SY202 – Cyber Systems Engineering

Due Date:	**0800 on 01 May 2019 (along with Part I)**

<u>Project II – Part II: Networked Control System and Cyber Attack Threats Experiments</u>

<u>ENCLOSURES:</u>

- Final Project Report Template
- Project II – Part I
- Data sheets for IMU sensor and servo motor
- Bin files: SY202_FinalProject_sensor.bin and SY202_FinalProject_servo.bin
- C++ program template: SY202_Final_Project_mbed_template.txt

<u>OBJECTIVES</u>

- To put in practice all concepts discussed throughout the course including:
  - the design of closed-loop systems,
  - the use of actuators, sensors, and microcontrollers to regulate a physical process,
  - the use of serial communication and communication networks,
  - the assessment of system performance,
  - and the design and launch of cyber attacks to control systems
- To identify and understand the hardware necessary to control and sense within a simple cyber-physical system, in this case, a robotic arm
- To revise the use of C++ programming and the mbed for the design of control systems
- To learn how to build and use CAN network in a control system
- To evaluate via experiments the effects of external disturbances and cyber threats on the closed-loop performance of the system

<u>INTRODUCTION</u>

In this course, you have learned about the control of different cyber-physical systems including NCSs. In brief, a NCS is a time-critical and safety-critical control process where the control feedback loop is closed via a real-time communication network. It is comprised of multiple spatially distributed nodes (e.g., sensors, actuators, computers, and controllers) that can send and receive information (e.g., commands and measurement signals) through a shared communication network. The sharing of a common communication networks among different control system components increases the chances of malicious agents tampering with the control system via cyber attacks. The goal of the cyber attack is to either disrupt the performance of the system, lead the system to failure, or gain control of the physical system without being detected.

In this lab exercise, you will conduct experiments with the networked control robotic arm system illustrated in Figure 1. The system is comprised of a robotic arm (a cylindric-shaped link) mounted on a aluminum frame, a BOSCH BNO055 Inertia Measurement Unit (IMU) Sensor, a Hitech HS-475HB Servo motor, and a set of four mbed microcontrollers mounted on CAN Bus ready protoboards.

The Bosch BNO055 IMU sensor is an intelligent 9-Axis absolute orientation sensor that integrates a triaxial 14-bit accelerometer, a triaxial 16-bit gyroscope with a range of ±2000 degrees per second, a triaxial geomagnetic sensor, and a 32-bit ARM Cortex M0+ microcontroller running

Bosch Sensortec sensor fusion software, in a single package. It provides angular position information about the arm, in this case pitch. The BNO055 sensor calibrates itself when first powered and provides angular position in radians using the I2C communication protocol. For more details, you may refer to the sensor's data sheet (enclosure).

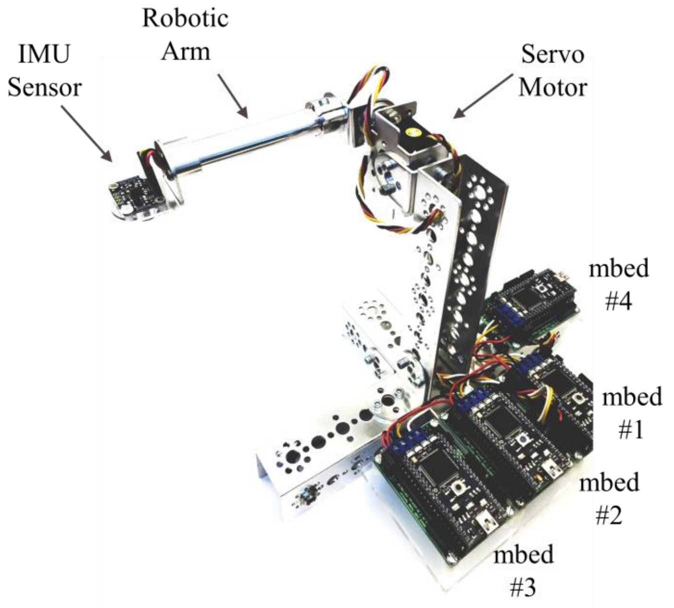

Figure 1. Networked Robotic Arm System

The Hitech HS-475HB Servo motor represents the actuator of the system. It has a range of rotation of about 180^0 and accepts a Pulse Code Modulated (PCM) signal with a minimum of 900 µs and a maximum of 2100 µs of width, where 900 µs corresponds to positioning the servo motor at $+90^o$ and 2100 µs at -90^o (or vice versa depending on the configuration of your testbed). More details on its operation are given in the data sheet (see enclosure).

In total, four mbed microprocessors are used, all interconnected using the Controller Area Network (CAN) protocol. Originally designed for the automotive industry, it has spread to a wide range of applications and industries including airplanes, medicine, manufacturing, and military systems. Each mbed in the testbed has a different function:

- Mbed #1 regulates the position of the servo motor (i.e., actuator) by reading control messages in the **CAN Bus** sent by the controller using pins p29 and p30 and passing a **PCM signal** to the servo (using pin p15)[1] according to the message received.
- Mbed #2 reads the position data (in radians) from the sensor using **I2C** protocol via pins p9 and p10. It then writes a **CAN Bus** message with the sensor data that all other mbeds can read.
- Mbed #3 acts as the controller and is the device implementing the Proportional-Integral control. It reads **CAN Bus** messages with the sensor's ID and compares the measured angular position with the desired one. Based on the error, it determines the control action to take and sends a control message via the **CAN Bus** network.
- Mbed #4 emulates any other device connected to the same **CAN Bus** network, potentially another sensor or actuator controlling a different process. In this lab exercise, mbed #4 will inject false data into the system by posing as the IMU Sensor

[1] Note that mbed's pin p15 is not a regular PwmOut output. Herein, we will be using the ServoOut.h library, which allows one to use any DigitalOut pin as a PwmOut output by implementing tickers.

with the aim of disrupting the closed-loop performance of the system. It can implement two types of attacks:

- **Stealth Attack:** Injects corrupted data into the system by adding a constant bias or measurement error to the sensor data. This attacks slightly corrupts the sensor's reading. Therefore, its effect in the system response is more slowly but harder to identify or detect by conventional fault and false data detectors, leading to the name of stealth attack.
- **Replay Attack:** Sends position (sensor) information from previously recorded sensor data. This attack effectively breaks the closed-loop system by injecting completely false information. The effects of this attack are typically more drastic.

All mbeds are connected to the CAN Bus using pins p29 and p30. When using CAN, all nodes (mbeds) can read and write messages into the bus. Messages from a particular node will carry an ID number to identify the origin. In this lab, mbed #4 will inject data posing as mbed #2.

A functional block diagram of the system is presented in Figure 2, where the external disturbances block represents external unwanted forces on the robotic arm. Similarly, a wiring schematic is given in Figure 3.

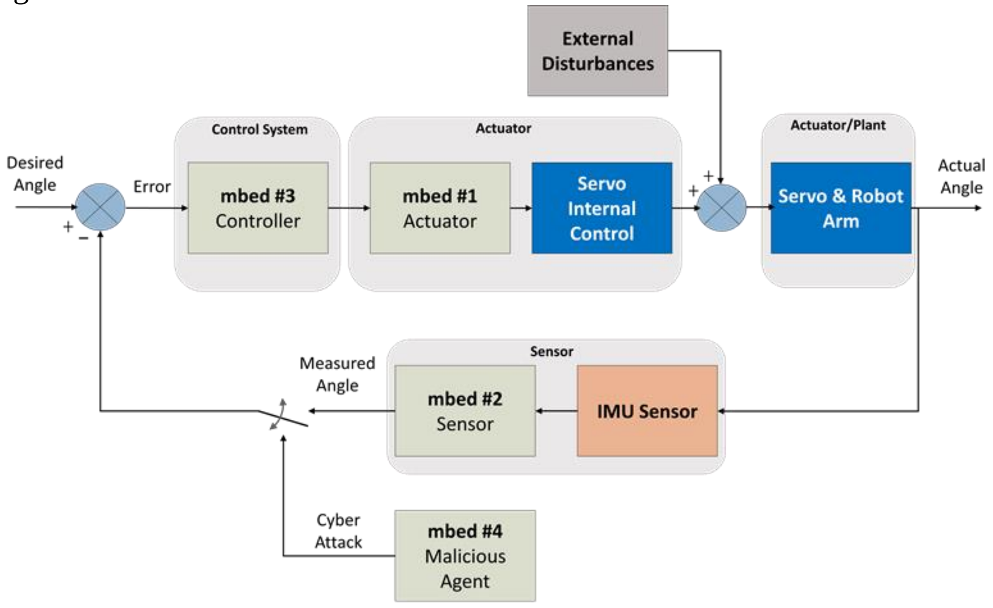

Figure 2. FBD for the NCS

<u>PART A: DESIGN OF PI CONTROL</u>

As part of the any closed-loop control system, you need at least:

- One sensor to provide feedback about the state of the plant,
- One controller to make decisions based on the feedback provided by the sensor, and
- One actuator to execute the action dictated by the controller.

In this exercise, you will design mbed program for the controller. The programs for the sensor and the actuators are being provided as two bin files: SY202_FinalProject_sensor.bin and SY202_FinalProject_servo.bin.

`SY202_FinalProject_sensor.bin` is to be placed on mbed #2. It continuously reads the IMU sensor using the I2C protocol and writes a CAN message (with ID# 2) to the CAN Bus. It sends a new message at approximately every 0.05 seconds (or at a frequency of 20 Hz) that should be read by the controller.

`SY202_FinalProject_servo.bin` is to be placed on mbed #1. This program continuously reads the CAN Bus waiting for messages coming from the controller (these are messages with ID# 3). Once a control message is received, it decodes the message and updates the control command to the servo. The control command to the servo is the width of the Pulse Code Modulated (PCM) signal that control the servo (refer to the servo data sheet and the course's Lecture #10).

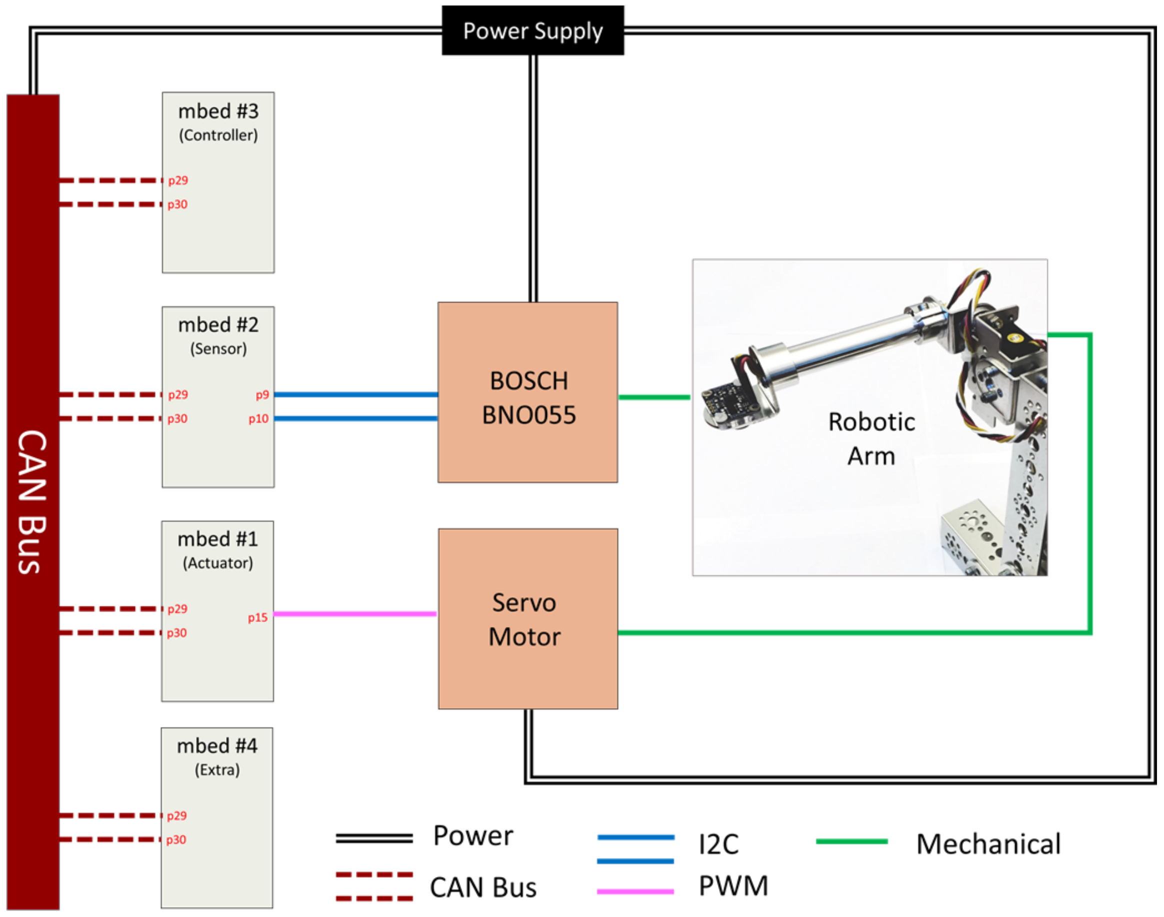

Figure 3. Wiring Schematic

Exercise:

Design a Proportional-Integral controller to regulate the position of the robotic arm at 0 radians (see Figure 4). Your program should:

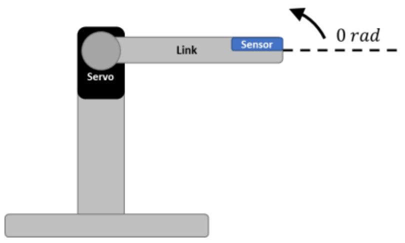

Figure 4. Control of Robotic Arm

- Periodically (every 0.02 seconds) read the CAN Bus.
- Identify sensor-related messages (CAN Messages with ID #2).
- Compute the error.
- Implement the proportional-integral control equation. This is equivalent to the pulse width of the PCM signal to be sent to the servo.
- Write a command message to the CAN Bus, to be later read by the servo. The servo waits for CAN messages with ID #3 (coming from the controller).
- Every time a CAN message is sent, print the current time, the angular position of link (data received from sensor), the error, and the PCM signal.

Procedure:

In order to complete this exercise, you will do four orderly tasks:

- Task 1: Read CAN Messages from sensor
- Task 2: Convert string message to a float
- Task 3: Write CAN Messages to servo
- Task 4: Design PI Controller
- Task 5: Modify controller to track a trajectory

Task 1 [deadline Wednesday 17-April]:

1. Start a new mbed program using the online mbed compiler. Use the `SY202_Final_Project_mbed_template.txt` given as start point for your program.
2. Start populating the first part of your program (before main function) by:
    - Declaring serial object to PC
    - DigitalOut leds 1 through 3
3. Within the main function, set the baudrate of your PC serial connection to 115200. Print to Tera Term the last name of each member as well as any instructions you would like to add. Do not forget to set the baudrate of your Tera Term terminal to 115220.
4. Set the frequency of the CAN protocol to 500000. You may use the following references for the later:

    CAN Object reference: https://os.mbed.com/docs/mbed-os/v5.11/apis/can.html
    Example: https://os.mbed.com/users/WiredHome/notebook/can---getting-started/
5. Within the while (1) loop—the cyclic control loop—use: `if(can.read(msg_read))` to check if there is a message in the CAN Bus. Then, using another if-loop statement, check if the message in the CAN Bus has the sensor ID#. Use the following reference to learn about the ID (.id) parameter as well as other parameters of a CAN Message.

    https://os.mbed.com/users/mbed_official/code/mbed/docs/8e73be2a2ac1/classmbed_1_1CANMessage.html
6. Print to Tera Term the current time along with any data message read by your controller coming from your sensor. The data of the CAN Message is stored in the .data parameter (e.g., `msg_read.data`). The data in a CAN Message is a **string** that starts with a letter 'b' (to reaffirm that the message comes from the BNO055 sensor) followed by the pitch angle of the arm in radians.

    For example: `bx.yz`, where x, y , and z are digits.
7. Toggle the value of LED#2 every time you read a message (blink).
8. Once you are done, compile and download your code into mbed #3. **Before testing your code, you should:**
    - Check that the testbed is plugged to the power supply.
    - Check that the only file downloaded in mbed #1 is: `SY202_FinalProject_servo.bin`. If it is not there, download it.
    - Check that the only file downloaded in mbed #2 is: `SY202_FinalProject_sensor.bin`. If it is not there, download it.
    - Delete any other previous code (bin file) in mbed #3.
    - Check that mbed #4 is empty (no bin file on it – someone else might have place a bin file performing an attack). If there is any bin file, then, delete it.
9. Test your program using Tera Term. You should print the current time and the messages received from mbed #2. Every message should be time-spaced by approximately 0.05-0.06 seconds in average and LED#2 should blink approximately 10 times per second. Show your results to your instructor.

<u>Task 2 [deadline Monday 22-April]:</u>

1. Once you are able to read and display CAN messages from the sensor to Tera Term, you will now extract the numeric sequence of the message (after the letter 'b') and convert into a float number.
2. This part has been deliberately left open so that you can come up with a solution.
   <u>Some Tips (optional)</u>:
   - The data part of the CAN Message from the sensor is a string of 8 bytes.
   - Convert the "data" parameter of your CAN message (e.g., `msg_read.data`) to a character array. You may use the fact that the data string has 8 bytes and that the first character is always a letter 'b' (as it comes from the sensor). So in brief, your character array should have a total of 7 characters, filled with the last 7 characters of the data string.
   - Convert the seven-character array to a float using `sscanf()`.
   - [Optional] You may create a function to do the above.
3. Print to Tera Term the angular position of the arm as a float number. Check that the float number displayed to Tera Term is the value stored in `msg_read.data` (printf statement from Task 1). Show your results to your instructor.

<u>Task 3 [deadline Tuesday 23-April]:</u>

1. You will now send/write a CAN Message to the servo indicating in microseconds the width of the PCM signal.[2] This value will be first saved in an integer variable named `pcmWidth` (see start of main function in `SY202_Final_Project_mbed_template.txt`). This should be a number between 900μs and 2100μs. A PCM with a width of 900μs commands the servo to $\pm 90^o$ ($\pm$1.57 radians) while a signal of 2100μs makes the servo (i.e., arm) rotate in opposite direction to approximately $\mp 90^o$ ($\mp$1.57 radians).
   <u>Note:</u> Each station is different. The range of motion of each servo might be much smaller than 180^o in total as well as the direction. For instance, the instructor station only covers from -62^o to 57^o.
2. Skip (for now) the PI Control section in `SY202_Final_Project_mbed_template.txt` and go directly to the Write CAN Message section in your code.
3. Convert the integer variable `pcmWidth` to a string. Note that strings meant for the servo must carry a letter 's' in front (similar to how messages from the sensor start with a letter 'b'). You can use the command:
   ```
   sprintf(msg_send, "s%d\r", pcmWidth)
   ```
   The output of `sprintf()` is saved in the empty string `msg_send` previously defined at the start of the main function. The command `"s%d\r"` attaches a character 's' at the beginning, the integer value of the pcm signal at the middle, and a return carriage at the end.
4. Using `msg_send,` format a CAN message to be sent to the servo. Example:
   `msg_write = CANMessage(`*the-ID-of-sender*`,msg_send,`*length-of-msg_send*`)`
   where *the-ID-of-sender* should be the ID# of the controller and the *length-of-the-msg_send* should be **8** in this case.
5. Write a CAN message into the network using `can.write(msg_write)`. For more details, do not forget to check the documentation for CAN in the mbed compiler website or the example in Task 1, step 4.

[2] Refer to the Introduction of this document (Servo Paragraph) or to the Data sheet included in the final project folder for details on the operation of the servo.

6. After the line in which your write (i.e., send) the CAN Message, add a printf statement to print to Tera Term the message sent to the CAN Bus. This should be done to verify that your message is being formatted properly. Recall from Task 1 that the data of the message is stored in `msg_write.data` as a string.
7. Toggle LED#3 every time you write a message (blink).
8. Compile and download program into mbed #3. Test your program using Tera Term. Show your results to your instructor.
9. In the mbed compiler, change the value of the pcmWidth signal from 900 to (a) 1200, (b) 1500, (c) 1800, and (d) 2100 and re-test your program for each of these values.
   - The servo arm should move from one extreme to the other. If for `pcmWidth = 900`, the arm of your testbed moves up, set `motionDir = -1` at the start of your main function. This variable indicates the direction of motion for the servo (similar to the polarization of the DC motor in the Elevator Project).
   - If for pcmWidth = 900, the arm moves down, set `motionDir = 1` at the start of your main function.
   - The direction of motion (clockwise or counterclockwise) of the arm will depend on how the arm and the servo were assembled.

<u>Task 4 [deadline Tuesday 23-April]:</u>

1. Recall that the control law for a proportional-integral controller takes the form of:
   $$pcmWidth = Kp * error + Ki * \int error$$
   Therefore you need to compute the error signal and the integral of the error as well as finding some suitable values for Kp and Ki.
2. In the **PI Control Section** of the `SY202_Final_Project_mbed_template.txt`, compute the error of the system, where the error is the desired angle (`desiredAngle` is defined as a float at the start of your main function) minus the measured angle (the converted float variable angle from Task 2). The error should be a float variable.
3. Then, integrate the error similar to how it is done in the Elevator Lab. That is, immediately after computing the error you should have:
   ```
   integralError = integralError + error*Ts
   ```
   Ts is the approximated time it takes to the control loop to execute (it should be close to 0.05). Make sure that `integralError` is initialized at 0 before entering the while(1)-control loop. It should already be set to zero in the template provided to you.
4. Implement the equation in step #1 of this Task. Because a pcmWidth value of zero is invalid and most servo motors may have a negative polarization (lower values increase their angular position), you will implement the modified equation:
   ```
   pcmWidth = motionDir*(int)(Kp*error+Ki*integralError)+1500
   ```
   where 1500 is an initial offset and *Kp* and *Ki* are variables you need to define. As starting point, you may use Kp=10 and Ki=1000.
   <u>Note</u>: `pcmWidth` is the value you will pass to the servo in the Write CAN Message section of your code.
   <u>Note</u>: The *error* and the `integralError` variables are floats, yet, `pcmWidth` should be an integer. Therefore, you need to type-cast the expression of the PI control into an integer value (using `int`).
5. The `pcmWidth` value may exceed the limits of the servo. Therefore, using if-statements, saturate the `pcmWidth` value to 900 or 2100 whenever the value `pcmWidth` exceeds these limits (similar to duty cycle truncation in the Elevator Lab). This `pcmWidth` value is now the one you should be sending in your CAN Message from Task 3.

6. Compile and download the program into the controller. Test your program using Tera Term and observe if the position is stabilized at 0 radians. Show the results to your instructor.
7. Tune your controller (Kp and Ki) until you get a response you are satisfied with (overshoot less than 10% and 2-3 seconds of settling time). Prove the overshoot and settling time by manually rotating the base of the system.
8. Once done tuning your controller and debugging the code, comment all print statements in the while(1)-control-loop and simply print the current time, the measured angular position of the arm, the desired angular position, the error signal, and the pcmWidth value sent to the servo. You will later export this information to MATLAB.
9. Add code to blink the LED #1 once per second. This will be used as a visual indicator that your program is running (like a heartbeat). This part is deliberately left open. You may use:
    - a counter and the fact that the control loop executes approximately every 0.05 seconds or similarly,
    - the timer `tlog` already defined in the code, or
    - a ticker (you may search for examples in the mbed website).

<u>Data Collection and Analysis:</u>

Start an experiment with the testbed stable in your desk. At approximately 5 seconds after the start, grab the testbed and turn the base by approximately 30º (as shown in Figure 5). Hold the base steady at ≈30º for approximately 5 seconds. Then, return the base to its stable position on your desk. Log data into MATLAB and plot measured angle vs time and PCM signal vs time (similar to Figure 6). Repeat the experiment and collect a second trial (it is always good to have more than one set of data in case you make a mistake).

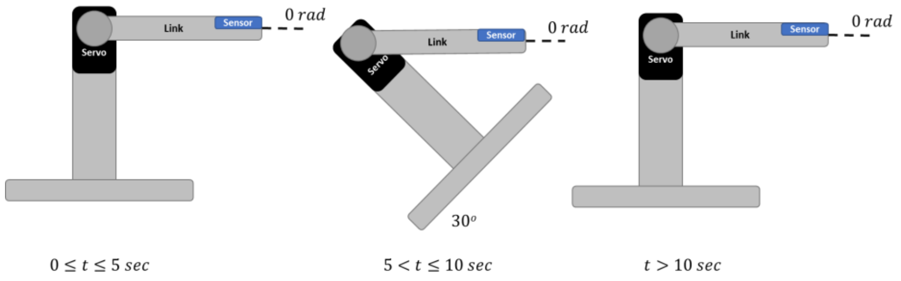

Figure 5. Experiment Sequence

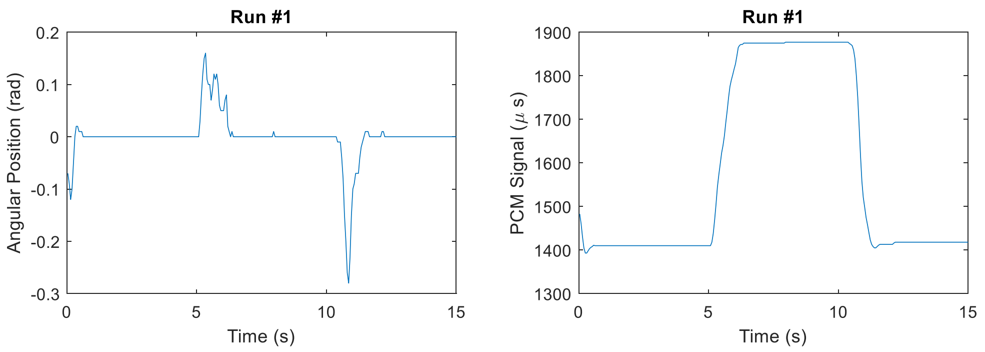

Figure 6. Example of Experimental Results

Task 5 [deadline Tuesday 23-April]:

Modify the PI Controller such that the arm can now follow a cosine wave trajectory with an amplitude of 30^o (or 0.79 radians) and a period of 10 seconds, i.e., $0.52 \times \cos(2\pi t/10)$. Right before you compute the error signal, redefined the desired angle to be now a sine function:

```
desiredAngle = 0.52*cos(2*3.1416*tlog.read()/10);
```

Data Collection and Analysis:

Run the control system when following the cosine trajectory. Log data for approximately 20 seconds. Repeat the experiment so that you have two trials (it is always good to have more than one set of data in case you make a mistake). Plot in a single plot the desired angle vs time and the measured (actual) angle vs time (similar to Figure 7).

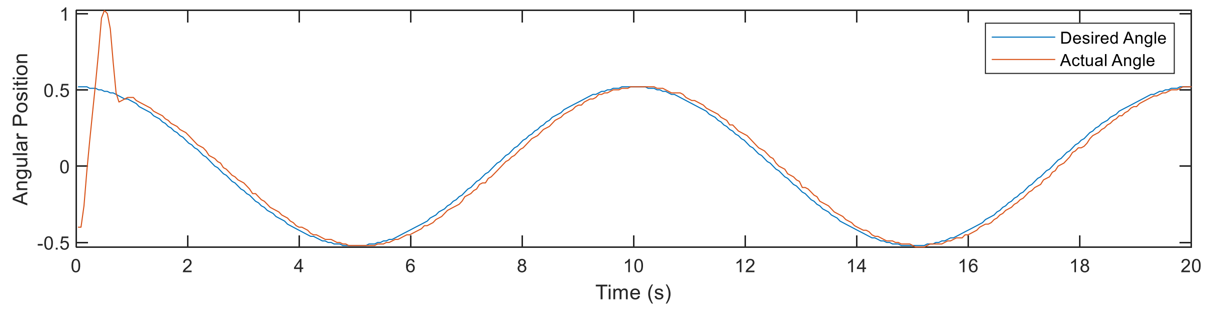

Figure 7. Trajectory Tracking

PART B: LAUNCH OF CYBER ATTACKS

The use of a shared communication network among different control system components offers several advantages. One main advantage is making data available to all devices, which might be used in more than one process. Think of an automobile. Information about the speed of your car can be used across multiple safety critical processes such as the engine and the breaks. In addition, it can be used to control less critical processes such as locking the doors of your vehicle or the seat belts as you speed up. Another advantage is the ability to easily swap, replace, and aggregate sensors, actuators, and controllers to the control system.

These advantage also may come as cost. A device connected to the same communication network can disrupt critical process. For instance, a device in the network can launch a Denial-of-Service attack by flooding a critical device with false messages and preventing critical information to reach its target. Similarly, it can launch an attack of deception by posing as another device and injecting false data into the system. In this project, you will launch two types of cyber attacks: a stealth attack and a replay attack.

Exercise: Stealth Attack

Design a stealth attack to be launched from mbed #4. The device should pose as mbed #2 by sending a CAN Message with mbed#2 ID and with the same data format, i.e., '`bx.yz`'. The angular position data '`x.yz`' should be a small difference of 0.20 radians from the "true" data, i.e.,

x.yz = measured_float_angular_position_from_mbed_#2 + 0.20

In order to write/inject a stealth attack, mbed #4 needs to first read the data coming from mbed #2 (i.e., sensor) and then write a CAN Message with the new false data. The program should execute at a rate of 2 false messages per second.

Procedure for Stealth Attack [deadline Monday 29-April]:

1. Use your code from Part A (the controller) as a reference.
2. Note that the program is quite similar. You need to:
   - read and identify CAN Messages from mbed #2 (those with the ID of the sensor),
   - get the measured angular position information as float,
   - add a small error of 0.20 radians to the measured position (as a float),
   - convert to a data string with the same format and precision as the sensor (i.e., a prefix letter '`b`' followed by the angular position with two digits after the period $x.yz$)
   - write a CAN Message with the corrupted data using as identification the ID of the sensor's mbed, and finally
   - regulate the frequency of the attack—corrupted data should be sent every 0.5 seconds (you may use the `wait()` function after sending a message or any other approach you may like: counters, tickers (best way), timers, etc.)
3. Compile, download into mbed#4, and test your program. You should see spikes in the response of the system every 0.5 seconds. Show the results to your instructor.

Data Collection and Analysis:

1. **Connect mbed #2 (the sensor) to Tera Term** and collect data for **two experiments** (similar to the final section in Part A of this final project). Do not add a disturbance by shaking or moving the base of the testbed. Simply let the testbed run with the cyber attack. Collect data for at least 30 seconds (make sure to start recording before the attack starts).
2. Import the data to MATLAB. The display from mbed #2 to Tera Term should have 3 columns. The first column is the time signal running in mbed #2 (which is different to the time signals of all other mbeds). The second column is the angular position in radians and the third column is the CAN message sent by the sensor.
3. Plot the response of the system: measured angle vs time (similar to Figure 8 – lower plot). You will need it for your report. You may need to estimate the time at which the experiment started and subtract that number from the time vector when plotting your data.
4. Change the size of the error to 0.10, 0.30, and 0.50. Similarly change the frequency of the attack to: 1 attack/second, 4 attacks/second, and 20 attacks/second. Annotate any observations.

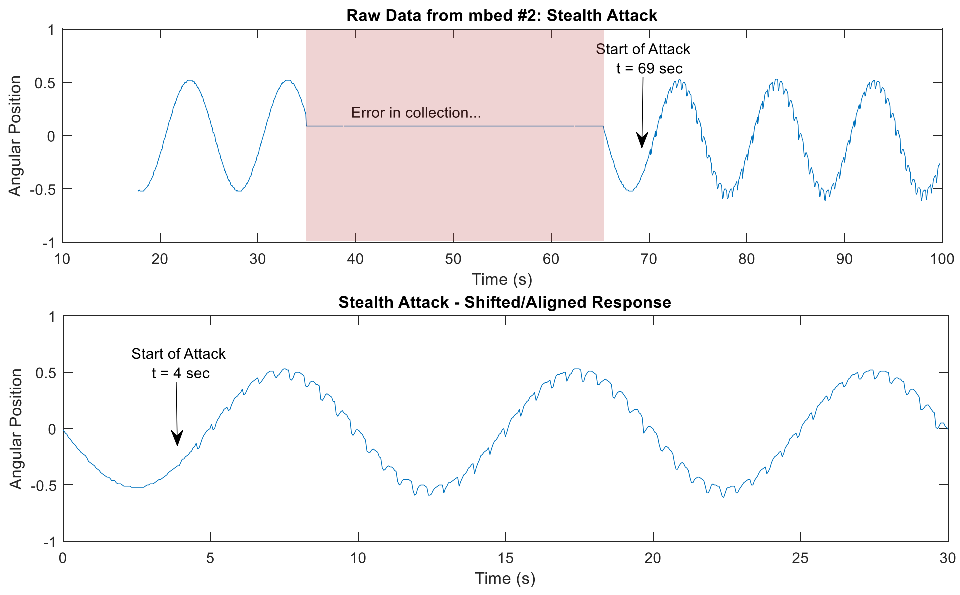

Figure 8. Response of the System under a Stealth Attack with magnitude of 0.5 rad.

<u>Exercise: Replay Attack</u>

Design a replay attack to be launched from mbed #4. The attack should record data for the first 9.5 seconds and then replay (send) the recorded data indefinitely (repeating every 9.5 seconds). The device (mbed #4) should pose as mbed #2 by sending a CAN Message with mbed#2 ID and with the same data format, i.e., '`bx.yz`'. The angular position data '`x.yz`' however, should be the recorded message.

<u>Procedure for Replay Attack [deadline Tuesday 30-April]:</u>

1. The design of the code is deliberately left open. Some things to consider:
    - The sensor sends information at approximately 20Hz or (20 messages per second). Consider this when creating an array to save 9.5 seconds of data.
    - You may alternatively use a timer (such as the tlog you use the controller code) to regulate the collection and replay of data.
2. Compile, download into mbed#4, and test your program.

<u>Data Collection and Analysis:</u>

1. Connect mbed #2 (the sensor) to Tera Term and collect data for **two experiments**. Start the replay attack (the recording) once the system is successfully tracking the cosine trajectory. Collect data for at least 60 seconds.
2. Import to MATLAB and plot the response of the system: measured angle vs time (similar to Figure 9 – lower plot). You will need it for your report.
3. Change the length of the replay attack from 9.5 to 5 sec and 10 sec. Annotate any observations.

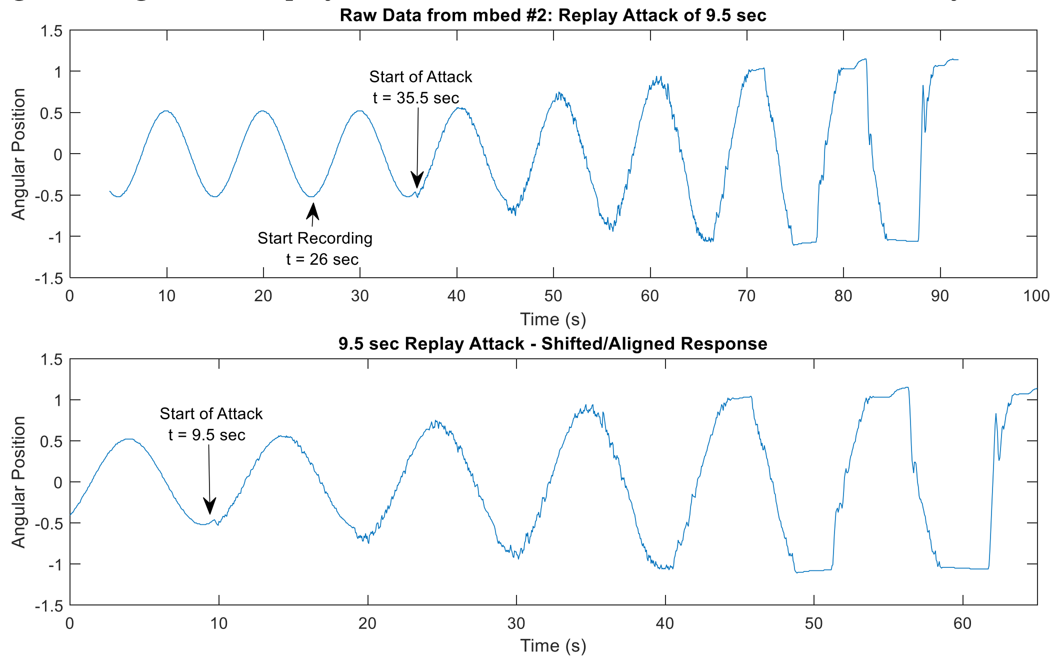

Figure 9. Response of the System under a 9.5 second duration Replay Attack. You can notice how overtime the oscillations become larger. The oscillations are stopped by the mechanical limits of the servo motor (it cannot exceed approx. ± 1.1 radians).

<u>REPORT GUIDELINES [Deadline 01 May]</u>

- Demonstrate completion (**live demo/provide .bin file for each required demonstration**) of Part A, Task #4 and Task #5 and Part B, Replay Attack (of 6 sec) to your instructor.
- Follow the final project report template and the general lab guidelines for SY202 lab reports. Refer to the lab rubric for the grading of the report.