A Motion-Controlled Endoscope for the da Vinci Surgical System

Written by Daniel Ingram

In fulfillment of the requirement for the Marshall Plan Scholarship

TABLE OF CONTENTS

THE DA VINCI RESEARCH KIT

Overview of the da Vinci Surgical System

The da Vinci Surgical System is an FDA-approved surgical robot that has been used since 2000 for procedures ranging from prostatectomies to cardiac valve repair.[1,2,3] It was used in an estimated two-hundred thousand operations in 2012 alone and, as of 2014, there were over three thousand units installed in operating rooms worldwide.[4]

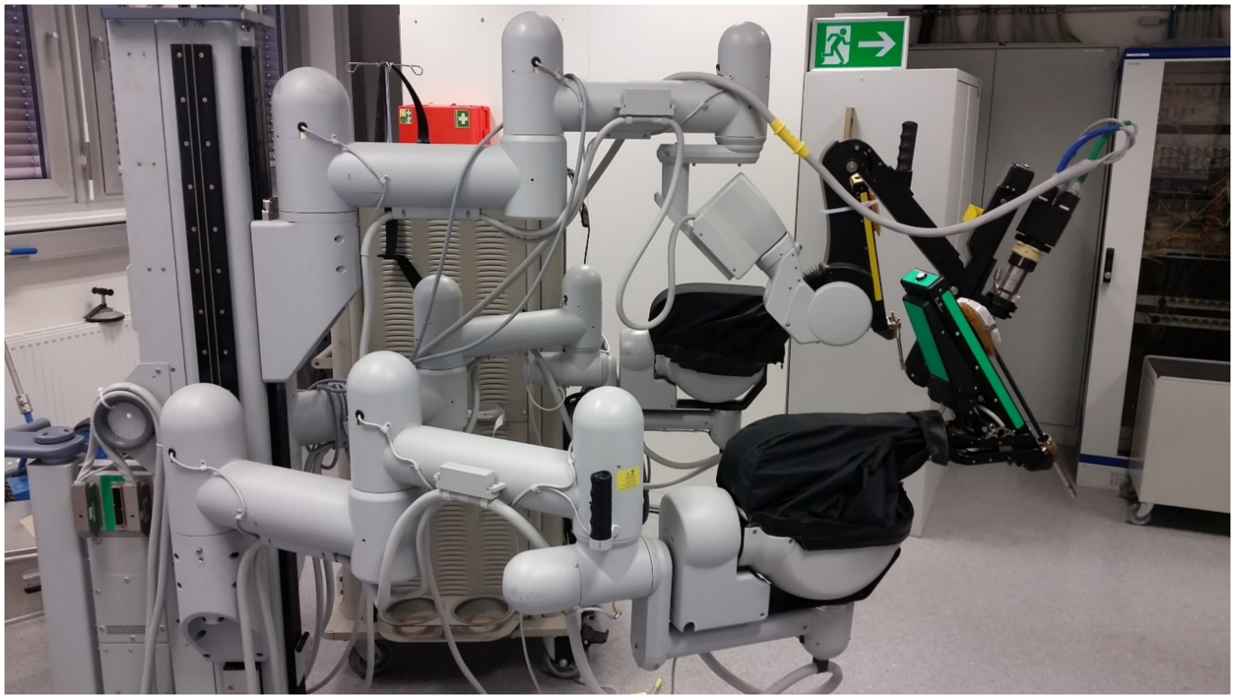


Figure 1. The first-generation da Vinci Surgical System patient-side cart features two robotic arms for manipulating surgical instruments and one robotic arm for controlling an endoscope camera.

The system enables a minimally invasive approach to traditional laparoscopic surgeries by allowing entire operations to be performed through relatively small incisions. In traditional laparoscopic surgeries, a larger incision is made in the patient's body and the surgeon performs the operation while standing using long-shafted instruments and viewing their movements on a nearby video screen. These surgeries typically last many hours and can be quite exhausting for the surgeon.

In contrast, the da Vinci Surgical System allows the surgeon to perform the entire operation while seated at an ergonomic console. The surgeon rests their head on a soft pad in a downward-facing position, as it would be if they were performing the surgery directly, and views their movements through a high-resolution, stereoscopic display of the video feed from the endoscopic camera inserted in the patient. The camera can be controlled with the Endoscopic Camera Manipulator (ECM). The surgical instruments are mounted on robotic arms known as patient-side manipulators (PSMs), and the surgeon controls these instruments using the master-tool manipulators (MTMs) on the console. The MTMs are placed where the surgeon's hands would be if they were performing the surgery directly. The motion of the MTMs is translated to scaled motion of the PSMs with seven degrees of freedom, and the PSMs also feature tremor cancellation to reduce potential shakiness of the surgeon's hands.

Though an incredible technological achievement, the da Vinci Surgical System is not perfect, so the project described in this paper was undertaken to implement a potential improvement to the system: seamless and natural control of the surgeon's viewing field using an Oculus Rift headset. The surgeon already controls the surgical instruments as they would if they were handling them directly, but controlling the endoscope camera is not done so intuitively. By instead mapping the orientation of an Oculus Rift headset to the endoscope camera, the surgeon would be able to look around the area of surgery in an entirely natural manner. The da Vinci Research Kit was used to facilitate the implementation, and mapping of the orientation of the Oculus Rift headset to a PSM was successful.

Overview of the da Vinci Research Kit

The da Vinci Research Kit (dVRK) began as an attempt to create an open-source telerobotics research platform from an existing complete telesurgical system. Because the da

Vinci Surgical System is a proprietary product, however, it was initially closed off to researchers, meaning that entirely new controller hardware had to be designed and fabricated to allow complete access to all levels of control.

The hardware was based on an approach known as *centralized computation and distributed I/O*, by which a real-time communication network allows all control computations to be implemented on a high-performance computer while keeping the I/O distributed, thereby preserving the advantages of reduced cabling.[5] This approach was implemented by using a Field-Programmable Gate Array (FPGA) to provide a low-latency interface between the high-speed IEEE-1394a serial network and the I/O hardware. IEEE-1394a, also known as FireWire, was chosen as the communication protocol because “it is widely available, has high performance (up to 400 Mbits/sec), supports daisy-chaining at the physical layer, and there is ample documentation to enable implementation of the link-layer protocol on an FPGA.”[5] Commands from dVRK software are issued via the IEEE-1394a bus to an FPGA which contains firmware that is responsible for converting these high-level commands into low-level hardware control. The hardware is controlled by a Quad Linear Amplifier (QLA) board which contains the necessary power electronics for driving the joint motors.

Though the newest model of the da Vinci Surgical System includes two MTMs, three PSMs, and one ECM, the actual dVRK includes only two MTMs and two PSMs. Nevertheless, it was straightforward to integrate the dVRK electronics and software with a full da Vinci Surgical System. To control a single manipulator, two FPGA1394-QLA controller boards were necessary and each of these boards required a unique ID to be properly addressed by IEEE-1394a. Because there are sixteen possible addresses on the onboard rotary switch, however, there were enough addresses for all the non-dVRK manipulators to be included in the controller chain.

The dVRK Hardware

IEEE-1394a

IEEE-1394a is a high-speed serial bus that supports both isochronous and asynchronous data transfer modes. For the dVRK, the asynchronous mode was chosen because of the relative ease of FPGA implementation and its sufficiency for a 1kHz servo control rate. Only a subset of the IEEE-1394a link-layer protocol was included in the firmware to conserve FPGA resources.

Unlike USB, IEEE-1394a allows multiple devices to be connected serially on the same bus. For this reason, each device must have a unique identification number. In the case of the dVRK, this unique ID is specified by an onboard rotary switch with a hexadecimal range of 0 to F. The recommended board IDs for the dVRK are given in Table 1.

Table 1.[6]

	MTML	MTMR	ECM	PSM1	PSM2	PSM3	Setup Joints
Board 1 ID	0	2	4	6	8	A	C
Board 2 ID	1	3	5	7	9	B	

FPGA Controller

The FPGA controller board is the interface between the QLA board and the control computer. Its main components are:

- One Xilinx Spartan-6 XC6SLX45-2 FPGA
- One configuration PROM
- IEEE-1394a physical layer
- Two IEEE-1394a 6-pin connectors
- Low-speed USB interface
- JTAG interface

The FPGA board also contains two 44-pin connectors for providing power and I/O to the QLA board. The FPGA is loaded with firmware which has three main responsibilities:

1. Communication with the computer via the IEEE-1394 bus

2. Control of I/O devices
3. Hardware-level safety checking

Figure 2 is a diagrammatic representation of how the FPGA firmware handles received IEEE-1394a packets. Upon reception of a packet, the firmware performs a cyclic redundancy check to ensure that there were no errors in transmission. If the packet is not valid, it is ignored. The header of a valid packet is checked to determine whether it is a read or write request. For write requests, an acknowledgement bit is sent, the desired data is written to a register via the Serial Peripheral Interface (SPI) protocol, and the digital-to-analog converter (DAC) is triggered. A common example of a write request would be setting the desired motor current. For a read request, the data at the specified address is read and sent back to the IEEE-1394a master via the SPI protocol along with an acknowledgement bit. This data is mainly the output of the analog-to-digital converters (ADC). Because one ADC conversion cycle takes approximately 0.7μs, the FPGA firmware continuously requests conversions and stores them in local registers to reduce latency.

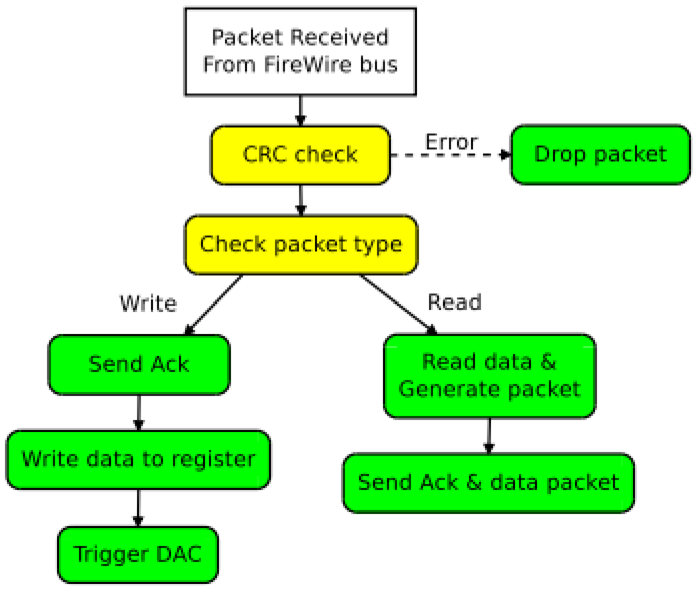


Figure 2. The algorithm for handling received IEEE-1394a data packets.[5]

Quad Linear Amplifier

The QLA board contains all the electronics necessary for controlling up to four joints of a da Vinci manipulator. Because the manipulators have seven degrees of freedom, two QLA boards and two FPGA boards are used for each one, with one channel left unused. The main components of each QLA channel are:

- One 16-bit digital-to-analog converter (DAC) for setting the desired motor current
- Two 16-bit analog-to-digital converters (ADC) to digitize the measured motor current and the voltage across a potentiometer used for absolute feedback of the motor position
- Differential receivers for a quadrature encoder (for incremental feedback of the motor position) with A, B, and Z channels
- Two power operational amplifiers to provide bidirectional control of a motor for a single DC power supply
- Digital inputs for one home and two limit switches
- One open-collector digital output with a high current drive

By providing two types of positional feedback from the motors, hardware issues related to the potentiometer and quadrature encoder can be easily detected. The potentiometer also allows an absolute reference for the motor position.

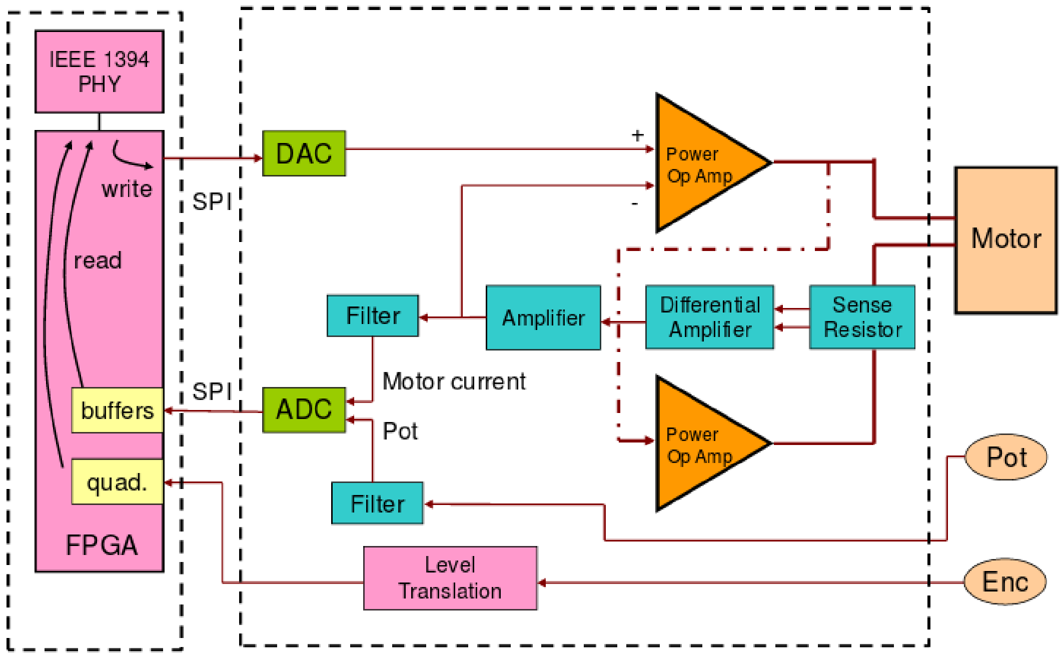


Figure 3. A high-level QLA circuit diagram.[5]

For each QLA channel, the FPGA sets a desired motor current and writes it to the DAC via SPI. The actual current through the motor flows through a sense resistor, resulting in a differentially amplified voltage which is used for current feedback. This feedback and the potentiometer voltage is converted to a digital signal with the ADC and written to an FPGA buffer via SPI. The quadrature encoder signal from the motor is level-shifted before being decoded by the FPGA.

The dVRK Software

Computer-Integrated Surgical Systems and Technology

The Computer-Integrated Surgical Systems and Technology (*cisst*) software package is a collection of C++ libraries designed to facilitate the development of computer-integrated surgical systems. The libraries contained in the *cisst* package are listed with a brief description of their purpose in the Table 2 below.

Table 2.[7]

Library	**Purpose**
cisstCommon	Logging, class/object registries, serialization/deserialization
cisstVector	Linear algebra and spatial transformation in 2D/3D
cisstNumerical	Thread-safe numerical methods
cisstInteractive	Scripts for Interactive Research Environment (IRE)
cisstOSAbstraction	Operating system services
cisstMultiTask	Defining tasks/devices/interfaces/objects
cisstStereoVision	Mono/stereo video acquisition/processing/display
cisstParameterTypes	Data types for objects in component-based framework
cisstRobot	Robot control elements e.g. trajectory control
cisst3DUserInterface	Support for 3D input/display

These libraries are central to the operation of the da Vinci Surgical System, as the rest of the dVRK software depends heavily on them. All *cisst* libraries except *cisstStereoVision* and *cisst3DUserInterface* were relevant to this project, though *cisstStereoVision* will likely be useful for future improvements, such as streaming the endoscope camera video to the Oculus Rift headset.

Surgical Assistant Workstation

Together with *cisst*, on which it is based, the Surgical Assistant Workstation (*saw*) package was used to facilitate control of the da Vinci Surgical System manipulators. For this project, the relevant *saw* components were *sawRobotIO1394* and *sawIntuitiveResearchKit*. The *sawRobotIO1394* component contains code for interfacing with the QLA board via IEEE-1394a, and it was used for calibration of the manipulators and to verify that the FPGA1394-QLA controller boards were working properly by allowing direct control of the manipulator joint positions through *sawRobotIO1394QtConsole*. *Cisst* and *saw* are bundled together as the single package *cisst-saw* for the dVRK.

Robot Operating System

The Robot Operating System (ROS) is a framework of libraries and drivers for the development of robotics systems. The ROS interfaces of the dVRK allow the da Vinci manipulators to be controlled by any conceivable means.

ROS is based on a modular architecture in which the primary constituents are nodes and topics. A topic is simply an efficient channel for data, known as messages in ROS, to be passed between nodes. Nodes are programs, usually with only one or two responsibilities, that can both publish messages to topics and subscribe to topics. Publishing and subscribing is ROS terminology for writing and reading, respectively. This publisher-subscriber model is ideal for robotic systems, where there are often many sensors to be read and many components to be controlled. By compartmentalizing simple tasks into nodes, ROS allows users to develop highly efficient systems.

Using topics, ROS allows dVRK users to control the manipulators in any way possible using publisher nodes. For this project, a node was written which subscribed to the Oculus Rift orientation data and converted it to the appropriate message type before publishing it to a topic that controlled the desired manipulator. The ROS interface for the dVRK is *dvrk-ros*.

INTEGRATING THE DVRK WITH A FULL DA VINCI SURGICAL SYSTEM

Assembling the FPGA1394-QLA Controller Boards

To save on the overall cost of this project, it was decided that the controller boards would be assembled onsite, rather than ordered prebuilt from the dVRK supply chain; this decision saved MCI approximately €34,000. The printed-circuit board (PCB) layouts were downloaded from the JHU-CISST GitHub repository and sent to Multi-CB for manufacturing.

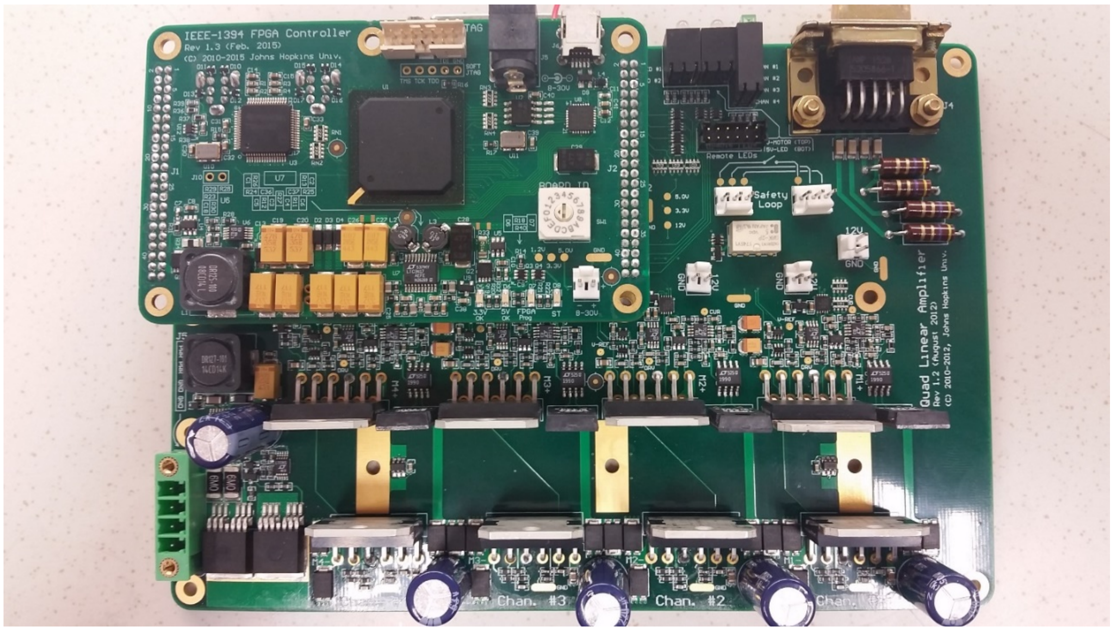

IEEE-1394 FPGA Controller
Rev 1.3 (Feb. 2015)
(C) 2010-2015 Johns Hopkins Univ.
Safety Loop
Quad Linear Amplifier
Rev 1.2 (August 2012)
(C) 2010-2012, Johns Hopkins Univ.
Chan. #3
Chan. #2

Figure 4. A completed FPGA1394-QLA controller board. The FPGA1394 board is mounted by two 44-pin connectors on the top-left section of the QLA board.

A single FPGA1394-QLA controller board contains 628 components with 156 different component types. These components were ordered from Digi-Key, Mouser, Linear Technology, and McMaster-Carr using the Bill Of Materials (BOM) from the JHU-CISST GitHub repository. Using a stencil, solder paste was applied to the PCBs and components were then placed using an LPKF ProtoPlace S. To melt the paste and form an electrical and mechanical connection between the components and the PCB, the board was baked in an LPKF ProtoFlow S with a preheat temperature and time of 160^{o}C and 170 seconds, respectively; a reflow temperature and time of 260^{o}C and 140 seconds, respectively; and a cooling time of 100 seconds. This was initially only done with two FPGAs and two QLAs so that they could be tested with a manipulator and any hardware issues could be solved before assembling the remaining boards.

Setting up the Software

Ubuntu

Ubuntu 14.04 was chosen for the operating system of the control computer, as it is well-supported by ROS Indigo, the recommended ROS distribution for the dVRK. An ISO image of Ubuntu 14.04.5 LTS AMD64 was downloaded and burned to a flash drive using Startup Disk Creator. The advantage to burning the image with Startup Disk Creator was that it automatically made the flash drive bootable, which simply allowed the computer to be booted from the flash drive rather than a preinstalled operating system. The flash drive was inserted into the computer and selected from the BIOS on boot to install Ubuntu. The computer contained 16GB of DDR3 1600MHz RAM and a 3.6GHz Intel i7-4790 CPU, so it was well-suited for the task of the *dvrk-ros* master, though it would later be necessary to use a computer with a better graphics card for interfacing to the Oculus Rift.

IEEE-1394a Library

The library for IEEE-1394a communication, *libraw1394-dev*, was installed using the Ubuntu command line interface.

```
$     sudo apt-get install libraw1394-dev
```

This library is an interface to the kernel side of the Linux IEEE-1394a subsystem, which allows direct access to connected IEEE-1394a buses. To avoid having to access the library as root, which could cause problems with the dVRK software, permissions were set for the library to allow the current user to access it.

```
$     sudo mkdir /etc/udev/rules.d
$     cd
$     echo ‘KERNEL==“fw*”,GROUP=“fpgaqla”,MODE=“0660”’ > ~/80-firewire-fpgaqla.rules
$     sudo mv ~/80-firewire-fpgaqla.rules /etc/udev/rules.d/80-firewire-fpgaqla.rules
$     sudo addgroup fpgaqla
$     sudo adduser ‘whoami’ fpgaqla
```

To ensure that the IEEE-1394a library was successfully installed, the output of

```
$     lsmod | grep ‘firewire\|1394’
```

was verified to be

```
      firewire_ohci 40551 0
      firewire_core 68769 1 firewire_ohci
      crc_itu_t 12707 1 firewire_core
```

Several additional tools for testing and troubleshooting IEEE-1394a devices were also installed.

```
$     cd ~
$     git clone https://github.com/jhu-cisst/mechatronics-software.git
$     cd mechatronics-software/util
$     make
```

This repository included the command-line tools *info1394*, *block1394*, *quad1394*, and *time1394*, all of which provided some function for testing IEEE-1394a devices. These tools were used primarily to verify that the IEEE-1394a physical layer of the controller boards had been assembled correctly.

ROS Indigo

To install ROS, the control computer first had to be setup to receive packages from packages.ros.org.

```
$    sudo sh -c ‘echo “deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main” >
     /etc/apt/sources.list.d/ros-latest.list
$    sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net --recv-key 0xB01FA116
$    sudo apt-get update
```

The full desktop install of ROS indigo was then initiated.

```
$    sudo apt-get install ros-indigo-desktop-full
```

Once ROS Indigo was installed, *rosdep* was used to install system dependencies that are required by ROS components. Normally, this process can be quite difficult, but *rosdep* is a command-line tool that automates the process.

```
$    sudo rosdep init
$    rosdep update
```

The final step performed in the installation of ROS was the sourcing of the ROS Indigo *setup.bash* file. By sourcing this file, ROS environment variables and tools were available via the command line. The source command was added to the *~/.bashrc* file so that *setup.bash* would be sourced in all future terminal sessions.

```
$    echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
$    source ~/.bashrc
```

Catkin Workspace

Catkin is the official build system for ROS. Build systems are responsible for generating usable targets from raw source code and, as such, one is necessary for building ROS packages. The dVRK packages are intended to be built with the command *catkin build* rather than the more widely used *catkin_make*, so the Python package *python-catkin-tools* had to be installed.

```
$    sudo apt-get install python-catkin-tools
```

Following this, a *catkin* workspace was created and initialized. The workspace is simply a directory for creating, modifying, building, and installing ROS packages.

```
$    mkdir -p ~/catkin_ws/src
$    cd ~/catkin_ws
$    catkin init
```

Because it was recommended by the dVRK community to significantly improve the performance of *cisst-saw* and *dvrk-ros*, the build type for the *catkin* workspace was set to *Release*. Once this was done, the workspace was built with the *catkin build* command to initialize the build and development directories. Finally, the *setup.bash* file in the development directory was sourced so that packages in this workspace would be recognized by the ROS command line tools.

```
$    catkin config –profile release -x_release
$    catkin profile set release
$    catkin config –cmake-args -DCMAKE_BUILD_TYPE=Release
$    catkin build
$    echo “source devel_release/setup.bash” >> ~/.bashrc
$    source ~/.bashrc
```

Cisst-saw

Because the *dvrk-ros* package depends on both *cisst* and *saw*, the *cisst-saw* package had to be downloaded and built first. *Cisst-saw* has many dependencies that were installed before proceeding with the build.

```
$    sudo apt-get install libxml2-dev libncurses5-dev qtcreator swig libopenigtlink-dev flite cmake-curses-gui
     cmake-qt-gui libopencv-dev subversion gfortran libcppunit-dev fluid
$    cd ~/catkin_ws/src
$    git clone https://github.com/jhu-cisst/cisst-saw --recursive
```

To successfully build *cisst-saw* with *catkin*, however, a few more steps were necessary. First, a separate CMake file had to be downloaded and run with *cmake*. After running *make* on the results of *cmake*, one of the generated shell files had to be sourced. Finally, the paths to several CMake files had to be explicitly defined and exported. After doing these steps in that order, *cisst-saw* was built with *catkin*.

```
$    mkdir build
$    cd build
$    wget https://raw.githubusercontent.com/jhudvrk/sawIntuitiveResearchKit/share/dvrk.cisst.initial.cmake
$    cmake ../
$    cd ..
$    make
$    echo “source ~/catkin_ws/src/cisst-saw/build/cisstvars.sh” >> ~/.bashrc
$    source ~/.bashrc
$    export sawRobotIO1394_DIR = “~/catkin_ws/src/cisst-saw/sawRobotIO1394/”
$    export sawIntuitiveResearchKit_DIR = “~/catkin_ws/src/cisst-saw/sawIntuitiveResearchKit/”
$    export sawTextToSpeech_DIR = “~/catkin_ws/src/cisst-saw/sawTextToSpeech/”
$    export sawControllers_DIR = “~/catkin_ws/src/cisst-saw/sawControllers/”
$    catkin build
```

Dvrk-ros

The process to build *dvrk-ros* was much simpler than that of *cisst-saw*, as it simply needed to be cloned and built. The reason for this is that much of *dvrk-ros* depends on *cisst-saw*, so as long as *cisst-saw* is built correctly, *dvrk-ros* will have no problems being built.

```
$    cd ~/catkin_ws/src
$    git clone https://github.com/jhu-dvrk/dvrk-ros
$    catkin build
```

Dvrk-ros is not yet part of the *cisst-saw* package as it contains approximately 30MB of CAD files that are irrelevant to users of *cisst-saw*.

Installing FPGA Firmware

Though the FPGA firmware contains a module that allows itself to be updated via the IEEE-1394a interface, it had to initially be uploaded via the onboard JTAG interface. The JTAG interface is much slower and unfortunately requires a more expensive cable. However, it was necessary for the initial upload of the FPGA firmware to the onboard PROM.

To upload firmware via the JTAG interface, a Xilinx Integrated Synthesis Environment (ISE) was used. Because the newest Xilinx ISE, *Vivado*, does not support the Spartan-6 FPGA family, for which the firmware was written, the older Xilinx ISE 14.7 had to be used instead. The software package was downloaded from the Xilinx website using the segmented download option, which downloads the package as four smaller zipped files rather than a single large zipped file. Doing this allowed the installer to check the integrity of each package before attempting to install its constituents. Once the install was complete, the JTAG cable drivers and libraries were installed.

```
$   sudo apt-get install libusb-dev
$   sudo apt-get install git-core gitk git-gui libusb-dev build-essential libc6-dev-i386 fxload
$   cd ~
$   git clone git://git.zerfleddert.de/usb-driver
$   cd usb-driver/
$   make
$   sudo ./setup_pcusb ~/14.7/ISE_DS/ISE/
```

The following lines were added to *~/.bashrc*.

```
LM32_TOOLS=/opt/gcc-lm32/bin
XILINX=~/14.7/ISE_DS
export XILINX
HOST_PLATFORM=lin64
export HOST_PLATFORM
XILINX_BIN=${XILINX}/ISE/bin/${HOST_PLATFORM}
export XILINX_BIN
PATH=${PATH}:${XILINX_BIN}:${LM32_TOOLS} #XILINX_EDK=${XILINX}/ISE/EDK
XILINX_CSE_TCL=${XILINX}/ISE/cse/tcl
export XILINX_CSE_TCL
export XKEYSYMDB=/usr/share/X11/XKeysymDB export DISPLAY=:0

function loadXilinx() {
 # use settings32.sh if your system is 32-bit
 if [ -f ~/14.7/ISE_DS/settings64.sh ]; then
               . ~/14.7/ISE_DS/settings64.sh
 fi
}
```

The *~/.bashrc* file was then sourced to apply these changes. The repository containing the necessary firmware files was cloned.

```
$   git clone https://github.com/jhu-cisst/mechatronics-firmware.git
```

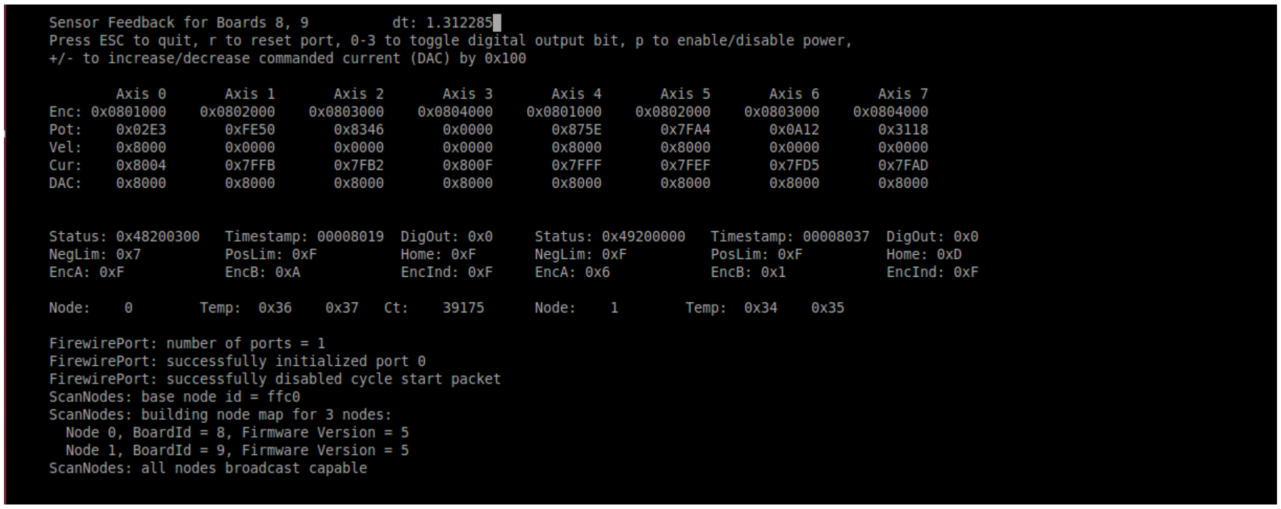
The most important file in the repository is *FPGA1394-QLA.mcs*, as this was the hex file that was uploaded to the PROM. The cable drivers had to be preloaded before opening Xilinx *iMPACT*, which was used to upload the firmware.

```
$    export LD_PRELOAD=~/usb-driver/libusb-driver.so
$    cd ~/14.7/ISE_DS/ISE/bin/lin64
$    ./impact
```

The *FPGA1394-QLA.ipf* iMPACT programming file, which contains information about how to program the PROM, was opened. The *Boundary Scan* screen, which opens by default, contains a high-level schematic of the FPGA controller architecture. On this screen, *FPGA1394-QLA.mcs* was assigned as the configuration file for the M25P16 SPI PROM device. The firmware was then uploaded. With the firmware now installed, it could be updated in the future via the IEEE-1394a interface, which is a much simpler process. The *cisst-saw* package contains a command line tool, *pgm1394*, which condenses the entire firmware upload process to one line.

```
$    pgm1394 <board-id> <path to FPGA1394-QLA.mcs>
```

Because the IEEE-1394a interface is used for this method, the board ID must be specified, whereas it did not have to be for the JTAG method. *Cisst-saw* contains another command line tool, *qladisp*, for verifying the boards are working correctly. *Qladisp* is also invaluable for troubleshooting, as it provides hardware feedback and FPGA status information.



```
Sensor Feedback for Boards 8, 9         dt: 1.312285
Press ESC to quit, r to reset port, 0-3 to toggle digital output bit, p to enable/disable power,
+/- to increase/decrease commanded current (DAC) by 0x100

        Axis 0       Axis 1       Axis 2       Axis 3       Axis 4       Axis 5       Axis 6       Axis 7
Enc: 0x0801000    0x0802000    0x0803000    0x0804000    0x0801000    0x0802000    0x0803000    0x0804000
Pot:    0x02E3       0xFE50       0x8346       0x0000       0x875E       0x7FA4       0x0A12       0x3118
Vel:    0x8000       0x0000       0x0000       0x0000       0x8000       0x8000       0x0000       0x0000
Cur:    0x8004       0x7FFB       0x7FB2       0x800F       0x7FFF       0x7FEF       0x7FD5       0x7FAD
DAC:    0x8000       0x8000       0x8000       0x8000       0x8000       0x8000       0x8000       0x8000


Status: 0x48200300   Timestamp: 00008019  DigOut: 0x0     Status: 0x49200000   Timestamp: 00008037  DigOut: 0x0
NegLim: 0x7          PosLim: 0xF          Home: 0xF       NegLim: 0xF          PosLim: 0xF          Home: 0xD
EncA: 0xF            EncB: 0xA            EncInd: 0xF     EncA: 0x6            EncB: 0x1            EncInd: 0xF

Node:    0        Temp:  0x36    0x37   Ct:    39175      Node:    1        Temp:  0x34    0x35

FirewirePort: number of ports = 1
FirewirePort: successfully initialized port 0
FirewirePort: successfully disabled cycle start packet
ScanNodes: base node id = ffc0
ScanNodes: building node map for 3 nodes:
  Node 0, BoardId = 8, Firmware Version = 5
  Node 1, BoardId = 9, Firmware Version = 5
ScanNodes: all nodes broadcast capable
```

Figure 5. Screenshot of the output of *qladisp*.

Connecting the Controller Boards to the da Vinci Surgical System

The FPGA1394-QLA controller boards do not connect directly to the da Vinci Surgical System manipulators; instead, the controller board must be connected to a da Vinci Manipulator Interface Board (dMIB), which itself is connected to a manipulator. Because the design for the dMIB is not open-source, however, the dMIBs could not be fabricated onsite, so they were ordered from the dVRK supply chain.

For this project, the relevant components of each dMIB were two VHDC168-F, two DB9-M, and one DL1-156R-F connector. One dMIB was used per manipulator, and it was connected to the manipulator’s DL1-156R-M cable. Each dMIB was connected to two FPGA1394-QLA controller boards through the VHDC168-F and DB9-M connectors for motor signals and motor power, respectively.

Calibration

I/O Configuration File Generation

The programs in the *cisst-saw* and *dvrk-ros* packages depend on an I/O configuration file, which contains information about the individual joints of the da Vinci Surgical System manipulators. The MATLAB program *configGUI* was used to generate these files. *ConfigGUI* requires the calibration file for a manipulator before its I/O configuration file can be generated. The calibration files for PSM1, PSM2, ECM, MTML, and MTMR were obtained through a contact at Intuitive Surgical, Inc. After selecting the appropriate calibration file from within *configGUI*, the manipulator name was specified and the I/O configuration file was then generated. The board IDs and digital input settings were left as their default values.

Current Calibration

To ensure that the motor current requested by the FPGA would be the current that actually drove the motors, a current calibration was performed. The command line program *sawRobotIO1394CurrentCalibration* was used to complete the calibration. The syntax for this program is

```
$     sawRobotIO1394CurrentCalibration -c <path to I/O configuration file>
```

The path to the I/O configuration file had to be provided so that the current offset values could be updated following the calibration. The current calibration was done by requesting a null current on each manipulator joint, reading the current feedback, and setting the feedback as the offset. Performing this current calibration ensured that the requested current would be the actual current in the future.

Testing with sawRobotIO1394QtConsole

After updating the I/O configuration file with the correct current offsets, the FPGA1394-QLA controller board set was tested on a da Vinci Surgical System manipulator to ensure proper operation using the program *sawRobotIO1394QtConsole*. The syntax for this program is

```
$     sawRobotIO1394QtConsole -c <path to I/O configuration file>
```

This program opened a GUI from which the individual joints of the specified manipulator could be controlled with sliders. The GUI also provided the same feedback as the *qladisp* tool in a more aesthetic format. *Qladisp* and *sawRobotIO1394QtConsole* were both used extensively throughout the troubleshooting process.

PID Tuning

Each type of manipulator (MTM, PSM, ECM) has an associated PID configuration file which was downloaded from the jhu-dvrk Github repository. However, because the PID parameters in this file were specifically for the system at JHU, some tuning was necessary. The program *sawIntuitiveResearchKitQtPID* facilitated this, and it was used with the syntax

```
$    sawIntuitiveResearchKitQtPID -i <path to I/O configuration file> -p <path to PID configuration file> -a
     <arm name>
```

Because *sawIntuitiveResearchKitQtPID* does not offer I/O access, *sawRobotIO1394QtConsole* opens alongside it. The PID parameters for individual joints can be tuned within the *sawIntuitiveResearchKitQtPID* GUI. The GUI also provides an option for setting the current manipulator position as a target position. So, a target position for the manipulator was set and the manipulator was moved to various different positions. The option for the manipulator to attempt to reach the target position as quickly and stably as possible was then selected. This process was repeated several times with slightly altered PID parameter values until an optimal set with little overshoot was found.

ROS Control

With the FPGA1394-QLA controller boards and the *cisst-saw* package working correctly and the manipulators properly calibrated, the da Vinci Surgical System could now be controlled through ROS as facilitated by the *dvrk-ros* package. However, whereas the *cisst-saw* programs required an XML configuration file, *dvrk-ros* required a JavaScript Object Notation (JSON) file, which had to be written. The JSON file specified the file paths for the I/O and PID configuration files, as well as other parameters needed by the *dvrk-ros* software, such as the period of the ROS control loop.

Because *dvrk-ros* is based on ROS, *roscore* had to be running in the background for its programs to run. This was accomplished by simply opening another terminal, typing *roscore*, and pressing enter. *Roscore* essentially allows ROS nodes to communicate. After starting *roscore*, the program *dvrk_console_json* was started with the syntax

```
$    rosrun dvrk_robot dvrk_console_json -j <path to JSON file>
```

Dvrk_console_json started a GUI which contains all the functionality of the *cisst-saw* programs as well as some features specific to *dvrk-ros*. *Dvrk_console_json* also creates many ROS topics that can be used to control the manipulator or get feedback about its state by publishing or subscribing to them, respectively.

Upon opening *dvrk_console_json*, the software state was DVRK_UNINITIALIZED. Homing a manipulator was the only possible action in this state, so homing had to be performed before the manipulator could be controlled through the ROS interface. After homing, the software state was DVRK_READY. From this state, a number of other states could be set which made the manipulator controllable. To change the state, a new terminal was opened and a message specifying the desired state was published to the *set_robot_state* topic. The syntax for this was

```
$    rostopic pub /dvrk/<manipulator name>/set_robot_state std_msgs/String “data: ‘<desired state>’”
```

One topic that was used for testing was the DVRK_POSITION_JOINT state. When in this state, the manipulator was controlled by a dVRK node subscribed to the *set_position_joint* topic. So, by publishing to this topic, the manipulator could be controlled.

CONTROLLING THE ENDOSCOPE WITH OCULUS RIFT

Proof-of-Concept by Simulation

Throughout the process of building the dVRK controller boards and integrating them with the full da Vinci Surgical System, there was downtime while waiting for parts to arrive. It was during this downtime that a simulation was developed in which real-world motion was used to control a virtual da Vinci Surgical System. Lacking at the time an Oculus Rift headset with which to control the virtual da Vinci Surgical System, an Inertial Measurement Unit (IMU) had to be made.

An IMU is a device which provides the orientation of an object in space. The Oculus Rift contains a highly optimized IMU which utilizes an accelerometer, magnetometer, and gyroscope for its orientation measurement. However, for the sake of time and because it was only being used for a simulation, the simple IMU utilized only an accelerometer and gyroscope and was far less optimized than that of the Oculus Rift headset. To make the IMU, an MPU9255 – an IC containing both an accelerometer and a gyroscope – was interfaced to an Arduino Nano on which ran a program to read measurements from the sensors and combine them into a stable, accurate measurement of the MPU9255 orientation. As there was not yet a reliable, open-source library for reading the MPU9255, code had to be written to accomplish this. The code was written in the Arduino language, which is simply a set of C and C++ functions.

The Arduino communicated with the MPU9255 via Inter Integrated Circuit (I^2C): a serial, half-duplex communication protocol. The Arduino library *Wire.h* was used for the code as it simplifies I^2C communication by providing functions for issuing requests to and reading the responses from a slave device. The MPU9255 was the slave device in this case, while the Arduino played the role of master. The accelerometer and gyroscope measurements were stored in the registers of the MPU9255 as the high and low bytes of 16-bit signed integers at the following addresses:

3B:	ACCEL_XOUT_H
3C:	ACCEL_XOUT_L
3D:	ACCEL_YOUT_H
3E:	ACCEL_YOUT_L
3F:	ACCEL_ZOUT_H
40:	ACCEL_ZOUT_L
43:	GYRO_XOUT_H
44:	GYRO_XOUT_L
45:	GYRO_YOUT_H

46:	GYRO_YOUT_L
47:	GYRO_ZOUT_H
48:	GYRO_ZOUT_L

Due to limitations of I^{2}C, the bytes had to be read one at a time and the high and low bytes were combined into a single value after they were acquired. This was done by bit-shifting the high byte eight bits to the left and adding it to the low byte.

Some orientation information can be acquired using only the accelerometer or the gyroscope alone, but a more reliable measurement is given by using both sensors simultaneously. The simplest sensor fusion method is known as a complementary filter, which was implemented for the simulation. There are better methods, such as Kalman filtering, but they are more complex and unnecessary for a proof-of-concept. The Oculus Rift headset, for example, uses well-optimized Savitzky-Golay filtering to achieve a smooth and accurate measurement of its orientation.[8] A complementary filter combines the accelerometer and gyroscope data in a way that cancels their respective weakness while taking advantage of their strengths. The accelerometer on the MPU9255 outputs the acceleration relative to free-fall in each of the three Cartesian directions. It is therefore straightforward to calculate from the accelerations the tilt of the accelerometer relative to the direction of the force of gravity. So, accelerometers provide a direct measurement of the angular position of an object. However, translational acceleration of the accelerometer will affect the angular position calculation. Fortunately, a gyroscope is completely unaffected by translational motion and outputs only the rate of change in its yaw, pitch, and roll. The problem with using a gyroscope to find the angular position is that its output must be numerically integrated to do so, which introduces drift. Because the sampling time is by necessity discrete, small errors in the integration will accumulate over time. The complementary filter utilizes both sensors in the following way

$$\theta_{new} = \alpha(\theta_{old} + gyro_{\theta}dt) + (1 - \alpha)acc_{\theta}$$

where θ_{old} and θ_{new} are the previous and current calculations of the angle, respectively, $gyro_\theta$ is the angular rate as outputted by the gyroscope, acc_θ is the angle as calculated from the accelerometer outputs, dt is time between measurements, and α is a parameter that can be tuned to improve the stability of the output. The complementary filter, if correctly tuned, combines the short-term stability of the gyroscope with the long-term stability of the accelerometer to provide a calculation of the angle that is accurate and fairly stable over time.

The program to read the raw sensor data and convert it into a useful orientation measurement was implemented on an Arduino Nano, which sent the orientation information to the control computer via the serial port at 115200 baud. A node, *imu_serial_data_reader*, was written in Python to read this information from the serial port and publish it in the appropriate form as a JointState message to the *joint_states_robot* topic. This topic was used in conjunction with a Unified Robot Description File (URDF) by the *robot_state_publisher* package to manipulate a virtual da Vinci Surgical System in *rviz*, a tool for ROS visualization. The virtual da Vinci Surgical System was created in SolidWorks, and this CAD model was exported with the *sw_urdf_exporter* plugin to obtain the URDF. Finally, a launch file was written to simultaneously start the *imu_serial_data_reader*, *robot_state_publisher*, and *rviz.* The orientation of the simple IMU could then be used to control the outer pitch and yaw of a virtual manipulator. In this way, a proof-of-concept was achieved by means of a simulation.

Accessing the Oculus Rift Inertial Measurement Unit

While the complementary filtered IMU was sufficient for a proof-of-concept, a more complex IMU was necessary for smooth translation of sensor orientation to the joints of the da Vinci Surgical System. The Oculus Rift headset has such an IMU, so an Oculus Rift DK2 was acquired for the project. A number of open source programs were investigated as a means for reading the IMU data.

Before proceeding with these investigations, however, a graphics card capable of interfacing with the Oculus Rift had to be acquired. Though it was not necessary to do any rendering to the Oculus display, there was a minimum requirement on the graphics card for the headset to even be detected by the computer. The Department of Computer Science at the University of Innsbruck had a high-end computer with a graphics card that exceeded the minimum requirement, and it was lent to MCI for this project. Now, with two computers, there were two options: 1) Run nodes across both computers – one for reading the Oculus Rift and one for controlling the da Vinci Surgical System, or 2) run all nodes on the Oculus-capable computer. Because it was of interest to verify the repeatability of the software setup, the second option was chosen. Therefore, Ubuntu was installed on the Oculus-capable computer along with ROS, *cisst-saw*, and *dvrk-ros*.

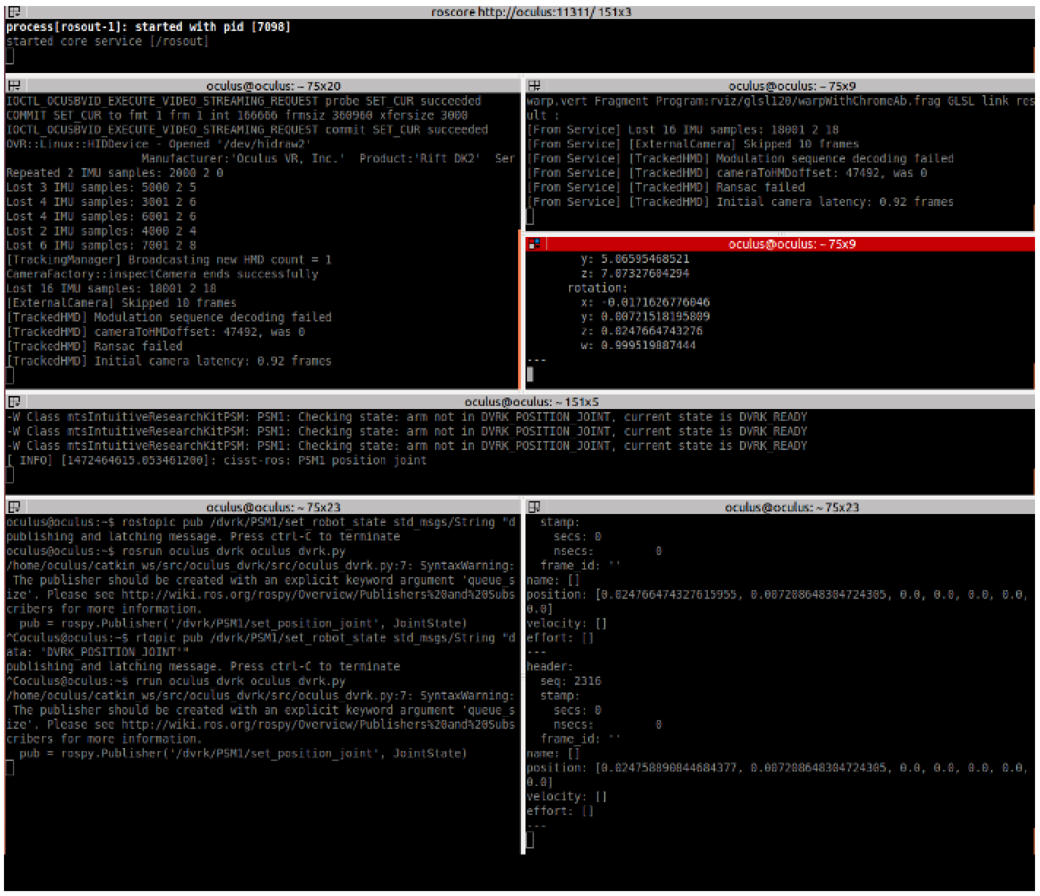
It was determined that the open source ROS packages *ros_ovr_sdk* and *oculus_rviz_plugins* from Oregon State University would work best for the project. These packages were downloaded from the OSUrobotics GitHub repository and built with catkin.

```
$    cd ~/catkin_ws/src
$    git clone https://github.com/OSUrobotics/ros_ovr_sdk.git
$    git clone https://github.com/OSUrobotics/oculus_rviz_plugins.git
$    catkin build
```

The *ros_ovr_sdk* package contains a node, *ovrd*, which allows access to the internal sensors of the Oculus Rift headset. Running this node, followed by opening *rviz* and adding the *OculusDisplay* plugin, granted access via ROS to the Oculus Rift headset orientation information.

Manipulator Control with Oculus Rift

To control a da Vinci Surgical System manipulator with the Oculus Rift headset, the Python node used for the proof-of-concept simulation was modified. The node was renamed to *oculus_dvrk* for this portion of the project. The principle was the same: subscribe to one topic, convert to the appropriate message type, and publish to another topic. In this case, the topic subscribed to was */tf*, which is where the *OculusDisplay* plugin in *rviz* published the Oculus Rift IMU data. The topic published to was *set_position_joint*, and the dVRK state was changed to DVRK_POSITION_JOINT prior to running the Python node. The message type of the Oculus Rift IMU data was TFMessage, which was converted into a JointState message for the *set_position_joint* topic.



```
process[rosout-1]: started with pid [7098]
started core service [/rosout]
```

```
IOCTL_OCUSBVID_EXECUTE_VIDEO_STREAMING_REQUEST probe SET_CUR succeeded
COMMIT SET_CUR to fmt 1 frm 1 int 166666 frmsiz 380960 xfersize 3000
IOCTL_OCUSBVID_EXECUTE_VIDEO_STREAMING_REQUEST commit SET_CUR succeeded
OVR::Linux::HIDDevice - Opened '/dev/hidraw2'
                    Manufacturer:'Oculus VR, Inc.'  Product:'Rift DK2'  Ser
Repeated 2 IMU samples: 2000 2 0
Lost 3 IMU samples: 5000 2 5
Lost 4 IMU samples: 3001 2 6
Lost 4 IMU samples: 6001 2 6
Lost 2 IMU samples: 4000 2 4
Lost 6 IMU samples: 7001 2 8
[TrackingManager] Broadcasting new HMD count = 1
CameraFactory::inspectCamera ends successfully
Lost 16 IMU samples: 18001 2 18
[ExternalCamera] Skipped 10 frames
[TrackedHMD] Modulation sequence decoding failed
[TrackedHMD] cameraToHMDoffset: 47492, was 0
[TrackedHMD] Ransac failed
[TrackedHMD] Initial camera latency: 0.92 frames
```

```
warp.vert Fragment Program:rviz/glsl120/warpWithChromeAb.frag GLSL link res
ult :
[From Service] Lost 16 IMU samples: 18001 2 18
[From Service] [ExternalCamera] Skipped 10 frames
[From Service] [TrackedHMD] Modulation sequence decoding failed
[From Service] [TrackedHMD] cameraToHMDoffset: 47492, was 0
[From Service] [TrackedHMD] Ransac failed
[From Service] [TrackedHMD] Initial camera latency: 0.92 frames
```

```
        y: 5.06595468521
        z: 7.07327684294
      rotation:
        x: -0.0171626776046
        y: 0.00721518195889
        z: 0.0247664743276
        w: 0.999519887444
---
```

```
-W Class mtsIntuitiveResearchKitPSM: PSM1: Checking state: arm not in DVRK_POSITION_JOINT, current state is DVRK_READY
-W Class mtsIntuitiveResearchKitPSM: PSM1: Checking state: arm not in DVRK_POSITION_JOINT, current state is DVRK_READY
-W Class mtsIntuitiveResearchKitPSM: PSM1: Checking state: arm not in DVRK_POSITION_JOINT, current state is DVRK_READY
[ INFO] [1472464615.053461200]: cisst-ros: PSM1 position joint
```

```
oculus@oculus:~$ rostopic pub /dvrk/PSM1/set_robot_state std_msgs/String "d
publishing and latching message. Press ctrl-C to terminate
oculus@oculus:~$ rosrun oculus_dvrk oculus_dvrk.py
/home/oculus/catkin_ws/src/oculus_dvrk/src/oculus_dvrk.py:7: SyntaxWarning:
 The publisher should be created with an explicit keyword argument 'queue_s
ize'. Please see http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subs
cribers for more information.
  pub = rospy.Publisher('/dvrk/PSM1/set_position_joint', JointState)
^Coculus@oculus:~$ rtopic pub /dvrk/PSM1/set_robot_state std_msgs/String "d
ata: 'DVRK_POSITION_JOINT'"
publishing and latching message. Press ctrl-C to terminate
^Coculus@oculus:~$ rrun oculus_dvrk oculus_dvrk.py
/home/oculus/catkin_ws/src/oculus_dvrk/src/oculus_dvrk.py:7: SyntaxWarning:
 The publisher should be created with an explicit keyword argument 'queue_s
ize'. Please see http://wiki.ros.org/rospy/Overview/Publishers%20and%20Subs
cribers for more information.
  pub = rospy.Publisher('/dvrk/PSM1/set_position_joint', JointState)
```

```
  stamp:
    secs: 0
    nsecs:         0
  frame_id: ''
name: []
position: [0.024766474327615955, 0.007288648384724305, 0.0, 0.0, 0.0, 0.0,
0.0]
velocity: []
effort: []
---
header:
  seq: 2316
  stamp:
    secs: 0
    nsecs:         0
  frame_id: ''
name: []
position: [0.024758890844684377, 0.007288648384724305, 0.0, 0.0, 0.0, 0.0,
0.0]
velocity: []
effort: []
---
```

Figure 6. Five separate terminals were necessary for controlling a manipulator with Oculus Rift. The other two terminals in this screenshot were for echoing the */tf* and *set_position_joint* states.

The steps to achieve control of a manipulator with the Oculus Rift were:

1. Start *roscore*.
2. In a new terminal, run *dvrk_console_json* with the appropriate JSON file.
3. Home the manipulator.
4. Set the state to DVRK_POSITION_JOINT.
5. In a new terminal, run *ovrd*.
6. In a new terminal, run *rviz* and add the *OculusDisplay* plugin.
7. In a new terminal, run *oculus_dvrk*.

Future Improvements

Though the project was a success, the current system is a prototype at best and there is much room for improvement. Three possible improvements will be briefly discussed in this final section. ASU and MCI will continue to collaborate on this project, and the proposed improvements will be implemented in the future.

The first and most obvious improvement would be to control the ECM itself with the Oculus Rift headset. In the interest of time and for the sake of achieving motion control of a manipulator, a PSM was controlled instead. The mechanics and everything about the system for the ECM would be the same, except a larger power supply would be necessary and there would be an extra step in the current calibration. The ECM brakes current would have to be calibrated, which is done by running *sawRobotIO1394CurrentCalibration* with the *-b* option. So, for the ECM, the current calibration would have to be performed twice: once for the joints and once for the brakes. The ECM I/O configuration file is special as it contains information about the brakes in addition to the normal information.

The second improvement would be to stream the endoscope camera video output to the Oculus Rift headset display. As both the da Vinci Surgical System master console and the Oculus Rift headset use a stereoscopic video display, streaming to the Oculus Rift would not be unlike how the system was intended to work. The *cisstStereoVision* library would likely be of use for this feature.

The final proposed improvement would be to write a launch file to automate the initialization process for controlling the ECM with the Oculus Rift. In its present state, the system requires five separate terminals to be opened and there are seven steps that have to be performed in exactly the right order. However, if a launch file was written, as with the proof-of-concept simulation, the entire process would be condensed to a single line.

```
$     roslaunch oculus_dvrk oculus_dvrk.launch
```

ACKNOWLEDGEMENTS

Thank you to Dr. Yeongmi Kim of MCI for supervising me on this project and for helping me define goals, both short-term and long-term, which led to the successful completion of this project. Thank you to Gerald Streng and Stephan Jäkel, also of MCI, for assembling the controller boards and helping me with troubleshooting. Thank you to Dr. Andreas Mehrle and the entire Department of Mechatronics at MCI for hosting me this summer and for a warm and bittersweet sendoff. Thank you to Ricardo Téllez, Alberto Ezquerro, and Miguel Angel Rodíguez of The Construct in Barcelona for teaching me about ROS. Thank you to Dr. Judson Clements of Appalachian State University for giving me this opportunity and teaching me the skills that made this project successful. Finally, thank you to the Marshall Plan Foundation for providing the funding that allowed me to spend the summer abroad to complete this project.

REFERENCES

1. Feder, B. J. (2008, May 4). *Prepping Robots to Perform Surgery*. Retrieved from http://www.nytimes.com/2008/05/04/business/04moll.html?_r=0
2. Intuitive Surgical, Inc. (2012). *Regulatory Clearance*. Retrieved from https://web.archive.org/web/20130116100318/http://intuitivesurgical.com/specialties/regulatory-clearance.html
3. Singer, E. (2010, March 24). *The Slow Rise of the Robot Surgeon*. Retrieved from https://www.technologyreview.com/s/418141/the-slow-rise-of-the-robot-surgeon/
4. Intuitive Surgical, Inc. (2015). *Investor FAQ*. Retrieved from http://phx.corporate-ir.net/phoenix.zhtml?c=122359&p=irol-faq
5. Kazanzides, P., Chen, Z., Duguet, A., Fischer, G. S., Taylor, R. S., & DiMaio, S. P. (2014). *An Open-Source Research Kit for the da Vinci Surgical System.* IEEE International Conference on Robotics & Automation, Hong Kong, China, 31 May-7 June 2014 (pp. 6434-6439).
6. Johns Hopkins University GitHub Wiki. (2016). *XMLConfig.* https://github.com/jhu-dvrk/sawIntuitiveResearchKit/wiki/XMLConfig
7. Johns Hopkins University GitHub Wiki. (2014). *cisst libraries and SAW Components.* https://github.com/jhu-cisst/cisst/wiki/cisst-libraries-and-SAW-components
8. Lavelle, S. (2013, July 12). *The Latent Power of Prediction*. Retrieved from https://developer3.oculus.com/blog/the-latent-power-of-prediction/

CODE APPENDIX

Simple IMU Arduino Code (.ino)

```
#include "Wire.h"

#define MPU9255_ADDRESS   0x68
#define WHO_AM_I          0x75
#define ACCEL_XOUT        0x3B
#define ACCEL_YOUT        0x3D
#define ACCEL_ZOUT        0x3F
#define GYRO_XOUT         0x43
#define GYRO_YOUT         0x45
#define GYRO_ZOUT         0x47

const double pi = 3.14159265358979;

int i, j;
int16_t a[3];
int16_t g[3];
int16_t m[3];
byte device_id;
byte error;
byte acc_address[] = {ACCEL_XOUT, ACCEL_YOUT, ACCEL_ZOUT};
byte gyro_address[] = {GYRO_XOUT, GYRO_YOUT, GYRO_ZOUT};
double roll, pitch, rollRate, pitchRate, rollAngle, pitchAngle, dt;

void setup() {
  Wire.begin();
  Serial.begin(115200);

  Serial.println("Initializing I2C communication with MPU9255...");
  //Check that the device ID stored in the WHO_AM_I register matches the
known ID of 0x73 for the MPU9255
  //This is to ensure that the device is properly connected
  device_id = read(WHO_AM_I, 1);
  Serial.println(device_id == 0x73 ? "Communication with MPU9255 successful"
: "MPU9255 not found");

  while(!(device_id == 0x73)); //Don't continue to loop() if ID check fails

  dt = micros();
  rollAngle = 0;
  pitchAngle = 0;
}

void loop() {
  for(i = 0;i < 3;i++){
    a[i] = read(acc_address[i], 2);
    g[i] = read(gyro_address[i], 2);
  }

  //Convert accelerometer data to degrees and gyroscope data to
degrees/second
  pitch = atan2(-a[1], a[2]) * 180.0 / pi;
  roll = atan2(-a[0], a[2]) * 180.0 / pi;
  pitchRate = g[0] * 500.0 / 32768.0;
  rollRate = g[1] * 500.0 / 32768.0;

  //Complementary filter
  rollAngle = 0.98 * (rollAngle + rollRate * (micros() - dt) / 1000000.0) +
0.02 * roll;
  pitchAngle = 0.98 * (pitchAngle + pitchRate * (micros() - dt) / 1000000.0)
+ 0.02 * pitch;
  dt = micros();
  Serial.print(pitchAngle); Serial.print(" "); Serial.println(rollAngle);
}

int16_t read(byte valueAddress, int bytesToRead){
  //Specify the register address from which to read
  Wire.beginTransmission(MPU9255_ADDRESS);
  Wire.write(valueAddress);
  error = Wire.endTransmission();

  //Read value from specified register
  //When reading the device ID, only one read needs to be performed as the ID
is one byte
  //The acc and gyro values are two bytes with the high and low bytes
alternating e.g. ACC_XOUT_H, ACC_XOUT_L, ACC_YOUT_H, etc
  //Since the high byte is read first, the value is bit shifted right 8 times
before being added to the low byte
  Wire.beginTransmission(MPU9255_ADDRESS);
  Wire.requestFrom(MPU9255_ADDRESS, bytesToRead);
  if(bytesToRead == 1){
    return Wire.read();
  }
  else{
    return (Wire.read() << 8) + Wire.read();
  }
  error = Wire.endTransmission();
}
```

imu_serial_data_reader.py

```
#!/usr/bin/env python

import roslib
import rospy
import sys
import serial
from math import pi
from sensor_msgs.msg import JointState

jnt_msg = JointState()

arduino = serial.Serial('/dev/ttyUSB0',115200)
```

```
def control_simulation():
        jnt_pub = rospy.Publisher('/joint_states_robot', JointState,
queue_size = 10)
        rospy.init_node('imu_serial_data_reader', anonymous = True)

        roll = 0
        pitch = 0
        deg_to_rad = pi / 180

        jnt_msg.position = [0,0,0,0,0,0,0,0,0,0,0,0,0]
         jnt_msg.name = ['linear_box_left_joint',
                         'PJ_1_left_joint',
                         'PJ_2_left_joint',
                         'PJ_3_left_joint',
                         'PJ_4_left_joint',
                         'PJ_5_left_joint',

                         'AJ_1_left_joint',
                         'AJ_2_left_joint',

                         'left_linear_tool_joint',
                         'left_outer_roll_joint',
                         'left_outer_wrist_pitch_joint',
                         'left_outer_wrist_yaw_joint',
                         'left_outer_wrist_open_angle_joint']

        while not rospy.is_shutdown():
                angles = arduino.readline().split()

                try:
                        float(angles[0])
                        float(angles[1])

                        roll = float(angles[0])
                        pitch = float(angles[1])
                except ValueError:
                        pass
                except IndexError:
                        pass

                jnt_msg.position[0] = roll * deg_to_rad
                jnt_msg.position[1] = pitch * deg_to_rad

                jnt_msg.header.stamp = rospy.Time.now()
                jnt_pub.publish(jnt_msg)


if __name__ == '__main__':
        control_simulation()
```

Simulation Launch File (.launch)

```
<?xml version="1.0"?>


<launch>
        <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
ingramds_davinci)/launch/urdf.rviz" required="true" />

  <group ns="ingramds_davinci">
    <param name="robot_description" textfile="$(find
ingramds_davinci)/robots/MCI_davinci.URDF" />
    <param name="rate" value="100"/>
    <rosparam param="source_list" subst_value="True">
      [joint_states_robot]
    </rosparam>
    <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
    <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
  </group>

      <node name="joint_publisher"
            pkg="ingramds_davinci"
            type="imu_serial_data_reader.py"
        output="screen"/>

</launch>
```

oculus_dvrk.py

```
#!/usr/bin/env python

import roslib
import rospy
import sys
import serial
from math import pi
from sensor_msgs.msg import JointState
from tf2_msgs.msg import TFMessage

jnt_msg = JointState()

def callback(data):
        global roll, pitch
        roll = data.transforms[0].transform.rotation.x
        pitch = data.transforms[0].transform.rotation.z

def control_daVinci():
        jnt_pub = rospy.Publisher('/joint_states_robot', JointState, 
queue_size = 10)
        imu_sub = rospy.Subscriber('/tf', TFMessage, callback)
        rospy.init_node('oculus_dvrk', anonymous = True)

        roll = 0
        pitch = 0
        deg_to_rad = pi / 180)

        jnt_msg.position = [0,0,0,0,0,0,0]

        while not rospy.is_shutdown():
                jnt_msg.position[0] = roll * deg_to_rad
                jnt_msg.position[1] = pitch * deg_to_rad

                jnt_msg.header.stamp = rospy.Time.now()
                jnt_pub.publish(jnt_msg)


if __name__ == '__main__':
        control_daVinci()
```

PSM1 JSON File

```
/* -*- Mode: Javascript; tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 4 -*- */
{
    "io": {
        "period": 0.0005, // in seconds
        "port": 0 // default is 0
    }
    ,
    "arms":
    [
        {
            "name": "PSM1",
            "type": "PSM",
            "io": "sawRobotIO1394-PSM1-30084.xml",
            "pid": "sawControllersPID-PSM.xml",
           "kinematic":"psm-large-needle-driver.json"
        }
    ]
}
```