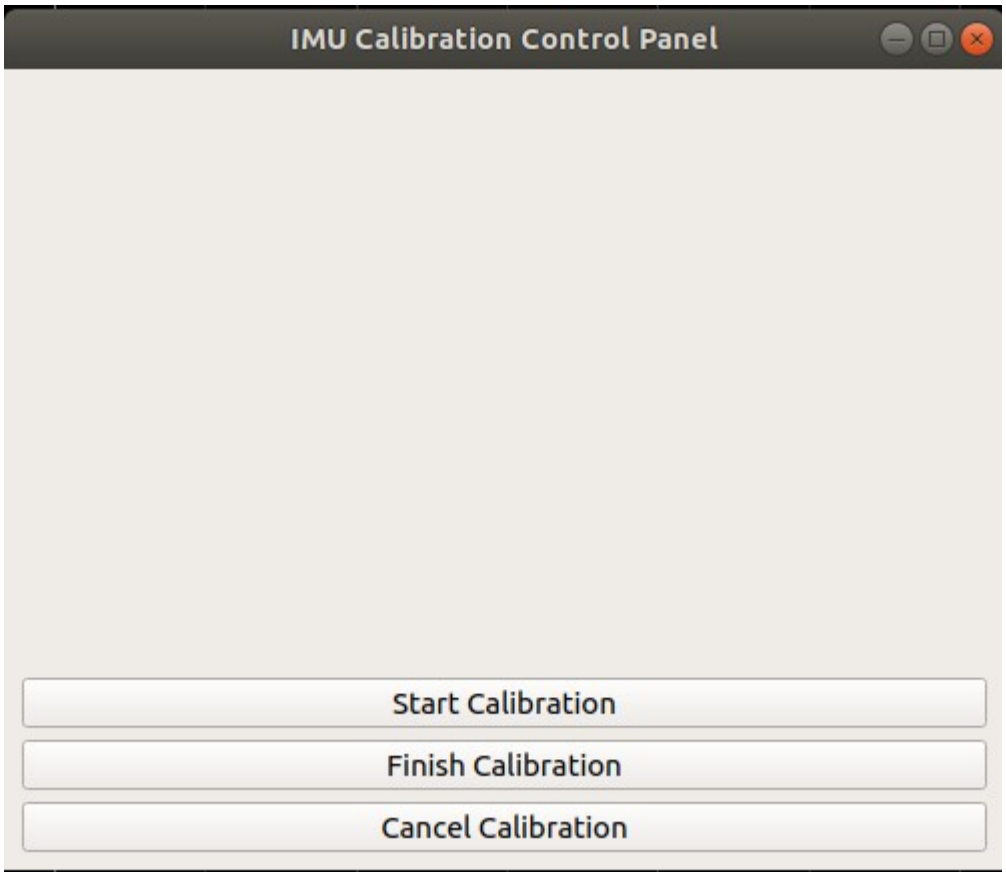


Figure 5. IMU Calibration

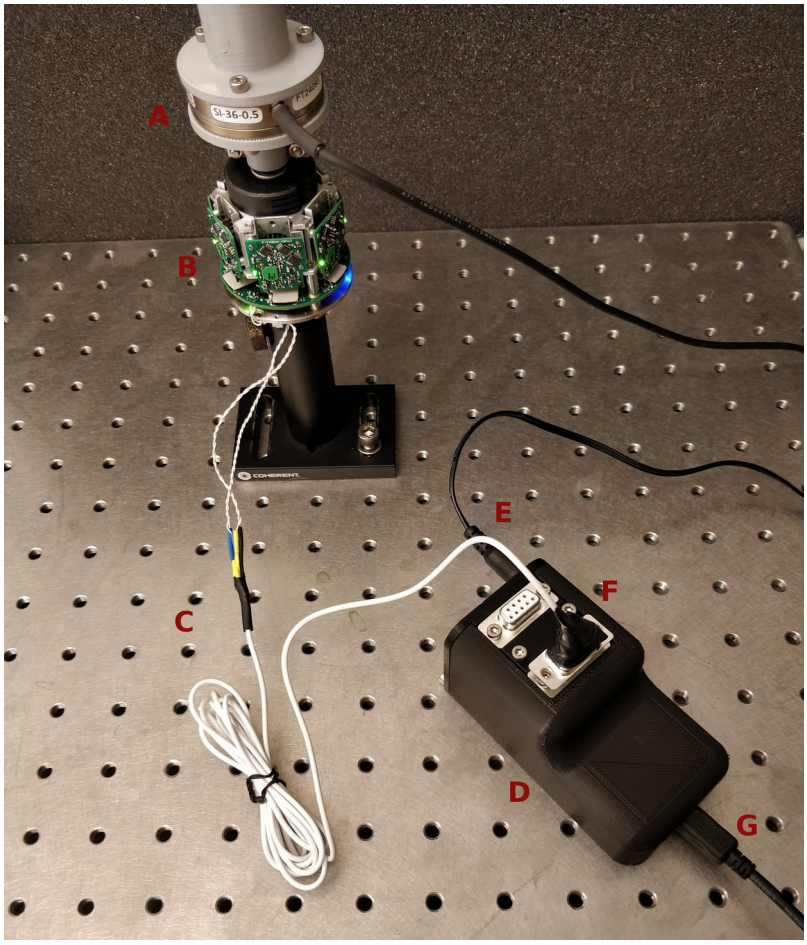
This scetion of the GUI is self-explanatory. To access it, click ‘Calibrate IMU’ (F on the IMU tab). When ‘Start Calibration’ has been pressed but no data is coming, ‘no data’ will flash on the screen. If data is coming, the accuracy of the game rotation vector and magnetometer values will be displayed. Keep moving the sensor through different orientations and monitor the accuracy values until they are high enough. Then click ‘Finish Calibration’ to finish the calibration and save the results to the IMU. To cancel the calibration and exit without saving, click ‘Cancel Calibration’.

Common Errors

Below is a list of common problems or errors that could occur and how to deal with them.

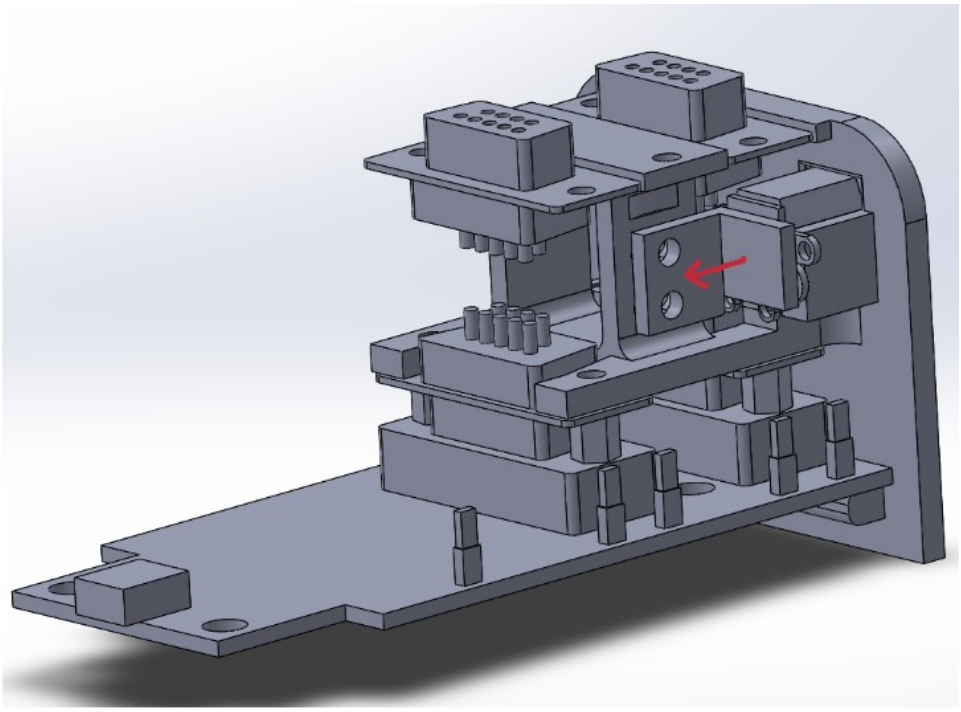
- Polling fails: The sensor is not responding as expected. Try waiting a second then polling again. Otherwise, restart the GUI. If this fails, unplug the sensor and plug it back in. If this fails, there may actually be something wrong with the sensor or firmware.
- Forces and torques are always 0.0: the sensor did not initialize properly. It should be unplugged from power and plugged in again. This usually fixes it.
- Measured wrenches are weirdly high and/or the saturated warning is on: it is likely that the OFC and FSC registers were not read into the software properly upon start-up. This causes the conversion of the difference signal from bits to voltage to be thrown off. To remedy this, the easiest thing to do is to close and reopen the GUI. Otherwise, one can go to the Utilities tab, select ‘all’ under channels, and select On/Reset, thus resetting and redoing the self-calibration on all the ADS’s.
- The GUI crashes with error messages to do with FTDI connection failed, bulk read failed, etc. Here the USB to RS-485 adapter is failing. Try unplugging the USB connection at the host PC and plugging it in again, then restarting the GUI. If this does not work, try it a few times in different USB ports. If this still does not work, unplug and plug back in the Mini-USB connection at the FTDI box. Doing all of these things a few times has always fixed the problem in my experience.
- Sending commands works fine, but the sensor never responds: First check that the sensor is on. There should be green LEDs shining. If yes, check that the sensor cable is plugged in to the correct port of the FTDI chip. Port 1 is the one closer to the Mini-USB connection.
- In timing a number of packets for a latency test or similar application, the results are strange: It is possible the buffer is full of data. Try clearing it first.
- When sending a byte, the byte also appears in the host PC’s receive buffer: Make sure pins 7 and 8 of the DB9 connector are shorted together to disable local echo.
- Sending bytes works, but only 0’s arrive at the other end: Check that the baudrates match.
- You are sending consecutive numbers counting up, but what is arriving is counting down in steps of 2 and restarting after 0x80 (or some similar discrepency): Check that the cable is set up for RS-485, not RS-232. If yes but this is still happening, it is quite possible the data+ and data- wires are switched.

Physical Setup



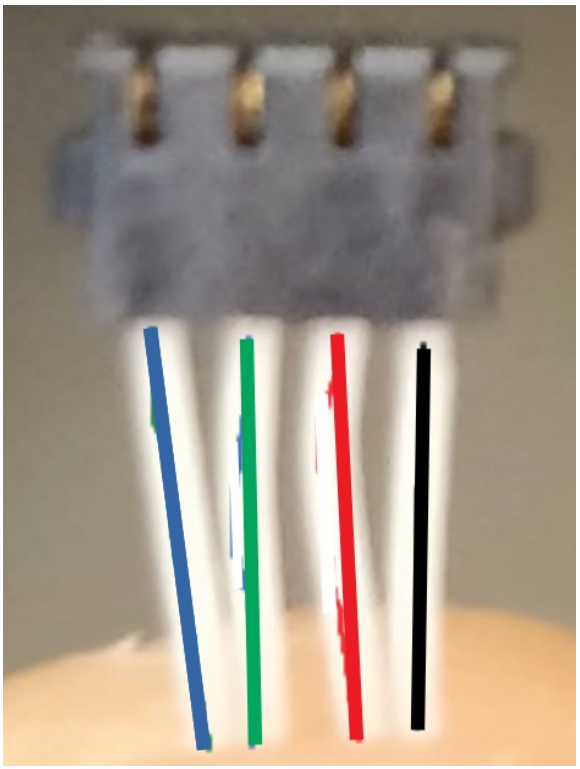
Above is an image of a typical setup with the sensor. In this case, the setup is for calibration. The sensor is mounted on a shaft, shown by label B. Since it is a calibration setup, an ATI force sensor is mounted above it, at A. C shows the power and signal cable to the sensor, which plugs into the FTDI box (D) in one of the ports (F). In this case, it is plugged into port 1 of 2. E is the power cable for the sensor, and G is the USB connection to the host PC.

The FTDI box is designed to be easy to modify or fix. One problem could be the power socket coming loose and pushing into the box when you try to plug it in. To fix this or any other problem, unscrew the two screws on that face of the box, as well as the screws at the ports on top. The whole active assembly can then slide out, and looks as is shown below. To fix the loose power socket problem, push the socket back into position and tighten the screws indicated in red below. To fix an electrical connection, it is possible to unscrew and remove each of the ports and the middle spacer independently.



This facilitates repairing them without having to solder awkwardly in a tight space. The FTDI chip itself can also be removed and replaced.

The cable to the FTDI chip includes power (12V, red), ground (black), data+ (blue), and data- (green). The sensor-side connector is arranged as follows shown below:



If the communication between the sensor and computer is garbled or not making sense, it is likely that the data+ and data- cables have been switched by mistake.
The FTDI-side connector is a male DB9 connector. The pin names in the FTDI datasheet are somewhat confusing, so just follow the pin numbers written on the actual DB9 connectors. Pin 1 -> ground; Pin 6 -> +12V; Pin 4 -> Data+; Pin 5 -> Data. More information on the FTDI chip is found in the appendix.

Sensor Control Software

Introduction

A software package was written to control the sensor and handle the data it sends. The first package was written in ROS for easy integration into dVRK. However, the overhead associated with ROS leads to very slow data transmission well below that which we want to achieve (max ~1kHz instead of 3+). A stand-alone package was therefore also created, which avoids the ROS overhead, can run much faster, and is still simple to use with dVRK.

The software is described in great detail in the Appendix. Ultimately, both packages are used to send commands to the sensor and read what it returns. In addition, both have a continuous data transmission mode. To handle the real-time flow of data, the standalone package creates a second thread which reads and parses the packets before emitting a PyQt Signal containing the parsed data. This allows a program to subscribe to the signal and handle the data in real time without blocking . Similarly, the ROS package starts a new node that simply publishes continually to a ROS topic.

The benefit of the PyQt thread as opposed to a normal Python thread is that the signal handling architecture is very useful and easy to implement in applications that use the sensor. Furthermore, it facilitates the creation of a Qt GUI with the measured data.

Installation

The following software is required for the package to run:
Python 2.7

```
sudo apt-get update
sudo apt-get install python
```

Also download the following, in this order:

```
sudo apt-get install libudev-dev
sudo apt-get install build-essential
suo apt-get install git-core
sudo apt-get install cmake
sudo apt-get install doxygen
```

Download libusb:

- https://github.com/libusb/libusb/releases/download/v1.0.22/libusb-1.0.22.tar.bz2
- Unzip using
  tar xvf libusb-1.0.22.tar.bz2
- Cd into the directory and call the following
  ./configure
  make
  sudo make install

Continue downloading these packages (some might already be installed):

  sudo apt-get install libconfuse-dev
  sudo apt-get install swig python-dev
  sudo apt-get install libboost-all-dev

Install libftdi manually from

- https://www.intra2net.com/en/developer/libftdi/download/libftdi1-1.4.tar.bz2
- Follow the directions in README.build. Note, calling sudo apt-get install libftdi1 does NOT work. There is an issue with baudrates.

Call the following to give permissions to the device:

  sudo echo 'SUBSYSTEMS=="usb", ATTRS{idVendor}=="0403", GROUP="dialout", MODE="0666"' > /etc/udev/rules.d/99-libftdi.rules

- If you get a message saying permission denied, call
  sudo vim /etc/udev/rules.d/99-libftdi.rules
- Press i
- Type this:
  SUBSYSTEMS=="usb", ATTRS{idVendor}=="0403", GROUP="dialout", MODE="0666"
- Press [esc] then type :wq and press [enter]
- Unplug and plug back in the device
- If there are still issues, try calling
  su - $user
  - e.g. I would call su – david

To install pylibftdi, call:

  pip install pylibftdi

- To test the installation, enter Python (type python in the terminal) and call the following
  import pylibftdi
  d = pylibftdi.Driver()
  d.list_devices()
- This should list some information about the connected FTDI device
- Also test that you can connect to the device. Try connecting the two ports of the FTDI board together (GND to GND, data+ to data+, data- to data-) and doing:
  import pylibftdi as ftdi
  a= ftdi.Device(interface_select=1)
  b= ftdi.Device(interface_select=2)
  a.write(b'\xff')

```
b.read(1)
```

- The read call should get the message that ‘a’ sent. If it fails to connect to the device, try calling python at the command line as root (sudo python) and do it again. If this works, then you need root privileges to access the device, which is annoying and I can’t remember how to fix it.

Some basic Python libraries are also needed:

```
pip install numpy
pip install numba
pip install codecs
pip install struct
```

Finally, if the sensor is to be used with a dVRK system, or using the ROS software package, you will need the following software. To install the dVRK, ROS, CISST-SAW stack, first install ROS:
http://wiki.ros.org/melodic/Installation/Ubuntu

Then follow the instructions here for CISST and SAW:
https://github.com/jhu-cisst/cisst/wiki/Compiling-cisst-and-SAW-with-CMake#13-building-using-catkin-build-tools-for-ros

Finally, install dvrk-ros:
https://github.com/jhu-dvrk/sawIntuitiveResearchKit/wiki/CatkinBuild

Appendix

Sensor Software and Firmware

FPGA-Based Optical Force Sensor: Design Architecture & Evaluation

Amir H. Hadi Hosseinabadi, David G. Black, and Septimiu E. Salcudean, *Fellow, IEEE*

***Abstract* — A transparency-optimized teleoperation system requires force sensing at the master and the slave manipulators [1]. A novel optical force sensor design was presented in [2], where it was also shown how the design could be adapted into a 6-axis force sensor that could be mounted on the shaft of common EndoWrist instruments. This paper presents our novel electronics and firmware architecture that allow this 6-axis optical force sensor to provide high resolution force measurement and to be easily integrated into research and development projects, providing a reliable 1 ms latency when interfaced to the Robot Operating System (ROS) environment and/or the da Vinci Research Kit (dVRK). The paper briefly explains the sensor electronics and outlines the Field Programmable Gate Array (FPGA) architecture that allows for a low latency operation through parallel processing.**

***Keywords* – Product Design, Development and Prototyping, Force and Tactile Sensing, Control Architectures and Programming, Surgical Robotics: Laparoscopy, dVRK**

I. INTRODUCTION

More than 1 million MIS procedures were completed worldwide in 2018 using da Vinci Surgical Systems [3]. Despite their growing popularity, no da Vinci or other Robot Assisted Minimally Invasive Surgery (RAMIS) system, which is clinically in use, provides force feedback to the surgeon. Without force feedback, the surgeons are deprived of the rich information in tissue palpation and direct interaction with the surgical tools. While experienced surgeons develop the skills over time by using visual cues, lack of force feedback causes longer operation times, possible tissue trauma (due to excessive forces), and surgical errors (e.g. suture breakage) by novice surgeons [4], [5].

Therefore, many studies have been published on instrumenting different tele-surgical platforms to evaluate the efficacy of force feedback, and to improve transparency (tele-presence) [6]. Presentation of force information can be in different modalities including direct haptic feedback [7] (kinesthetic or tactile), visual feedback [8] (augmented reality or physical indicators), and sensory substitution (vibration [9], audio [10], or tactile [11] feedback). Common to all of the above modalities is the need for a means of force measurement or force estimation.

In [2], we evaluated the feasibility of using an optical force sensor that is adaptable to common surgical instruments. The sensor uses a light-blocking slit that moves in response to the applied force and modulates the light incident on a bicell photodiode from an emitter (Infrared LED). The promising results of this work led us to the design of a six-axis force sensor. While an overview of the sensor's mechanical design is presented, this paper focuses on the electronics and firmware design and the developed software interfaces to access sensor data.

A.H. Hadi Hosseinabadi (ahhadi@ece.ubc.ca), D. G. Black (dgblack@ece.ubc.ca), and S. E. Salcudean (tims@ece.ubc.ca) are with the Robotics and Control Laboratory, Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, BC V6T 1Z4, Canada.

Figure 1. Six-axis optical force sensor (a) assembled view (b) active and passive components

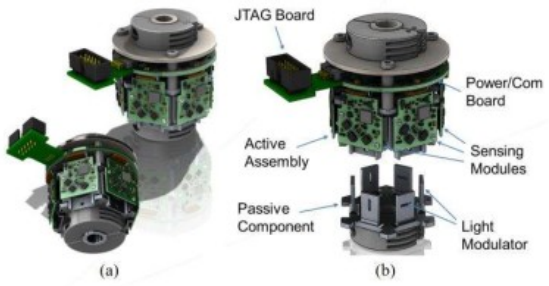
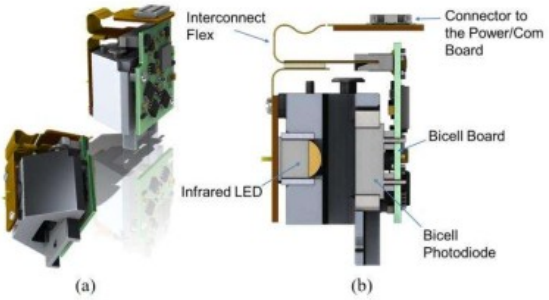


Figure 2. Sensing module (a) Assembled (b) Section view



The optical force sensor is comprised of six sensing modules in a hexagonal configuration as shown in Fig. 1. Each sensing module has an infrared LED that is placed in-line with a bicell photodiode as shown in Fig. 2. Three of these modules are configured to be most sensitive to axial force and lateral moments. The other three modules are interleaved with the first three modules and configured to be most sensitive to lateral forces and axial torsion. The six modules and all the electronics for power conditioning, signal conditioning, and communications form an active assembly. Six aluminum slits which work as light modulators form a passive component.

The active and passive components are installed on the load carrying structure, e.g. the shaft of a surgical instrument.

During installation, the slit associated with each sensing module is aligned with the gap that separates the two active elements of the corresponding bicell. Lateral deflections, twist, and axial strain of the shaft lead to a relative motion between the passive and active components, which causes the slits to shift with respect to the LED-bicell pairs. By combining outputs from all six modules, it is possible to determine axial, radial, and torsional loads.

In addition to force sensing, an Inertial Measurement Unit (IMU) that measures linear accelerations, angular velocities, and rotation vectors is integrated into the sensor design. The IMU measurements can be used in sensory substitution applications [12], as well as gravity and inertia compensations [13] of the measured force data.

Field-Programmable Gate Arrays (FPGAs) can be exploited in the design of reconfigurable sensor systems [14]; the FPGA resources can be efficiently utilized to deliver the desired performance in a particular application. We therefore incorporated an FPGA into the electronics design. In addition, we separated the analog and digital electronics into two custom boards to provide a high level of flexibility for future developments. The digital electronics board is a generic board that can be interfaced with any peripheral with different sensing technologies (i.e. strain gauges, capacitive sensing, etc.), as long as the data is locally digitized. The sensor firmware is developed to be generic and does not limit the flexibility mentioned above. To the best of our knowledge, no force sensor design with a similar electronics and firmware architecture has been presented in the literature.

One of the widely used teleoperation research platforms is the daVinci Research Kit (dVRK) [15]. This open-source framework is developed on the retired first-generation da Vinci Surgical Systems. The community of researchers using the dVRK has grown over the past decade, to some 35 research institutions world-wide [16]. This framework is developed to provide low-level access to all the control functions as needed by the researchers. We therefore developed a software package in ROS that allows for seamless integration of the force sensor into dVRK. A Python package for stand-alone applications has also been developed.

Section II elaborates on the electronics design. The firmware architecture is detailed in Section III. The software packages along with the latency test results are presented in Section IV. Section V concludes this paper and briefly comments on future work.

II. Electronics Design

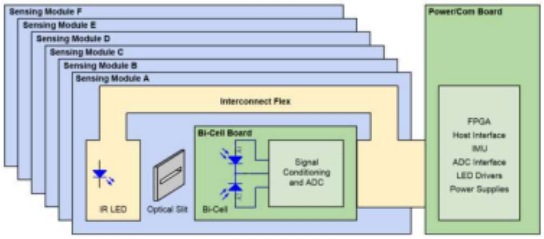
The sensor electronics is based on three custom boards: (1) a Bi-Cell board, (2) a Power and Communication board, and (3) an Interconnect Flexible board.

The Bicell board integrates signal conditioning (explained in [2]) for electro-optical conversion and an analog-to-digital (ADC) converter. The bi-cell photocurrents are conditioned through appropriate offsets and amplification to DIFFERENCE and SUM signals, and are converted to digital signals in close proximity to the sensors for low electromagnetic interference. The DIFFERENCE signal normalized by the SUM value gives the position of the slit centroid w.r.t the bicell's gap which can be calibrated for force estimation. Due to space limitations on the Bicell board, the nulling circuitry is removed and a high resolution ADC (ADS1257) is used instead. Each sensing module includes one Bi-Cell board.

The digital interface to each of the six sensing modules is incorporated on a single Power/Com board along with a Field-Programmable Gate Array (FPGA), LED Drivers, an Inertial Measurement Unit (IMU), power supplies, and interfaces for host communications. The electronics is kept compact by using multi-layer boards, and the cable to the device is kept thin by using half-duplex (2-wires) RS485 serial communication. A USB-RS485 bridge is used in the current system, however it could be replaced by an RS485 PCI adapter if shorter latency is needed (Section IV).

Each of the Bi-Cell Boards connect to the Power/Com Board through a flexible printed circuit board (Interconnect Flex). A block diagram of the system is shown in Fig. 3.

Figure 3. System block diagram



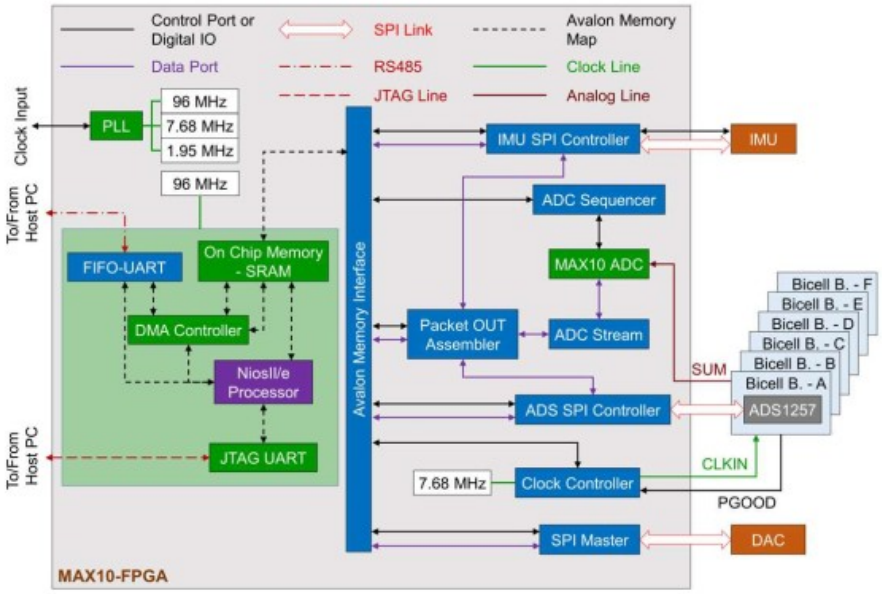
III. Firmware Design

The FPGA has two main functions: (1) exchanging data with the PC through the RS485 link, and (2) interfacing with the FPGA peripherals (Bicell boards, IMU, DAC, Temperature Sensor and EEPROM). Fig. 4 shows an overview of the FPGA hardware configuration. The programming is done in VHDL through Intel Quartus Prime 16.1.

A Nios II/e processor is instantiated into the FPGA. It initializes the device peripherals (IMU, ADC, and DAC) after sensor power up. During normal operation, the Nios processor is idle; it only listens for the input commands from the software running on the host PC and triggers a pre-defined action.

The "ADS SPI Controller" continuously reads the DIFFERENCE signal from all the bicell boards at 30 kSPS. As shown in Fig. 5 (a), the controller includes a SPI master, an arbiter, a SPI driver, and a moving average filter. The SPI master handles 1-byte full-duplex transactions with the converter. The arbiter manages access to the SPI master by the Nios processor or the SPI driver. The Nios processor initializes the ADS1257 upon power-up and then hands over the control of the SPI master to the SPI driver for the continuous read of the sampled data as explained above. Because one ADS conversion cycle requires 33 µs, the ADS-SPI Controller continuously requests data conversions and stores the results in registers.

Figure 4. FPGA hardware configuration



The read data is fed to a moving average filter that further reduces the risk of aliasing and the noise level in the measurements. The filter averages the incoming data over a moving window of 27 samples. The frequency response of an M-point moving average filter can be represented by (1) where f_n is the normalized frequency and f_{DR} is the sampling frequency of the ADC. A 27-point moving average filter with f_{DR} = 30 kHz has its -3 dB bandwidth at 492 Hz which is close to the desired bandwidth of 500 Hz [2].

$$H[f_n] = \frac{sin(\pi f_n M)}{M sin(\pi f_n)}, f_n = \frac{f}{f_{DR}} \quad (1)$$

The "IMU SPI Controller" continuously reads the IMU data after it is initialized by the Nios processor. The controller, as shown in Fig. 5 (b), has a design identical to the ADS SPI controller without the moving average filter. The IMU data rate can vary between 100 to 1000 Hz depending on its configuration and the sensor of interest (gyro, linear acceleration, rotation vector etc.).

The FPGA's integrated ADC is used for sampling the SUM signal from all the Bicell boards. The "ADC Sequencer" controls the multiplexer of the FPGA's integrated ADC. The "ADC Stream" parses the sampled data and saves it in the register associated with the sampled channel. Because the ADC can sample at data rates up to 1 MSPS, a moving average filter can optionally be cascaded to the output data stream to improve the noise level in the measurements. The communication with the DAC that controls the LED currents is through another SPI master and is directly managed by the Nios processor.

Data transfers to the host PC are managed by a Direct Memory Access (DMA) Controller and through a UART core with FIFO buffer. The UART to RS485 bridge can operate at baudrates up to 20 Mbps. When the software requests data in polling mode, the Nios processor enables the Packet-Out Assembler. The Packet-Out Assembler reads one snapshot of all the peripherals' registers with their most recent values in one clock cycle. It combines the sampled data in a pre-configured structure of 47 bytes, calculates a CRC32 checksum, and prefixes the data with a header. The header is comprised of a start byte, a 1-byte packet number, and a CRC8 checksum. The data-out packet, for transmission to the host PC, is 54 bytes long including the header and the checksum.

Figure 5. HDL designs for (a) ADS SPI Controller (b) IMU SPI Controller

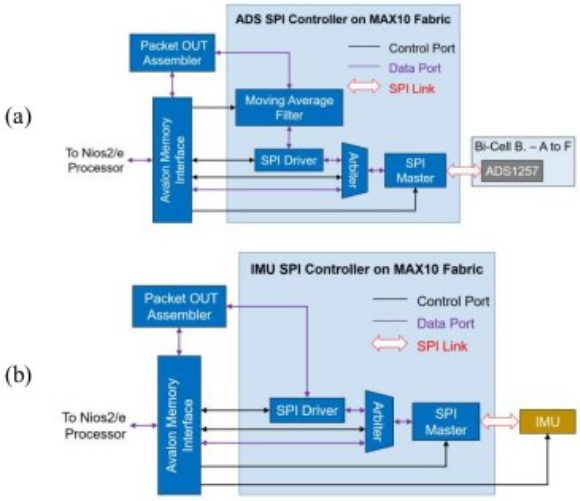


Figure 6. Packet-Out Assembler execution time - Modelsim simulation

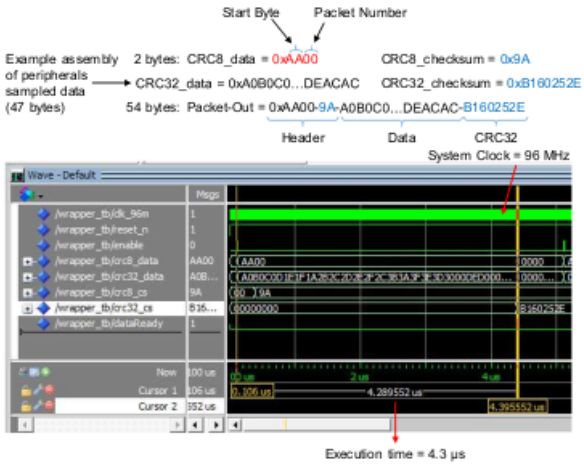
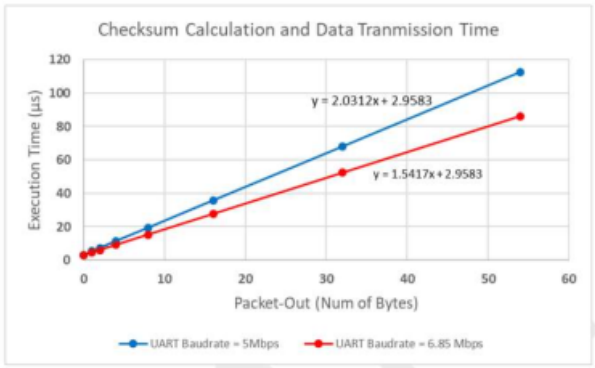


Figure 7. Processor execution time to a polling request



As shown in Fig. 6, with the FPGA core running at 96 MHz, one call to the Packet-Out Assembler takes only 4.3 µs to complete. Once the data-out packet is ready, an interrupt is triggered that initiates the DMA controller. The UART core transmits data as long as the FIFO buffer is not empty. With the USB-RS485 Bridge operating at 6.85 Mbps, each transfer of the 54-byte packet takes only 55 µs. Therefore, upon initiating the CRC calculation, it takes less than 60 µs until the data-out packet is completely transferred. With the firmware code overhead, the execution time required after receiving the command from the host PC is approximately 86 µs (Fig. 7 shows the execution time for two different baudrates). This allows for data rates up to 11.5 kHz. In the current design, the latency is limited by the USB frame rate (up to 1 ms as shown later) because of its polling mechanism [17].

Alternatively, a RS485 PCI adapter, as mentioned in Section II, can be used for a shorter latency performance if needed. With the adapter operating at 20 Mbps, the transfer of the 54-byte data-out packet takes only 21 µs. Therefore, upon initiating the CRC calculation, it takes less than 26 µs until data-out packet is completely transferred (4.3 µs for CRC calculation). With the firmware code overhead, the execution time since receiving the command from the host PC will be about 52 µs; when pushing the RS485 interface to its limit, it is possible to achieve the data rate of 19.2 kHz. However, it is not recommended to operate the serial communication link at a higher baudrate than what is needed, because it will require the Packet-out Assembler and the UART core to operate more frequently, which consequently leads to more heat generation and temperature rise at the FPGA.

The sensor firmware can also operate in streaming mode, in which it continuously transmits data at a specified rate. The data rate can be adjusted by the input command. The continuous data streaming uses a timer interrupt to enable the Packet-Out Assembler explained above.

IV. Software

Two software packages are developed for host PC transactions with the sensor: 1) A standalone library in Python, and 2) A ROS package for dVRK integration.

A. Python Library

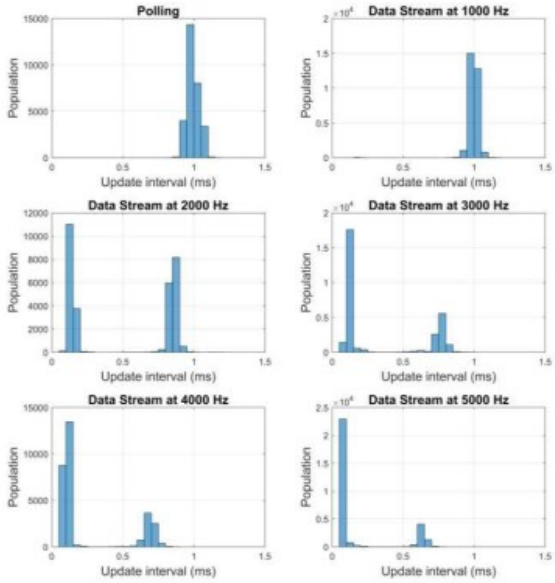
The software reads and parses the incoming data into (1) the DIFFERENCE and SUM values for each Sensing Module, and (2) the IMU data (Linear accelerometer, gyroscope, rotation vector, etc.). The parsed data will be fed into a calibration routine to resolve force and torque values.

The low-level hardware-abstraction layer that transfers bytes between the PC and the FTDI USB-RS485 bridge is libftdi, an open source FTDI driver library for C/C++ in Ubuntu. On top of this layer, the software's main thread relays user commands (configuration and data requests) to the sensor, and receives data in polling mode. In streaming mode, a separate thread runs continuously; once started, it constantly reads from the input buffer, parses the data, resolves the force/torque and IMU data, and writes the resolved wrench and IMU data into an internal LIFO (Last In First Out) buffer. The LIFO buffer is globally accessible for reading to all threads and has limited capacity so it keeps only the most recent few packets. This architecture has multiple advantages; (1) it is fast and non-blocking, (2) the receive buffer is emptied constantly, and the LIFO internal buffer ensures the most recent available packet is always used, and (3) memory usage is not a concern because only the most recent few packets are retained.

In order to evaluate the latency and data throughput, 30,000 packets were read in polling and streaming modes at different data rates of up to 5,000 Hz. The UART baudrate was set to 6.85 Mbps for all cases. As Fig. 8 shows, the latency in polling mode and streaming mode at 1,000 Hz is approximately 1 ms due to the USB polling mechanism and error correction protocol. By increasing the data rate, more data-out packets are being combined in one USB frame; at 2,000 Hz, the ratio of the short latency (~125 µs) population to long latency (~875 µs) population is close to 1:1. At 3,000 Hz, the ratio is 2:1, at 4,000 Hz, the ratio is 3:1, and at 5,000 Hz, the ratio is 4:1. In typical control applications, the bandwidth of the position control loop is less than 50 Hz. Therefore, the 1 ms latency is not expected to cause significant performance degradation. However, in applications with more stringent latency requirements, an RS485 PCI adapter can be used to achieve lower latency. No

packet drop was observed in testing which shows a low noise communication with delivery rate of close to 100%.

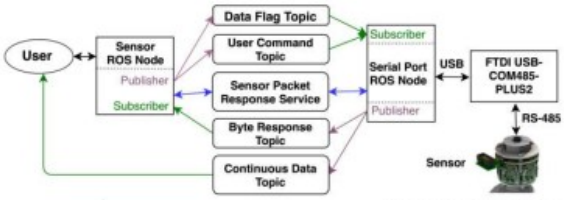
Figure 8. Latency and data throughput test for ROS package



B. ROS Package

A ROS package was developed to ensure instant compatibility with the dVRK system. The main ideas behind the architecture are very similar to the previously described system, whose multi-threaded, real-time nature fits well into a ROS structure.

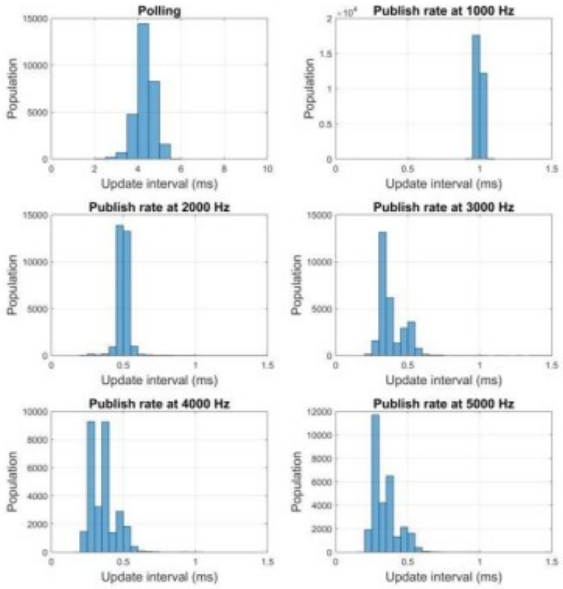
Figure 9. ROS architecture for dVRK integration



Topic	Data Flag Topic	User Command Topic	Sensor Packet Response / Continuous Data Topics
Message Type	FlagMsg	ByteMsg	SensorOutput
Contents	bool data_flag int32 data_rate int32 sensor_num	int32[ ] byte_data int32 num_bytes int32 sensor_num	float32[6] wrench float32[5] imu string report_id int32 sensor_num

Similar to Section IV.A, the hardware abstraction layer is libftdi. It is packaged as a ROS node ("Serial Port ROS Node" as shown in Figure 16) for each connected sensor. The node subscribes to topics on which commands are published, and carries out the command if the sensor number in the message matches the node's sensor number. It takes care of low-level interactions with the FTDI chip and publishes the sensor response to a response topic. Polling a single package is implemented slightly differently; since it is a command that expects a response, it is carried out using a ROS service. Data transfer in streaming mode is triggered by another topic, on which the desired state of the streaming mode (on or off and sampling rate) is published. The serial port node subscribes to this topic and will start or stop data transfer immediately when the state of the topic changes. In streaming mode, data is received by the serial port ROS node, parsed, and published to the continuous data topic, where it can be read by any client program or user. As shown in Fig. 9, the "Sensor ROS Node" publishes to the topics that control the serial port node and request the polling service. This node provides a high-level interface for the user. The benefits of using ROS are its effortless compatibility with any dVRK project, and its convenient structure, which suits the application well.

Figure 10. Latency and data throughput test for ROS package

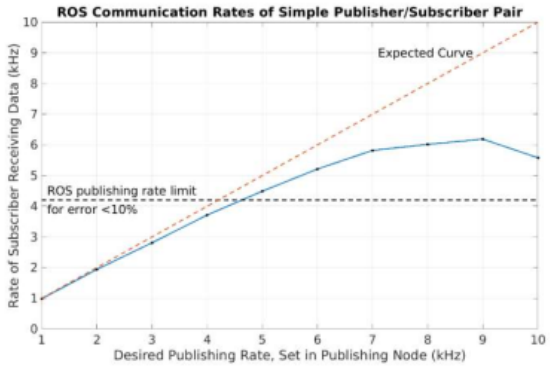


Similar to our approach for the Python libraries, latency and data throughput tests are conducted for 30,000 packets received from the sensor in polling and streaming modes. As Fig. 10 shows, the ROS implementation shows longer latency of close to 6 ms in polling mode. This is due to the extensive overhead associated with ROS services. However, the publisher update interval is much shorter when operating in streaming mode. As Fig. 10 shows, the publisher can report new data at the specified publish rate up to 2 kHz after which degradation of the publish rate is observed. According to [17], the dVRK Mid-Level Control typically runs at 1 kHz so the 2kHz limit mentioned above should not be a limitation.

In general, the maximum publishing frequency in ROS depends on the CPU speed, memory bandwidth, queue sizes, message size, OS internal network buffer size, and whether the C++ or Python client is used for ROS. In order to confirm that the 2 kHz publish rate is a ROS limitation for our dVRK setup, we ran a simple test in which two nodes are configured; one continuously publishes a single integer to a topic at a defined rate, and the other subscribes to the topic. The publish rate degradation is shown in Fig. 11. If better latency performance is needed, switching to Computer-

Integrated Surgical Systems and Technology (CISST) libraries is an option. Another option is integrating the stand-alone Python package into a control routine to directly use the sensor data. This approach eliminates the communication between two ROS nodes and improves latency performance.

Figure 11. ROS publish rate degradation



V. Conclusion

This paper outlines the electronics architecture, firmware architecture, and software interface of an optical force sensor that is under development for telesurgical haptics research but can be used in a wide variety of applications. It therefore should have low latency and high data throughput as required by the control loop. The electronics design is based on three custom boards: (1) Bicell boards for local signal conditioning and digitization to maximize noise performance, (2) Power/Com board that hosts an FPGA as the main processor and an IMU, (3) a flexible PCB that links the Bicell boards to the Power/Com board. The firmware architecture was developed for simultaneous sampling and parallel processing of the opto-electronic conversion channels and the IMU data. This allowed for a low latency operation and eliminated the risk of aliasing. A ROS package was developed for easy integration of the force sensor into dVRK. Another standalone Python library was also developed for sensor integration into other control platforms. Test results proved a latency of 1 ms and lower in polling and streaming modes which makes the sensor suitable for real time control applications.

In the next steps, the fabricated mechanical parts and electronics will be assembled, calibrated, and integrated into the dVRK for haptics research. Once we confirm the sensor functionality as intended, all the developments can be made available to the dVRK community for further research.

VI. Acknowledgement

Amir Hossein Hadi Hosseinbabadi would like to appreciatively recognize scholarship support from the NSERC Canada Graduate Scholarships-Doctoral program. Professor Salcudean gratefully acknowledges infrastructure support from CFI and funding support from NSERC and the Charles Laszlo Chair in Biomedical Engineering.

VII. References

[1] K. Hashtrudi-Zaad and S. E. Salcudean, "Analysis of control architectures for teleoperation systems with impedance/admittance master and slave manipulators," *Int. J. Rob. Res.*, vol. 20, no. 6, pp. 419–445, Jun. 2001.

[2] A. H. Hadi Hosseinabadi, M. Honarvar, and S. E. Salcudean, "Optical Force Sensing in Minimally Invasive Robotic Surgery," in *International Conference on Robotics and Automation*, 2019, pp. 4033–4039.

[3] "www.intuitivesurgical.com/Company," *Intuitive Surgical Web-site.* .

[4] J. C. Gwilliam, M. Mahvash, B. Vagvolgyi, A. Vacharat, D. D. Yuh, and A. M. Okamura, "Effects of haptic and graphical force feedback on teleoperated palpation," in *2009 IEEE International Conference on Robotics and Automation*, 2009, pp. 677–682.

[5] C. E. Reiley, T. Akinbiyi, D. Burschka, D. C. Chang, A. M. Okamura, and D. D. Yuh, "Effects of visual force feedback on robot-assisted surgical task performance," *J. Thorac. Cardiovasc. Surg.*, vol. 135, no. 1, pp. 196–202, 2008.

[6] A. L. Trejos, R. V Patel, and M. D. Naish, "Force sensing and its application in minimally invasive surgery and therapy: A survey," *Proc. Inst. Mech. Eng. Part C J. Mech. Eng. Sci.*, vol. 224, no. 7, pp. 1435–1454, Jan. 2010.

[7] L. Meli, C. Pacchierotti, and D. Prattichizzo, "Experimental evaluation of magnified haptic feedback for robot-assisted needle insertion and palpation," *Int. J. Med. Robot. Comput. Assist. Surg.*, vol. 13, no. 4, Dec. 2017.

[8] A. Talasaz, A. L. Trejos, and R. V. Patel, "The Role of Direct and Visual Force Feedback in Suturing Using a 7-DOF Dual-Arm Teleoperated System," *IEEE Trans. Haptics*, vol. 10, no. 2, pp. 276–287, Apr. 2017.

[9] S. Okamoto, M. Konyo, and S. Tadokoro, "Vibrotactile Stimuli Applied to Finger Pads as Biases for Perceived Inertial and Viscous Loads," *IEEE Trans. Haptics*, vol. 4, no. 4, pp. 307–315, Oct. 2011.

[10] J. K. Koehn and K. J. Kuchenbecker, "Surgeons and non-surgeons prefer haptic feedback of instrument vibrations during robotic surgery," *Surg. Endosc.*, vol. 29, no. 10, pp. 2970–2983, Oct. 2015.

[11] Y. Kamikawa, N. Enayati, and A. M. Okamura, "Magnified Force Sensory Substitution for Telemanipulation via Force-Controlled Skin Deformation," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 1–9.

[12] K. Bark *et al.*, "In Vivo Validation of VerroTouch: Tactile Feedback of Tool Vibrations for Robotic Surgery," *Surg Endosc.*, vol. 27, pp. 656–664, 2013.

[13] S. Shimachi, F. Kameyama, Y. Hakozaki, and Y. Fujiwara, "Contact force measurement of instruments for force-feedback on a surgical robot: Acceleration force cancellations based on acceleration sensor readings," in *Lecture Notes in Computer Science*, 2005, vol. 3750 LNCS, pp. 97–104.

[14] G. J. Garcí, C. A. Jara, J. Pomares, A. Alabdo, L. M. Poggi, and F. Torres, "A Survey on FPGA-Based Sensor Systems: Towards Intelligent and Reconfigurable Low-Power Sensors for Computer Vision, Control and Signal Processing," *Sensors*, vol. 14, pp. 6247–6278, 2014.

[15] P. Kazanzides, Z. Chen, A. Deguet, G. S. Fischer, R. H. Taylor, and S. P. Dimaio, "An open-source research kit for the da Vinci® Surgical System," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2014, pp. 6434–6439.

[16] "https://github.com/jhu-dvrk/sawIntuitiveResearchKit/wiki," *Johns Hopkins University.* .

[17] Z. Chen, A. Deguet, R. H. Taylor, and P. Kazanzides, "Software architecture of the da vinci research kit," in *Proceedings - 2017 1st IEEE International Conference on Robotic Computing, IRC 2017*, 2017, pp. 180–187.

API Documentation

sensor.py:

Public Methods

__init__(self,portNum=1, quick=False):

Opens a connection to the sensor a serial port and creates an object with which to control it

:param portNumIndex of port of device to open. Default value is 1, but 2 is also common. (See hardware setup)

:param quick a flag to tell the software to do a quick startup. This skips some initialization steps like reading the OFC and FSC registers and is not recommended.

disconnect(self)

Disconnects the sensor object from its serial port without destroying any of the data or configurations associated with the sensor object itself. This is useful if you wish to connect to the serial port directly with different software but want to save the sensor state, including OFC and FSC values, gain, etc.

start_data_transfer(self,data_rate=1500)

Start continuous transmission of data from the sensor by starting a thread that reads continuously without blocking and writes to a global FIFO buffer.

:param data_rate the rate at which the sensor should send data

:return string that indicates success or failure

stop_data_transfer(self)

Tell the sensor to stop sending continuous data. Also shut down the extra thread

:return string sent by the sensor

poll(self)

Request one measurement from the sensor

:return a parsed data packet in the form of a SensorOutput object

None if the packet somehow fails to arrive

reset_device(self)

Reset the device by sending the corresponding bytes to sensor

:return the sensor’s response as a string

reset_dac(self)
Reset the DAC by sending the corresponding byte to sensor
:return the sensor’s response as a string

reset_imu(self)
Reset the IMU by sending corresponding byte to sensor
:return the sensor’s response as a string

reset_ads(self,ads_num)
Reset a transducer. Then redoes the self-calibration and saves the OFC and FSC values
:param transducer_num the index of the transducer to reset, indexed 1-6
:return the sensor’s response as a string

deactivate_ads(self,ads_num)
Deactivate a transducer
:param ads_num the index of the transducer to deactivated, indexed 1-6
:return the sensor’s response as a string

config_dac(self, channel, voltage)
Configure the DAC by setting the output voltage of a specified channel or powering off a channel
:param channel the channel to set the voltage of (int between 0 and 5, inclusive)
:param voltage the voltage (in mV) to set the output of the selected channel to. Input 0 to shut a channel off
:return the sensor’s response as a string. This should be DAC-CNFG-S, DAC-PWUP-S, or DAC-PWDN-S.

turn_on_led(self,ledNum)
Turn on the given LED on the DAC. This is just a convenience function which calls config_dac.
:param ledNum the number of the LED to turn on (0-5)

turn_off_led(self,ledNum)
Turn off the given LED on the DAC. This is just a convenience function which calls config_dac.
:param ledNum the number of the LED to turn off (0-5)

turn_on_leds(self,ledNums)
Turn on the given LEDs on the DAC. This is just a convenience function which callsconfig_dac.
:param ledNums array of the numbers of the LEDs to turn on

turn_off_leds(self,ledNums)
Turn off the given LEDs on the DAC. This is just a convenience function which calls config_dac.
:param ledNums array of the numbers of the LEDs to turn off

turn_off_leds_all(self,ledNums)
Turn off the LEDs on the DAC. This is just a convenience function which calls config_dac.

config_imu(self, mode, delay)
Configure the IMU output
:param mode the type of output to set the IMU to output
Mode can be given as the string or int from the list below:
1 Accelerometer
2 Gyroscope
4 Linear Acceleration
5 Rotation Vector
8 Game Rotation Vector
The strings are not case sensitive and only the first 3 letters are considered
so 'acc', 'accel', 'Acc', 'acceleration', 'AcCeletometer' are all valid strings for mode 1
:param delay the interval between reports in ms
:return the sensor’s response as a string. A successful call to this should return IMU-CNFG-S.

set_imu_accelerometer(self, delay):
Convenience function to configure the IMU to output Accelerometer readings rather than calling config_imu()
:param delay the interval between reports in ms
The readings are a 5-component vector where the first three components are the x, y, and z components of the accelerometer readings and the last 2 components are 0

set_imu_gyroscope(self, delay):
Convenience function to configure the IMU to output Gyroscope readings rather than calling config_imu()
:param delay the interval between reports in ms
The readings are a 5-component vector where the first three components are the x, y, and z components of the gyroscope readings and the last 2 components are 0

set_imu_lin_accel(self, delay):
Convenience function to configure the IMU to output Linear Acceleration readings rather than calling config_imu()
:param delay the interval between reports in ms
The readings are a 5-component vector where the first three components are the x, y, and z components of the linear acceleration readings and the last 2 components are 0

set_imu_rotation(self, delay):
Convenience function to configure the IMU to output Rotation Vector readings rather than calling config_imu()
:param delay the interval between reports in ms
The readings are a 5-component vector where the first three components are the i, j, and k components of a unit quaternion, the fourth element is the real component of the quaternion, and the last element is an accuracy estimate

set_imu_game_rot(self, delay):
Convenience function to configure the IMU to output Game Rotation readings rather than calling config_imu()
:param delay the interval between reports in ms
The readings are a 5-component vector where the first three components are the i, j, and k components of a unit quaternion, the fourth element is the real component of the quaternion, and the last element is an accuracy estimate

imu_start_calibration(self):
Start the IMU calibration routine. During calibration, the sensor should be moved between 6 orientations, and the accuracy of the magnetometer and game rotation vectors monitored (parse_calibration_state). Once they are satisfactorily high, the calibration is complete and can be finished

(imu_finish_calibration). To cancel and stop without saving, call imu_cancel_calibration.

imu_cancel_calibration(self)

Cancel the IMU calibration and stop it without saving the result in the IMU. See imu_start_calibration for more details.

imu_finish_calibration(self)

Finish the calibration, saving the results to the IMU. See imu_start_calibration for more details.

parse_calibration_state(self)

Read the sensor's output during IMU calibration, parse it, and return the accuracy and type of the output. During calibration, the sensor continually and alternatingly sends the magnetometer and game rotation vector accuracy. Thus, this function should be called in a loop in a separate thread and the result printed, so the user carrying out the calibration can see when the accuracy of each is sufficiently high.

:return Status, statusType where status is the accuracy of the game rotation vector/magnetic field output and statusType is which mode is being reported (mag or grvec)

Accuracy is given as:

0	Unreliable
1	Accuracy Low
2	Accuracy Medium
3	Accuracy High

config_ads(self,ads_num, category, setting)

Configure the ADS1257-x device(s).

:param ads_num specifies which of the six ADSs to control

Indexing is 0-5 for ADS A-F. -1 targets every ADS at once

:param category what configuration category to change.

The following are the available categories:

0 'root' configure the root registers
1 'drate' configure the data rate
2 'pga' configure the PGA gain
3 'pos' configure the positive channel
4 'neg' configure the negative channel

These names are not case sensitive, and the category can be selected either by passing the string or the number from the above list

:param setting what to change the selected category's setting to
The following are the available settings, organized by their categories
0-Root Register Configuration:
This setting should be passed as a binary tuple or string. The first element or character
enables (1) or disables (0) ACAL. The second enables (1) or disables (0) IN-BUFF.
For example, (1,0) or '10' would both enable ACAL and disable IN_BUFF
1-Data Rate Configuration
The following rates in SPS are permitted. Any other number will be
rounded to the nearest value from this list:
2.5, 5, 10, 15, 25, 30, 50, 60, 100, 500, 1e3, 2e3, 3.75e3, 7.5e3, 15e3, 30e3
2-PGA Gain Configuration
The following PGA Gains are permitted. Any other number will be rounded to the nearest value from this list:
1, 2, 4, 8, 16, 32, 64
3-Positive Channel Configuration
Enter the int x to select AINx where x is 0, 1, 2, or 3
4-Negative Channel Configuration
Enter the int x to select AINx where x is 0, 1, 2, or 3
:return the sensor’s response as a string. A successful configuration should return ADSx-CNFG-S, where x is the ADS number.

set_ads_drate(self,ads_num,data_rate)
Convenience method that calls config_ads to set the data rate
:param ads_num which ADS to do this to (1-6)
:param data_rate the desired data rate. If this is not one allowed by config_ads, it will be rounded to the nearest allowed value

set_ads_registers(self,ads_num, ACAL, IN_BUFF)
Convenience method that calls config_ads to configure the root registers
:param ads_num which ADS to do this to (1-6)
:param ACAL 1 to enable ACAL, 0 to disable it
:param IN_BUFF 1 to enable IN_BUFF, 0 to disable it

set_pga_gain(self,ads_num, gain)
Convenience method that calls config_ads to set the PGA's gain

:param pga_num which PGA to do this to (1-6)
:param gain the desired gain value. If this is not one allowed by config_ads, it will be rounded to the nearest allowed value

set_ads_channel(self,ads_num, channel, positive=True)
Convenience method that calls config_ads to set the positive or negative channel number
:param ads_num which ADS to do this to (1-6)
:param positive
:param channel the desired channel number (0-3)
:param positive flag that states whether to change the positive or negative channel
True changes the positive channel, False does the negative one

Private for internal implementation:
__do(self, task, verbose=True,expect_package=False, with_crc8=True, toHex=True, regex=None):
Send an arbitrary command given by task and potentially read the response. This method is thread safe and acquires/releases a lock around its read/write operations so the serial communication does not get messed up
:param task byte message to send to device
:param verbose indicates whether or not to print the bytes received from the sensor in response to the command
:param expect_package indicates whether to attempt to read in and parse a 53 byte packet after sending the command
:param with_crc8 indicates whether to append a crc8 to the message (True), or not (False).
:param toHex indicates whether to convert the sensor's output to a human-readable hex string. This should be False if expect_package is true.
:param regex a regular expression for what the sensor response is expected to look like. If this is not None, the response will be searched for this regex, and only matches will be returned or printed.
:return parsed response from sensor if expect_package is True. Otherwise, a string containing the sensor's response

__reset(self,timeout=100)
Reset all input and output buffers until there are no more bytes waiting. If the buffer keeps filling up with new data after the reset, this will try timeout

times to clear the buffers before it gives up because something is continuously writing to the buffer
:param timeout number of reset commands to send before giving up

purge_rx(self, verbose=True):
Reset the internal and FTDI receive buffers
:param verbose prints a success message if the purge is successful and verbose is True

purge_tx(self, verbose=True):
Reset the internal and FTDI transmit buffers
:param verbose prints a success message if the purge is successful and verbose is True

purge(self, verbose=True):
Reset the internal and FTDI receive and transmit buffers
:param verbose prints a success message if the purge is successful and verbose is True

__check_crc(self, crc_, p, n=32)
Check CRC Checksum with 8 or 32 bits
:param crc_ the n bit checksum as a list of bytes (ints) or an int
:param p the packet to compare to the checksum. (This is bytes 0 to 46 in the 53 byte sensor packet)
:param n the number of bytes in the checksum (8 or 32)
:return True if the checksum matches, False otherwise

wait_for_packet(self, timeout=100, verbose=False)
Scan the incoming stream of data for the initialization byte. If it is found, read the next 2 bytes (count and checksum) and check the checksum of the init byte and count together. If this matches, return True, indicating the next 53 bytes are a data packet. If a number of bytes equal to timeout are read without finding the <start-byte, count, CRC> sequence of bytes, return False.
:param timeout number of bytes to scan through before giving up and returning False
:param verbose if True, print all the bytes that were read in while looking for the init byte
:return True if the next 53 bytes in the buffer are a data packet, false otherwise

read_packet(self, timeout=200)

Read an arbitrary set of bytes and return them as a list of integers

:param timeout if there is currently no available data, attempt to read bytes this many times before giving up. Also, if bytes are found, continue to read for more bytes until timeout bytes have been read in, or nothing was read in 20 consecutive times.

readLine(self, startChar=None, startChar2=None, endChar='\n', timeout=100)

Attempt to read one line from serial, starting with startChar, and ending with endChar

:param startChar the character to expect as the first character of the line. None by default allows any character to be the first character

:param startChar2 an alternative character which can also be the first character. If None, this is ignored.

:param endChar the character at which to stop reading. By default this is the new line character, '\n'

:param timeout how many bytes to read, looking for the startChar and then endChar before giving up

__parse(self,byte_data)

Parse the data packet of 53 bytes using the SH-2 structure explained in Data Packet Structure below

:param byte_data53-byte-long array of bytes in decimal form (array of 8-bit ints)

:return SensorOutput object which contains the parsed data in a useful form

to_int(self, byte,lsb_first=False)

Helper method to convert a list of bytes where the least significant byte is first or last into an int. If the LSB is first, set lsb_first to True. This is used for IMU data, for example. The default value of lsb_first is False.

:param byte the list of 8-bit decimal integers representing the number which is to be converted into a single integer

:param lsb_first flag to tell function whether LSB is first or last

:return the integer representation of the list of bytes

toBytesList(self, string)

Split a bytes object or hexadecimal string of any length into a list of bytes (intsin decimal). Since each byte is 2 characters (e.g. F0 or AB), the function checks that the string has an even number of characters. If not, it puts a 0 in the MSD, thus not changing the value and making sure each character is paired up properly into a byte. Note this is only valid for numbers where the MSD comes first or there is an even number of characters. Otherwise by adding the 0 we multiply each byte by 16.

:param string the byte string in hexadecimal. e.g. 'AF04BB9ED' without the 0x prefix

:return a list of bytes where each byte is represented by an 8-bit integer in decimal

toBytes(self,byteList)

Convert list of ints to a bytes object. E.g. [1,255,160] -> b'\x01\xff\xa0'. This uses struct.pack(num, "B") and will work identically in Python 2.7 or 3. This function is used to prepare a message for sending over the serial port.

:param byteList the list of ints to convert to a bytes object

toStr(self,byte_list, with_crc4=False)

Do the opposite of toBytesList. Convert a list of bytes (array of 8-bit ints in decimal) to a byte string in hex.

:param byte_list the list of bytes (as decimal ints) to convert to a byte string

:param with_crc8 True if a CRC8 checksum for the number given by byte_list should be appended to the string

:param format can take the value 0 (bytes) or 1 (string))

If format = 0, numbers will undergo the following conversion: 15 → 0x0f -> b'\x0f' which is a bytes object useful for sending to the sensor. This works in Python 2.7 and 3

If format = 1, numbers will be converted directly to a hex string: 15 -> '0f', which is actually 2 bytes b'\x30\x66'.

These are the ASCII values of the characters. This is not usable for sending to a sensor.

toHex(self, num)

Convert an int to hex. This is used instead of the standard hex() function because hex() adds "0x" to the beginning and sometimes randomly appends an "L" to the end.

:param num the int to convert to hex

readBytes(self,num_bytes)
Helper method to clean up read operations from the serial port. This takes the number of bytes we want and requests that number of bytes times a factor that is determined by whether the thing sending the bytes is sending them as strings or actual bytes. If it is sending as strings, each byte is represented by 2 characters and is thus actually sent as 2 bytes, hence the factor would be 2. The sensor-FTDI interface sends bytes objects, so the factor is 1. This function then converts the output to a byte list using toBytesList() and returns that.

:param num_bytes the number of bytes we wish to read in
:return list of bytes that was read in

read_thread Class
Class that inherits PyQt threading and defines the behavior of a separate thread to read data continuously without blocking the main thread. To start this thread, first initiate it using the constructor. Then use thread.start(), which executes the run function in a new thread.

The helper methods in this class are identical to those above, so they are not described a second time.

__init__(self,parent, port,threadLock,threadID=0, name='read_thread'):
Initialize the thread, nothing fancy. Parent should be the sensor object, while port is the serial port FTDI object (also accessible by parent.port)

run(self):
Checks the run flag. If it is true, it calls wait_for_packet (see above) and then reads in and parses the next packet If there is one. If there are any errors, it ignores them and proceeds, since the next packet will be coming right away. Packets that are successfully read and parsed are saved to the global LIFO buffer.

crc.py

File to calculate CRC checksums. Since we only need the 8 and 32 bit versions, not general solutions, none of this code can be used for an arbitrary number of bytes. In order to check a byte response, simply calculate the value and see if it equals the CRC that was sent with it.

crc4(p, polynomial=0x3, table=[])

Calculate the CRC4 checksum using one of two methods: if table is empty, use bitwise xor and shifting operations to find the CRC4 of p using the given polynomial. If the table is set, use it as a lookup table to calculate the checksum of p, with the polynomial 0x3

:param p the list of bytes (8-bit decimal integers) to find the CRC of
:param polynomial the polynomial dividend used to divide up at each step of the bitwise operation
:return the 4-bit CRC checksum

crc8(p, polynomial=0x07, table=[])

Calculate the CRC8 checksum using one of two methods: if table is empty, use bitwisexorand shifting operations to find the CRC8 of p using the given polynomial. If the table is set, use it as a lookup table to calculate the checksum of p, with the polynomial 0x07

:param p the list of bytes (8-bit decimal integers) to find the CRC of
:param polynomial the polynomial dividend used to divide up at each step of the bitwise operation
:return the 8-bit CRC checksum

crc32(p, polynomial=0x04C11DB7, table=[])

Calculate the CRC32 checksum using one of two methods: if table is empty, use bitwise xor and shifting operations to find the CRC32 of p using the given polynomial. If the table is set, use it as a lookup table to calculate the checksum of p, with the polynomial 0x04C11DB7

:param p the list of bytes (8-bit decimal integers) to find the CRC of
:param polynomial the polynomial dividend used to divide up at each step of the bitwise operation
:return the 32-bit CRC checksum

calculate_CRC4_table()

Calculate the lookup table to use in the above crc4 function
CRC4 with lookup table is solidly faster than without.

calculate_CRC8_table()

Calculate the lookup table to use in the above crc8 function

calculate_CRC32_table()

Calculate the lookup table to use in the above crc32 function
CRC32 with lookup table is more than 2x faster than without

sensor_output.py

When the sensor software reads and parses data from the sensor, it is saved as a SensorOutput object. This takes in the bytes received from the sensor, parses them into useful information, and makes that information accessible in one object. The class members that can be accessed are:

byte: the raw bytes received
differential: the differential voltage signal in volts
differential_raw: the differential signal in bits
sum: the sum signal in volts
sum_raw: the sum signal in bits
imu: the IMU output
quaternion: the quaternion given by the IMU if the IMU mode is rotation vector or game rotation
rotation: the rotation matrix calculated from the quaternion if the IMU mode is rotation vector or game rotation
report_id: the report ID (indicates the IMU mode) as a number
report_id_str: the IMU mode as a string
temperature: the temperature of the sensor
wrench: the measured wrench, calculated from the difference and sum signals and transformed with the given calibration matrix into N and N.mm
saturated: a boolean which shows whether a channel is saturated
saturatedChan: list of all 6 channels. If channel i is saturated, saturatedChan[i] is 1. Otherwise it is 0.

There are also a few useful methods beyond the ones used internally for parsing:

failed(self)
Check if the parsing was successful and this is a valid SensorOutput
:return True if the provided set of bytes could not be parsed – i.e. parsing failed.

to_array(self, detailed=False)
Convert the most important data to an array organized as [<wrench>, <imu>, <temperature>] if detailed is False. If detailed is True, it is: [<wrench>, <difference>, <sum>, <imu>, <temperature>]. Here wrench, difference, sum are 6 elements each, imu is 4 elements, and temperature is a single float.
:param detailed read above
:return array form of sensor output. Read above.

string(self, detailed=False)
Print the sensor output in a readable manner. If detailed is True, include the actual and raw difference and sum signals. Otherwise print only the wrench, IMU output, and temperature, as well as a warning if something is saturated.

:param detailed read above

:return Nothing, The output is printed

Further Documentation

Further documentation can be found in the following documents:

Sensor Overview:
https://ieeexplore.ieee.org/document/8793589

ADS1257:
http://www.ti.com/lit/ds/symlink/ads1257.pdf

IMU:
https://cdn.sparkfun.com/assets/4/d/9/3/8/SH-2-Reference-Manual-v1.2.pdf

DAC 605082:
http://www.ti.com/lit/ds/symlink/dac80508.pdf

Cyclic Redundancy Checks:
https://en.wikipedia.org/wiki/Computation_of_cyclic_redundancy_checks

FTDI USB-RS485 Plus2:
https://www.ftdichip.com/Support/Documents/DataSheets/Modules/DS_USB-COM485-PLUS2.pdf

libftdi:
https://www.intra2net.com/en/developer/libftdi/documentation/group__libftdi.html