

CprE 488 – Embedded Systems Design

MP-4: UAV Control

Assigned: Monday of Week 11

Due: Monday of Week 13

Points: 100 + bonus for advanced control capabilities

[Note: controlling dynamic systems is an essential usage scenario for embedded devices. While both MP-1 and MP-3 involved using our ZedBoard to direct a mechanical system (the quad UAV in MP-1, the USB rocket launcher in MP-3), we experienced the significant challenges involved in attempting to control a device with little to no corrective response (an open-loop system). Now that we have sufficient experience with FPGA-based embedded system design, we will incorporate feedback to be able to control our UAV more effectively. Specifically, the goal of this Machine Problem is for your group to gain familiarity with three different aspects of embedded system design:

- 1. Advanced serial interfacing – you will configure your Zynq processing system to send serial data over a Bluetooth PMOD interface.*
- 2. Telemetry and data interpretation – you will reverse engineer the serial data protocol of the quad UAV platform in order to set channel inputs and read back sensor data.*
- 3. Control implementation and tuning – you will implement and later tune a basic PID controller to command the quad in order to help to stabilize its flight.*

In anticipation for your final projects, it is expected that you will be able to leverage your experience to get through the main phases of this assignment without nearly as much explicit step-by-step instruction.]

1) Your Mission. “Get to the choppa!” Although the rest of your team of elite special forces didn’t make it, you (Major Alan “Dutch” Schaefer) still have a chance to evacuate. There are only two things sitting between you and a pleasant flight to freedom: 1) a trophy-hunting, sentient humanoid extra-terrestrial *Predator*, and 2) the small fact that your piloting skills are questionable at best.

Getting around the Predator is relatively straightforward: summon an alien from the *Aliens* franchise to defeat it in battle, and then use your MP-3 USB missile launcher to take out the alien. Obviously. However, our MP-1 experience clearly demonstrated that piloting the quadcopter is a non-trivial task. For that reason we will explore using telemetry data to feed a basic PID control loop in order to properly control the quad. Armed with only a Zedboard and a link to a high-end geosynchronous GPS satellite, you only have two weeks to complete your flight assistance system before the guerilla rebel forces descend upon your base camp, so you had better get started!

2) Getting Started. We have retrofitted each of the Mini Fly QuadCopter ARF systems with a Bluetooth module that can be used for feedback and control. To communicate with the module, follow these high-level steps:

- While there is no code provided with MP-4, we have several documents available in the MP-4 zip file. Retrieve this file from the link provided to you via email.
- Create a standard ZedBoard design in VIVADO that incorporates the switches and buttons.
- Configure “UART 0” of the processing system to connect to PMOD-A using the EMIO interface. Documentation that will assist with this step:
 - Section 2.5 of docs/Zynq/ug585-Zynq-7000-TRM.pdf provides an overview of the system-level MIO and EMIO I/O routing architecture.
 - Scan Chapter 19 of the same document to get a better understanding of the UART subsystem. For the purposes of routing through EMIO to PMOD-A, Section 19.5 contains some useful information.
- Propagate the TX/RX pins of “UART 0” to the PMOD-A interface. The appropriate port names are described in the docs/Zynq/pg082_processing_system7.pdf document.

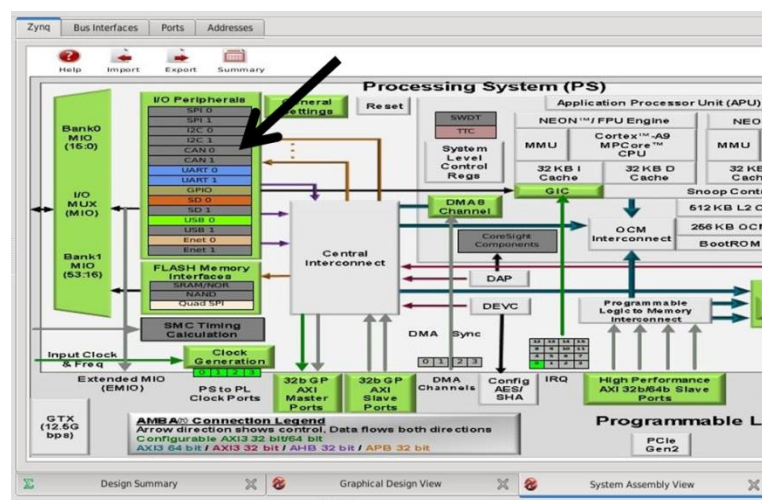


Figure: EMIO configuration region of the Zynq Processing System in VIVADO

In your lab write up explain how you routed the “UART 0” signals to external pins on the ZedBoard (i.e. the PMOD-A interface)? Generate a bitfile for this project and export to SDK.

3) Bluetooth Configuration. Create a software project (called p3_UART0_test) that sets the UART0 baud rate to 115200, and writes to and reads from UART0. Some suggestions as you go through this part:

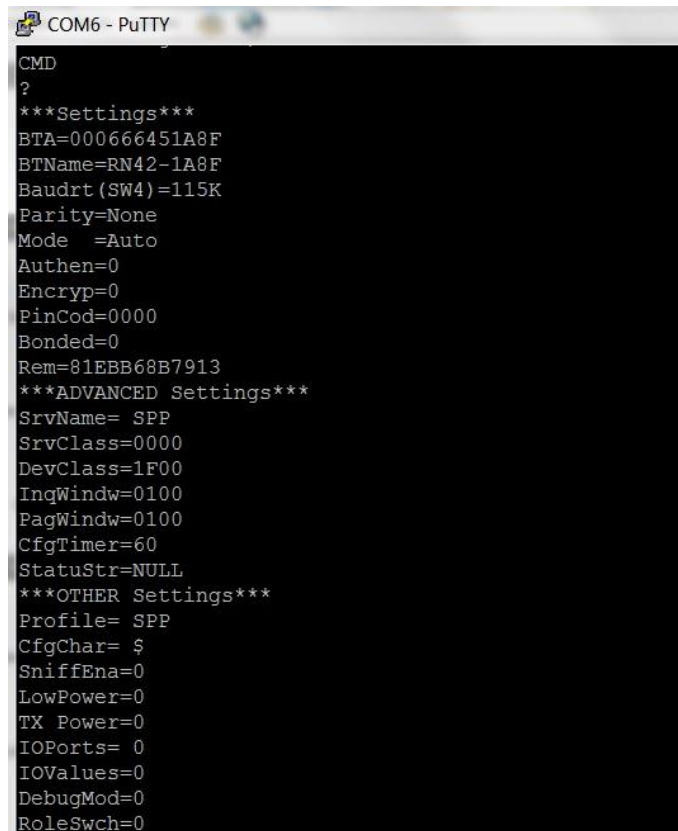
- We haven’t connected anything (yet) to PMOD-A, so one easy way to test both reads and writes is to connect a loopback wire to the appropriate pins in PMOD-A.
- The ug585 document provides guidance for writing UART software, specifically Section 19.3 and Appendix B.
- The SDK-generated UART drivers are also useful: see xuartps_hw.c, xuartps.c, and associated header files.

In your writeup, provide a description of the registers most relevant to enabling the UART0 subsystem’s RX and TX ports, and reading and writing bytes over the UART0 interface.

Next, your job is to configure the Bluetooth PMOD by sending AT commands via the UART0 interface. For this section, refer to both `Bluetooth_PmodBT2_rm.pdf` and `Bluetooth_ATcommands.pdf` documents available under `docs/Bluetooth`. Create a new software project (called `p3_BT_configure`) that can send the appropriate configuration commands. To avoid damaging both the ZedBoard and the Bluetooth PMOD peripheral, follow these precise steps:

1. Power down the ZedBoard before plugging in the Bluetooth PMOD into PMOD-A.
2. Via software, place the Bluetooth PMOD in configuration mode (confirm that the LEDs match what is described in `Bluetooth_ATcommands.pdf`, Section 7.3).
3. Check the Bluetooth configuration (commands 'D', 'E', and 'O').
4. Set the Bluetooth to operate in auto-connect mode 3.
5. Set the remote MAC address of the Bluetooth PMOD to be the quadcopter bluetooth module's MAC address (each quad is labelled with its corresponding Bluetooth MAC address).
6. Set the PIN to "0000".

In your writeup, provide a screenshot of your Bluetooth PMOD configuration settings. It should look something like the following:



```
COM6 - PuTTY
CMD
?
***Settings***
BTA=000666451A8F
BTName=RN42-1A8F
Baudrt (SW4)=115K
Parity=None
Mode =Auto
Authen=0
Encryp=0
PinCod=0000
Bonded=0
Rem=81EBB68B7913
***ADVANCED Settings***
SrvName= SPP
SrvClass=0000
DevClass=1F00
InqWindw=0100
PagWindw=0100
CfgTimer=60
StatuStr=NULL
***OTHER Settings***
Profile= SPP
CfgChar= $
SniffEna=0
LowPower=0
TX Power=0
IOPorts= 0
IOValues=0
DebugMod=0
RoleSwch=0
```

7. Switch the module back to Data Mode, and power off the ZedBoard.
8. Power up the quad (the quad's Bluetooth must be powered before the Zedboard's Bluetooth), then power the ZedBoard back up.
9. Send the following commands to the quadcopter: `0x24 0x4d 0x3c 0x00 0x6c 0x6c`. If everything is configured properly, the quad will respond with the following data: `0x24 0x4d 0x3e 0x06 0x6c 0xXY`

0xXY 0xXY 0xXY 0xXY 0xXY 0xXY, where 0xXY is the quad sensor data we will process in the next section.

4) Quad Commands. Create another software project (called `p4_quad_commands`) that will be used to send and receive data from the quad. **IMPORTANT: for the remainder of the lab, make sure the quad is tied down, and an RC controller is powered on and bound to the quad. In the event the quad stops responding to Bluetooth commands, power off the Zedboard. The RC controller will automatically take over control. In general, if the quad is not continuously receiving Bluetooth channel commands, it will return control to the RC controller.**

Scan the following documents to become familiar with the quad's serial command format, and what commands are available: i) `docs/Quad/MultiWii_Serial_Protocol.pdf`, ii) `docs/Quad/Protocol.cpp`, and iii) http://www.multiwii.com/wiki/index.php?title=Multiwii_Serial_Protocol.

In your `quad_commands` app, perform the following steps:

1. Request raw inertial measurement unit (IMU) data from the quad (i.e. `MSP_RAW_IMU`)
2. Request orientation information from the quad (i.e. `MSP_ATTITUDE`)
3. Send arm and disarm commands to the quad (i.e. `MSP_SET_RAW_RC`). Switch between armed and disarmed about every 10 seconds.
 - a. ARM: Arm the quad by sending a value less than 1100 to the throttle channel, a value greater than 1900 to the yaw channel, and mid-range (~1500) values to the roll and pitch channels. When the quad becomes armed, a red light will turn on and the blades will spin at a slow rate (motor-dependent).
 - b. DISARM: Disarm the quad by sending a value less than 1100 to the throttle channel, a value less than 1100 to the yaw channel, and mid-range (~1500) values to the roll and pitch channels. When the quad is disarmed, the red light will turn off and the blades will stop spinning.

In your writeup, provide the command ID number for requesting raw IMU data, orientation information, and sending Roll, Pitch, Yaw, and Throttle commands.

5) PPM Passthrough. Next we will leverage the PPM capture module from MP-1 in order to create a software passthrough of the command signals from the PPM input to the Bluetooth output. In your writeup, provide a diagram for the high-level system architecture for this part. Show how the following components connect to one another: i) UART0 subsystem, ii) MIO/EMIO interface, iii) PMOD-A (with specific usage of pins), iv) Bluetooth PMOD, v) PPM capture module, vi) PMOD-B (with specific usage of pins), vii) the ARM processor.

Add your PPM capture module from MP-1 into your existing XPS project. A solution for the MP-1 `axi_ppm` pcore will be posted on CANVAS for those who need it. Export this design back to SDK, and create a new software project called `p5_PPM_BT_passthrough` that continuously reads the Roll, Pitch, Throttle, and Yaw commands captured by the `axi_ppm` module, and sends these values to the quad over Bluetooth.

Note: to ensure that arming and disarming still work through this passthrough code, for the yaw channel, the maximum value (left stick to the right) must be at least 1900, and for the throttle channel, the minimum value (left stick down) must be less than 1100.

To verify that this `PPM_BT_passthrough` application is working correctly, arm the quad and then fly it around using the PPM passthrough. The transmitter that you use to test this portion should not need to be paired with the quad, since we are using it only for the PPM data, and not directly for RF communication to the quad.

6) Data Analysis. Create a new application (called `p6_sensor_analysis`) that displays the orientation of the quad. Use the `MSP_RAW_IMU` command from the previous section to collect IMU data and compute the quad orientation: a) based on accelerometer data only, and b) based on gyroscope data only. Refer to `docs/Quad/ITG_3205.pdf` (gyro data sheet), and `docs/Quad/BMA180.pdf` (accelerometer data sheet).

Have a team member hold the quad, statically, at different orientations. Cross-check your computed orientation against the quad's computed orientation (i.e. `MSP_ATTITUDE`). In your write up, provide a table showing your computed orientation and the quad's computation of its orientation, and briefly explain how you derived Roll and Pitch and how your results compared against the quad's calculated values. Was it necessary to know the units of the accelerometer?

How does shaking the quad impact your estimation of roll and pitch when a) using accelerometer data only, b) when using gyroscope data only? In your writeup, briefly describe your observations.

7) Pseudo-GPS. For this part you will interface with the pseudo-GPS client that has been provided in the `MP-4/Utils` directory. Note that we use "pseudo" GPS as real GPS does not work well indoors. Our client provides the X,Y,Z position of the quadcopter in centimeters, and the Yaw direction (D) in degrees.

From the command prompt on your windows workstation, navigate to the appropriate directory, and run the pseudo-GPS client with the following command:

```
> getGPS.exe 10.24.85.134 4560 6
```

For this command, the '4560' value corresponds to the port that is open on the server, and '6' refers to COM6 on your Windows workstation. The `getGPS` client will continuously request pseudo-GPS information from the quad tracking server, and will check the Zedboard COM port for GPS requests. 4650 is the UDP port that the server listens to.

Back to SDK, create a new application (called `p7_request_GPS`) that will interface with the pseudo-GPS client:

- **Request GPS:** to request a GPS update, send an 'S' character to the ZedBoard to the client.
- **Request GPS with debug message:** since we are now using both UART interfaces for purposes other than printing messages, PuTTY will not be usable for the rest of this assignment. To provide a debug message to be printed out, send a 'D' character to the client, followed by a 1000-byte debug string. The client will respond with a GPS update, and display the debug string.

Due to the overhead of sending long messages over UART, you will want to use the debug feature selectively (i.e. when a button is pressed on the Zedboard).

To confirm that your ZedBoard is interfacing with the pseudo-GPS client appropriately, have it send the most recently-received X,Y,Z,D data to the client as a debug message when the center button has been pressed. In your writeup, provide a screen shot of the pseudo-GPS client displaying your debug message.

8) PID Controller. Create a new application called `p8_PID_control` (this is the last one, we promise) that implements a generic PID control function for controlling the quad's yaw. Refer back to HW-4 for the appropriate equations. Some suggested structure for this application:

- You will be calculating the PID value repeatedly, so create a separate function called `compute_pid(pid_t *myPID)`.
- The PID structure should have the following members, and all should be single-precision floating-point:
 - `sensor` // Current value of property being controlled
 - `setpoint` // Goal value of the property being controlled
 - `KP, KI, KD` // P, I, D constants
 - `prev_error` // Previous difference between Current and Goal value
 - `acc_error` // Accumulated error
 - `pid_correction` // Correction output of the PID controller
- For determining the current value (and error), the application should request yaw direction information from the pseudo-GPS client.
- Use the center button to configure your application to provide debug information to the pseudo-GPS client. Useful debug values include: i) the current yaw direction, ii) the setpoint yaw direction, iii) the correction value computed by your PID controller (total correction and P, I, and D contribution, and iv) the yaw command to send to the quad.
- Use a switch to switch between sending yaw commands from the RC controller, or from your PID controller. Note that just as in the previous section, the roll, pitch, and throttle should all be directly passed through from the sampled PPM values to the Bluetooth.

Do NOT test with a live quad until you have demonstrated to the TA that your PID controller is behaving reasonably. One way to test this is with a quad that has its battery disconnected – by manually rotating the quad and checking the output of the `PID_control` application (via the debug messages sent to the pseudo-GPS client) you should be able to confirm that the controller is attempting to turn the quad in the right direction.

For example, if your setpoint for yaw is 0 degrees, and the measured yaw is 20 degrees, does the PID controller try to correct properly? You will need to test the behavior of the individual P, I, and D components. Explore how different values for the P, I, and D constants affect the PID controller output. In your writeup, describe your testing procedure, observations, and how your experiments with PID correction output helped you to derive an initial constant for P.

9) Flight Testing. For all testing in this section, 1) make sure the quad is tethered, and that an RC controller is powered on and bound to the quad. 2) In the event the quad stops responding to Bluetooth commands, power down the Zedboard. The RC controller will automatically take over control.

First, test that the yaw control is working when the quad is securely tied to the turntable. Run your `PID_control` application to test if the PID behaves reasonably when the turntable is nudged. Adjust your PID constants appropriately (you will likely only need a P correction component). In your writeup, explain your testing procedure, observations, and the constant you chose for your P at this stage.

Next, assuming the yaw stays stable in the turntable configuration, tether the quad to the ground and manually control pitch, roll, and thrust, while the ZedBoard controls the yaw component. To confirm that your PID controller is properly functioning, make the setpoint 0 degrees, and orient the quad on the ground at 30 degrees. As you take off, the yaw controller should correct the direction of the quad. In your writeup, explain your testing procedure, observations, and any modifications made to your design at this stage.

What to submit: A .zip file containing your modified source files (your final UART0_test.c, BT_configure.c, quad_commands.c, PPM_BT_passthrough.c, sensor_analysis.c, request_GPS.c, PID_control.c) and your writeup in PDF format containing the highlighted sections of this document. In the Blackboard submission, list each team member with a percentage of their overall effort on MP-4 (with percentages summing to 100%).

What to demo: At least one group member must be available to demo the current state of your implementation. A full demo score requires a system that can maintain the quad's Yaw direction using the PID controller, can switch between manual and PID yaw control, and can modify Yaw setpoint, but partial credit will be given for effort. Be prepared to briefly discuss your source code.

BONUS credit. While having autopilot Yaw PID control does ease flying the quad, it will be more interesting to have more channels successfully simultaneously under autopilot PID control:

- Roll or Pitch control in addition to Yaw (5 points)
- Roll, Pitch, and Yaw control (15 points)
- All 4 channel auto pilot – can your quad maintain a hover with small external disturbances? (30 points)
- All 4 channel auto pilot – can your quad follow a flight plan that moves in a stable fashion between X,Y,Z coordinates? (40 points)

Each group is limited to 100 bonus points for the entire semester.