Robotic Arm

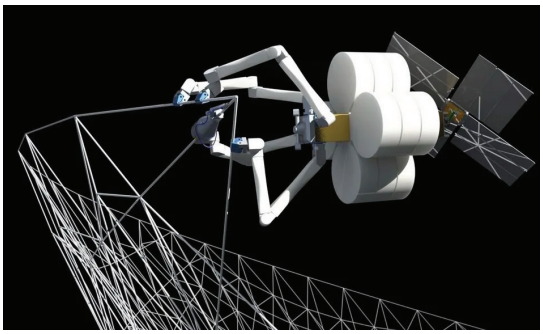
SPRING 2022

Weiping Huang, Jacob Kuczynski, Henry Pan, Landon Knipp, & Jason Guardado

Abstract

Our design is a robotic limb with a gripper attached at the end that mimics the motion of the user. The position of the user's arm and hand is sampled with sensors to obtain data in order to depict the positioning of four servos involved with the actuation. As the data is collected, it is sent via the serial port on the microcontroller to an external computer running LabVIEW that depicts the data on a custom GUI. The motivation for the microprocessor based mechatronics system was founded by a new frontier of technology in our society. Medicine has evolved to the point where surgeries have begun to be performed using robotics for increased position. The future of the human race and our existence in space is continuing to advance, and this technology could potentially be used for constructing new structures in space.

(a) Robotic Instrument Being Used to Perform a High-Precision Surgery



(b) Robot Building Onto a Space Structure Being Remotely Controlled

Figure 1: Key Motivations Behind the Design of Our Robotic Arm

1 Conceptualization

The main design of our system is to have the microcontroller be the central aspect of taking in and computing information. Breaking it down into simple parts: there are the inputs, computations, and outputs. The software was developed using Modus Toolbox™ by Infineon Technologies, written in C.

1.1 Inputs

Outside of calibration (will be further discussed later), the only inputs to the system is the information received from the IMUs and the flex sensor. The IMU's purpose was to measure the orientation of the user's arm, using an accelerometer and gyroscope). Two IMU's were used: one on the back of the hand and another on the upper part of the forearm. The IMU's relied on I2C to communicate and send information to the microcontroller. A flex sensor is placed in the hand, which detects any gripping motion.

1.2 Computations

Before the CPU can actuate the servos, the data from the IMU's and the flex sensor need to be converted to values that position the servos accordingly. A pinned on the microcontroller is set to create an output

as a pulse-width-modulated (PWM) wave. The duty cycles that work for the motors range from $2.5 - 12.5$. 2.5 outputs the position of $0°$ and 12.5 results in 180 (the base motor for the structure is different from the other motors because it has an angular range of $0 - 270°$). The motors have PID (Proportional-Integral-Derivative) controllers built into them so we did not have to directly incorporate positioning controls into our software. We linearly convert the values received from the IMU's to a duty cycle of the appropriate value in order to actuate the position we want for each servo at each instant.

1.3 Outputs

The data about the positions of each servo is sent by printing a string to the serial port of the microcontroller. This is how information is sent to be shown on the GUI (more information about the setup is discussed in Software Design & Implementation). This is one of the outputs of the PSOC 6. Another output is directly done with the GPIO pins, which creates the PWM waves for the servos. A unique GPIO pin is designated to a servo, each sending out a unique PWM with a precise duty cycle.

2 CAD Models, Mechanics, & Materials

The initial design for the robotic arm was to use hollow cylindrical shaped rods to make up the actual body of the design. We then ran into complications on how we would develop a structure to support the servos, and further create joints that functions properly. We then decided that instead of machining aluminum rods ourselves and having to incorporate implementation of the servos, we decided to order aluminum frames that were compatible with the servos we were using. This is because we wanted to limit material waste and minimize the amount of money being spent (none of us had makers passes to the machine shop, which tend to get relatively expensive, especially for only one of us to have access to it). Using a frame also limited the weight, and hence the angular moment of inertia of the body, resulting in less total energy of use, and we could rely on motors with less torque capabilities. We could have build transmissions in order to increase the torque, however that would also increase the complexity and weight of the design, and it was somewhat important to reduce the amount of total energy used.

Motor Actuation:

All of the servos are shown in Figure 2. Servo 1 operates the gripper. For the lowest duty cycle the claw remains open. Higher duty cycle values close the claw, up to when both extensions that make up the claw touch each other. Therefore, it should be able to care any sized object that can fit in the space of the gripper if it is light enough. The object needs to be of a certain weight based on the max torque Servo 1 an provide to create enough friction to hold the object, and also the weight of the object being held needs to be supported by the other servos as well, mainly Servos 2 and 3. Servo 2 operates somewhat as a wrist joint for the system. It does not rotate around the axis that mirrors the two sides of the claw, but rotates it more up and

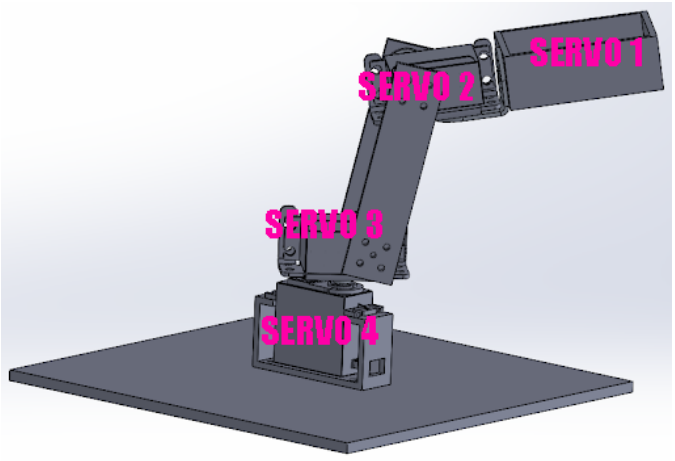


Figure 2: CAD of Robotic Arm Design

down. Servo 3 operates as the elbow joint. This is the main joint that move the gripper closer to the ground. We specifically chose a motor that possesses higher torque capabilities specifically for this motor since it has to support the weight of most of the arm. Servo 4 is the only motor that has a range from $0 - 270°$. We wanted a larger range of motion for this motor since we wanted a larger angular area to be able to be covered.

Materials/Products	Vendor
LSM6DSO: always-on 3D accelerometer and 3D gyroscope	STMicroelectronics
Short Flex Sensor	Adafruit
Rolled Steel Plate	Bonanza
ZOSKAY 35kg high Torque Coreless Motor Servo	Zemei Iu
Servo Mount Aluminum Brackets	BaBlock
Seamuing Micro Servo Motors	Deegoo-FPV
Cypress, PSOC 6, CY8CPROTO-063-BLE	RS Components Ltd

3 Software Design & Implementation

We had two separate lines of software that function together: LabVIEW and compiled C code. It was important to incorporate both lines of the software because we wanted there to be some visual output that made sense to the user, and incorporate an aspect of multitasking to our design.

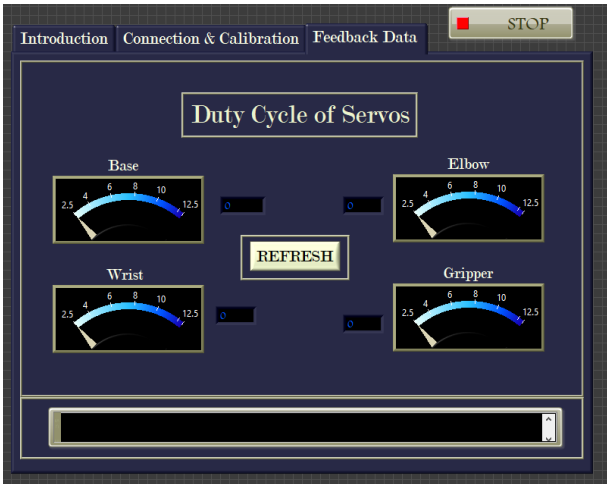
LabVIEW runs directly on an external computer, such as a laptop. The C code was developed and compiled on Modus Toolbox IDE. It was usefull to use the Modus Toolbox because there we were able to build environments that work seamlessly with the specific microcontroller in use, the PSOC 6 BLE 63. The toolbox made it simple to initialize peripherals, set up timers and interrupts, and assign GPIO pins for specific uses.

3.1 LabVIEW

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) is an application that uses data flow language to operate. This is different from sequential flow, which is seen in languages such as C, C++, Python, Java, MATLAB, etc. This changes the perspective of the code because it relies on what information is ready to be manipulated, rather than in the order of the lines it is written. It should be noted that the real time aspect of the project was accomplished on the side of the compiled C code, and not on the LabVIEW portion. This is because LabVIEW runs on Windows, and Windows inherently is not a system that operates in "real time". The main purpose of LabVIEW in our project was so to incorporate a GUI (Graphical User Interface).

(a) Introduction Tab on LabVIEW GUI

(b) Connection & Calibration Tab on LabVIEW GUI



(c) Feedback Data Tab on LabVIEW GUI

Figure 3: LabVIEW GUI

Figure 3 (a) shows the first tab of the GUI. It is a simple introduction that talks about the design of our system, just so there is a bit of background knowledge for the user.
The second tab, seen in Figure 3 (b) is where the user establishes a connection with the microcontroller. Assuming the user is running LabVIEW on Windows, you can determine which COM port is the correct number by searching on the device Settings ¿ Device Manager ¿ Ports. If the microcontroller is plugged into the computer running LabVIEW, and the correct COM port is chosen, when the VI (Virtual Instrument) is ran, the LED on the GUI should change from 'DISCONNECTED' and light up, displaying 'CONNECTED'. This tab is also where the calibration of the peripherals takes place. The calibration is necessary because it initializes the range and boundary conditions for the values of the IMU's and the flex sensor. The 'Claw Fully Closed' and 'Claw Fully Open' buttons calibrate the motion of the flex sensor, which outputs different resistances for how much it is bent/flexed. The 'Shoulder Right' and 'Shoulder Left' buttons calibrate the positions of the IMU's.
The third tab, Figure 3 (c) is where the user can see the updated position of all of the motors. The text

boxe at the bottom of the of the tab shows messages printed to the serial line of the PSOC confirming messaging are being received throughout the run-time.

State Transition Diagram Used For Communication and Calibration

Figure 4 shows the state transition diagram for the communication system on LabVIEW. It sends and receives information via the serial port of the PSOC. The system is initialized when the LabVIEW VI begins running, and Connect is the connection process to the microcontroller. Idle is where it awaits for further instruction. The state of the system will then transition as soon as it is told to send information or when it detects data being received. Information is sent from this line of software during the calibration of the peripherals, via the buttons in Figure 3 (b). The principal functionality after the calibration of the robotic arm is to receive data about the positions of the servos. If an error occurs at any step throughout the process, the program will automatically stop running. This is the heart of the LabVIEW code. The other portion of the LabVIEW software was assigning the incoming information to different indicators, such as the different servos. Figure 5 shows information being received through the serial line and breaking up the information to be shown by different indicators.

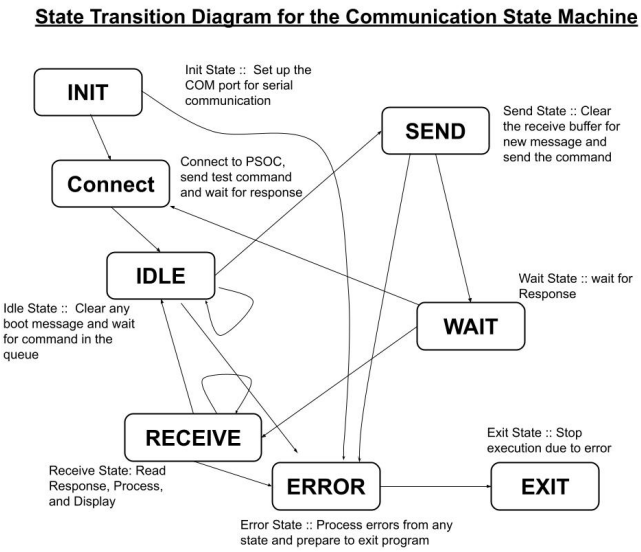


Figure 4: State Transition Diagram for Communication State Machine On LabVIEW

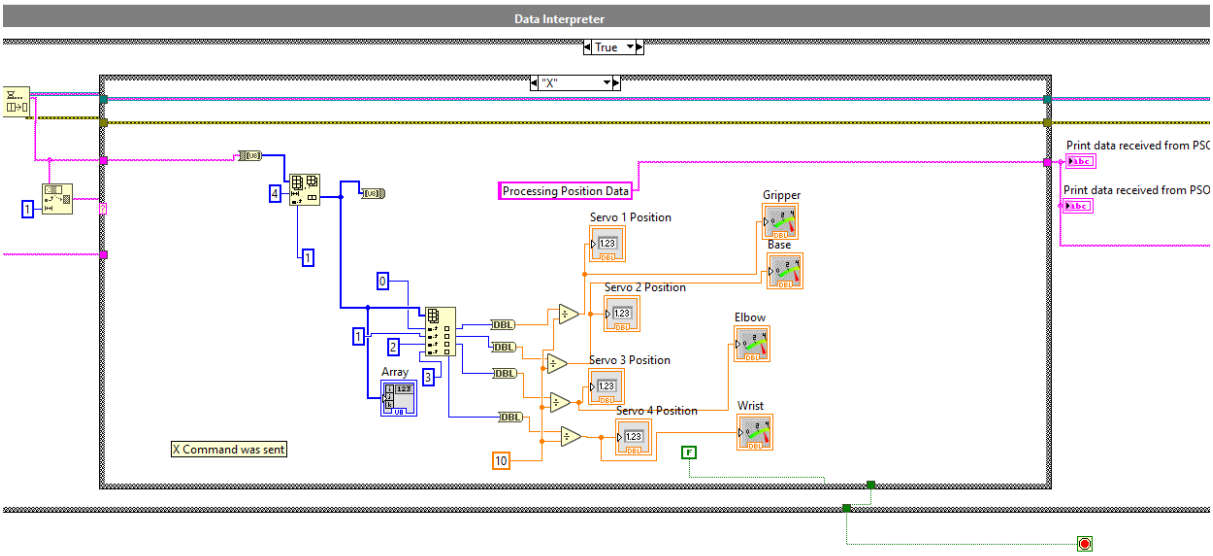


Figure 5: LabVIEW VI Block Diagram For Received Information From PSOC

3.2 Compiled C Code Using Modus Toolbox™

The code compiled onto the PSOC is the second line of software implemented into our embedded system. It is where all of the data is received and sent to all of the peripherals. It is also the side of the software that incorporates both multitasking and the real time aspect of the project. Multitasking is accomplished by how the PSOC controls the servos, IMU's, and flex sensor, while also communicating with the LabVIEW VI. We specifically made the decision to invest in purchasing PSOC's as our choice of microcontroller over utilizing the ESP32 given in the kits because we valued the superior multitasking capabilities of the PSOC.

The base of the code is setting up the software environment, initializing variables, and setting up timers and interrupts. A timer interrupt is set up, not to use for any GPIO pins, but signals to other interrupts, such as those for the motors. An interrupt is created for each motor. These interrupts are triggered by the timer interrupt. At this instant, the PSOC draws information it receives from the IMU's and flex sensor. It then converts this data into appropriate duty cycles for correct positioning of the servos. Data is also printed as a string at the serial port. It is this string that is then detected on the LabVIEW side, which is further discussed in Figure 4.

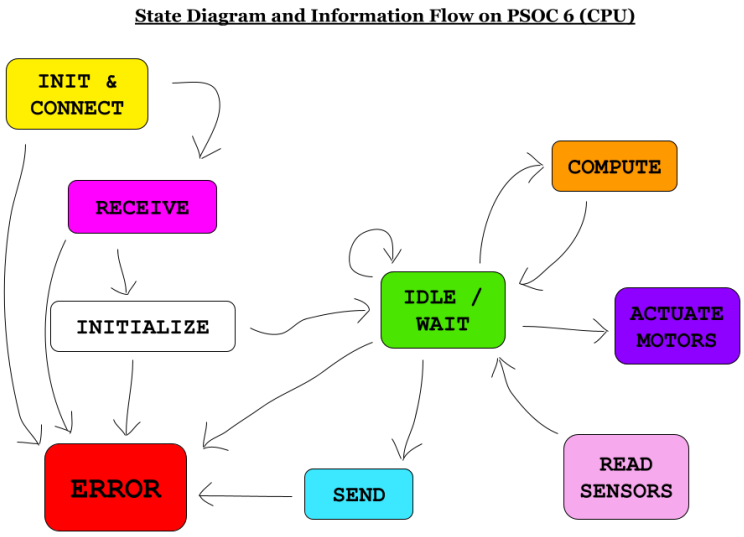


Figure 6: State Transition Diagram & Information Flow On PSOC

State Transition Diagram & Information Flow On PSOC

Figure 6 visualizes the states of the software on the PSOC and how tasks are accomplished. The initialization and connection occurs when the PSOC establishes a connection with the external computer that runs LabVIEW. Receive is when it awaits to receive the information for calibration, which is then it initializes the conditions. After this, it runs continuously: receiving data from the inputs, computing the information, and sending out data for the outputs. Again, you can see that if there is an error at any instant, the program will automatically detect the error and stop running. It is important to highlight the fact that both lines of the software have to communicate properly with each other. This is what makes the PSOC function as an embedded system alongside LabVIEW.

4 Putting It All Together

Figure 7 shows servos all in place and connected to the PSOC, which is connected on a breadboard and glues to the base, which is constructed from the rolled steel sheet. The following link leads to a video that shows that robotic arm that mimics the motion based on inputs of the IMU's. *NOTE: You must be signed in on a UC Berkeley account in order to have viewing access to the video.* https://drive.google.com/file/d/1AffIrMP3nC6IIzBENZquhNmiZcro3FzF/view?usp=sharing

5 Reflection & Issues

We have learned a lot when it comes to conceptualizing a model and what the process is like to put it all together. We had a rough start when it comes to the mechanical part of the design. We did not take action and get passes to the workshop or get a Makers Pass for Jacob's Hall (temporally and financially had constraints). This resulted in us purchasing cheap parts online. This was not necessarily a negative, but it would be thrilling to fully design a system from the ground up from scratch.

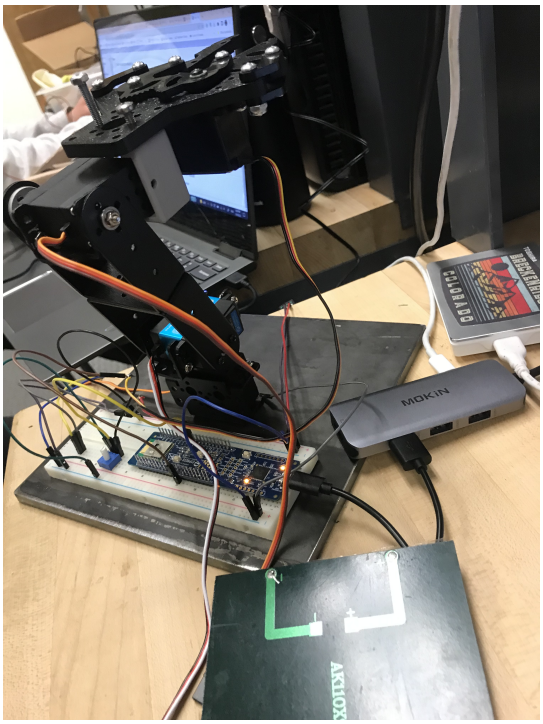


Figure 7: Robotic Arm Assembled with Servos Wired to PSOC

It was nearly all us group members' first time even coding in C, which made it much more of a challenge. We don't necessarily believe that we would have chosen differently and gone with the ESP32. The PSOC 6 ended up being the better choice for the scope of our project, but it may have been much easier to come up with the initial software if we were to have developed it using Micro-Python, and overall then could have performed more smoothly. This vigorous environment of learning C definitely made us as a group and as individuals work much harder just to get a sense of what we were doing software-wise on the PSOC side.

Obviously, a major aspect in hindsight is that it would have been a lot more enjoyable to begin the development of the software earlier on. Framed in the mind, it does not seem like it would take a lot to set up the environment of the code and initialize variables, but a lot of it is tedious work, especially when we as a group were not familiar with the syntax of C.

Currently the GUI we designed lists the angle of each servo on a dial. However, to elevate this, it would have been really cool to potentially model these values in a full on simulation where there is a 3D-like visual of the robotic arm that moves around just like the user's arm and the robotic arm. This could then dive deep into running simulations with basic sensors like the IMU's and flex sensor we used in our design, but could then be extended into a numerous amount of models or projects.