Final Project

EE 474 Introduction to Embedded Systems
Lab 5

Daniel Park
Samuel Johnson
Reilly Mulligan

Introduction

The purpose of this lab was to really demonstrate what we've learned from this class, and we did this by creating an Android app that controls a tank via bluetooth. Specifically what makes our project interesting is our decision to not use any APIs, excluding the APIs included with Android studio, our decision to write a robust Android app from scratch, and our app's support for tilt-driven steering. These design decisions agreed with a philosophy to both work with what we had (what we were given for the class, as well as the phones we already had) and to learn as much as we could about the systems we were using.

Hardware Connections

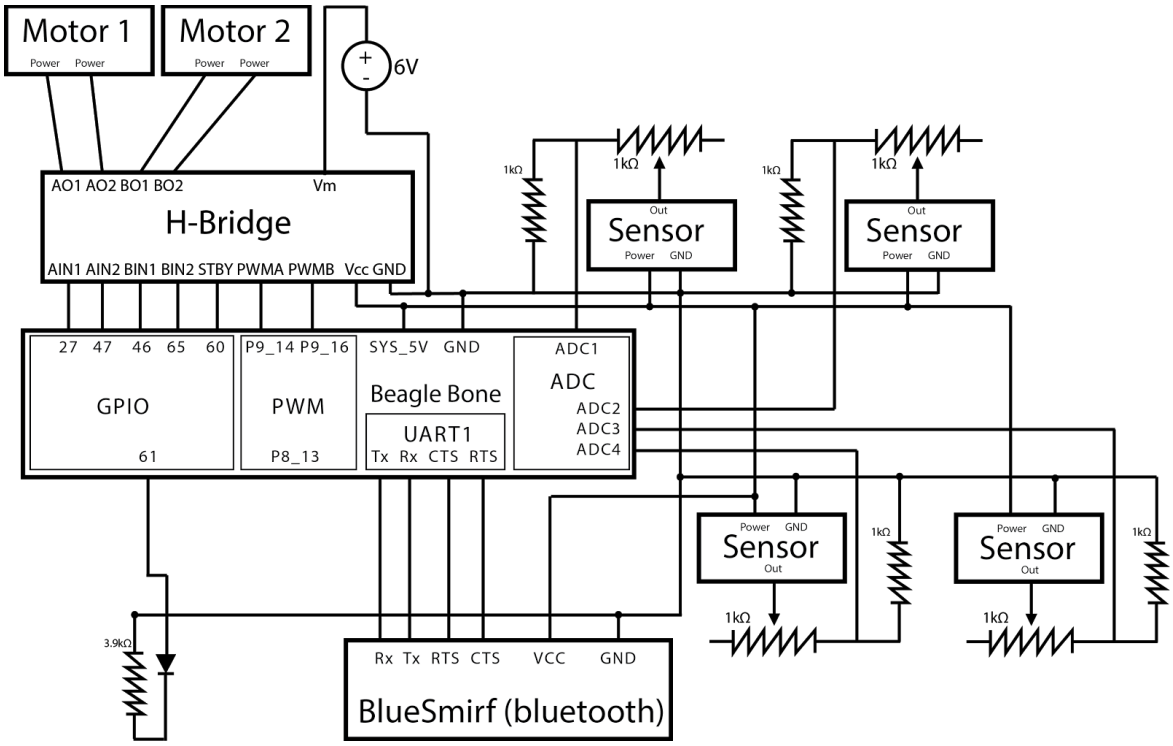


Figure 1: Hardware connections on the BeagleBone.

As shown in Figure 1 above, the GPIO and PWM pins are connected to the motor through the H-bridge. By controlling the motors in this way, the vehicle has the ability to move forward, back and turn, simply by switching the values on these pins. Furthermore, ADCs 1 - 4 on the Beaglebone are attached to the output of distance

sensors so that the distance to an object on every side of the vehicle can continuously be monitored. By wiring the Beaglebone in this way, it has the ability to drive the motors in reaction to elements in the outside world.

Additionally, there is one GPIO pin that is hooked to an LED. The LED serves the important function of notifying the user when the Beaglebone has successfully self-booted and is ready to be switched on and be controlled by the app on the android phone.

Finally, the BlueSmirf bluetooth module is wired to the beaglebone through the UART1 port which is designed to take a serial input and output. By wiring the transmit of the beaglebone to the receive of the BlueSmirf module, and vice versa, the Beaglebone is able to communicate with the android app via bluetooth. This hardware component is a necessary part in sending commands to the Beaglebone.

On the phone side of the hardware, the geomagnetic sensor and the accelerometer were used to determine the pitch, roll and asmuth of the phone in tilt-drive mode.

Software Overview

Our software consists of four scripts that control the Beaglebone and its driving capabilities. The first script is bt_listener.exe, which is the main control for the driving capabilities of the beaglebone. This script processes commands from the android app, and is the highest controlling script running on the beaglebone. The second script is called tank_entry.exe, which is the process that initializes the driving capabilities of the vehicle and puts it into a “standby” state. Tank.exe controls the driving functions of the tank, while adc_listener.exe polls the distance sensors and sends them to other processes when necessary. These processes work in parallel, and communicate through the use of signals to drive the vehicle. The specifics of the four scripts will be discussed more extensively in the following sections.

Our control system consisted entirely of a single Android app. The app consisted of 2 activities, which each represented drive modes for the tank. Both drive modes and details about the app are contained in the section labelled “TankDrive.”

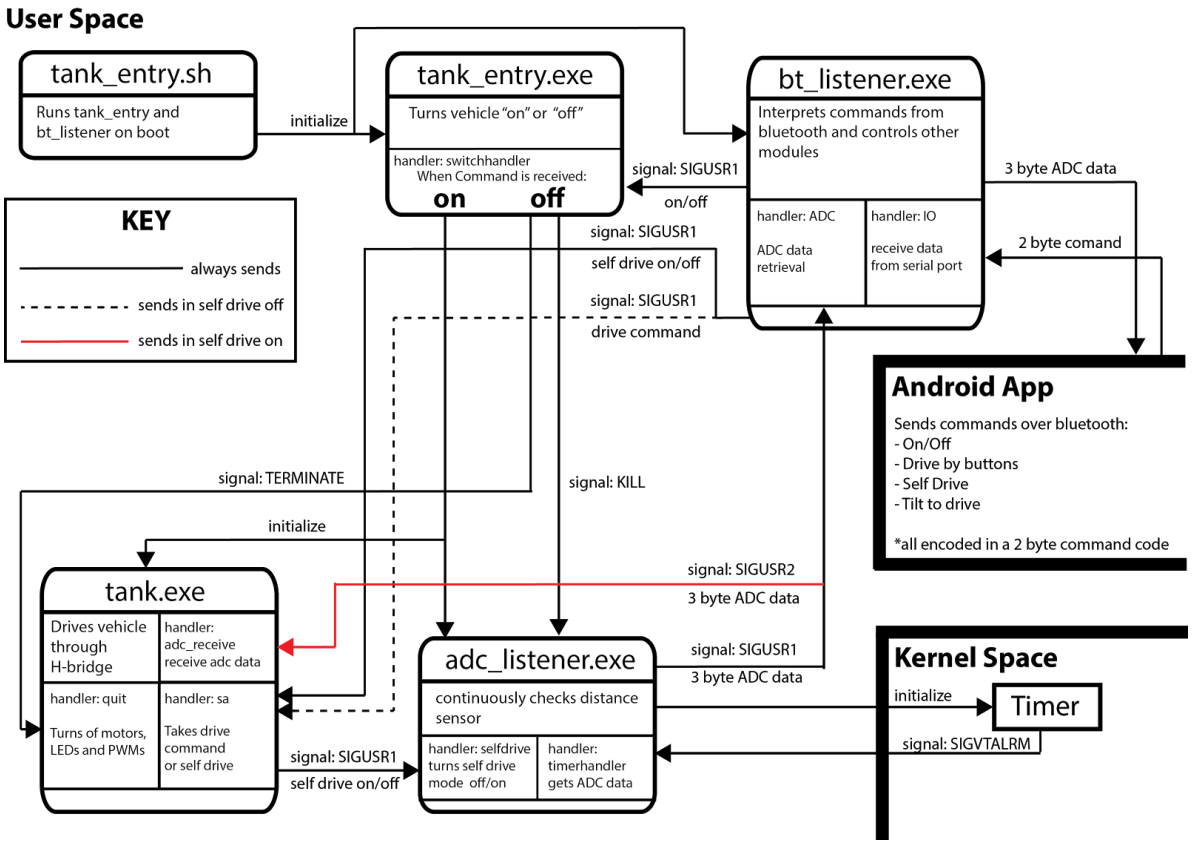


Figure 2: Diagram showing software interactions

Table 1: 2 byte commands from App to bt_listener.exe

2 byte command	**Action**
0xFF00	Power Off
0xFFFF	Power On
0xFF01	Self Driving On
0xFF02	Self Driving Off
Drive commands: - first byte is left motor, second byte is right motor - first bit represents direction (1 = forward, 0 = reverse) - other 7 bits represent duty - e.g. 0xFAFA is drive forward	Maneuver tank

bt_listener.exe

This script is the highest controlling script that runs on the beaglebone. It receives commands from the android app through the UART1 serial port via bluetooth and processes these commands to be executed by other processes on the beaglebone. These commands are encapsulated in two bytes, and include normal driving commands that are recognized by bt_listener.exe and sent to tank.exe to maneuver the beaglebone.

However, there are other more important commands that bt_listener.exe receives from the android app. Specifically, there is the overall on/off command for the driving ability of the beaglebone. When bt_listener.exe receives an overall on command, it sends a signal to tank_entry.exe which creates tank.exe and adc_listener.ex, while an off command terminates these two processes via tank_entry.exe. In this way, bt_listener.exe controls the overall behaviour of the beaglebone by executing the commands that it receives.

Additionally, bt_listener.exe can process a command which puts the beaglebone into self drive mode by sending a command to tank.exe. This causes bt_listener to avoid all drive commands that it receives until self drive mode is terminated, or an overall off command is received. This allows tank.exe to autonomously drive the beaglebone without interference of user-based drive signals.

Finally, bt_listener also processes distance sensor information that is received from adc_listener.exe and sends it back to the android app via the UART1 port. At the moment, this information has not been utilized on the app, but with more time the app could easily be made to display distance information that is being sent from bt_listener.exe.

tank_entry.exe

This is the script that initializes the vehicle to on mode and runs tank.exe and adc.listener.exe. The script takes an interrupt signal from bt_listener.exe as input to control whether the vehicle is in self-drive mode or not. When an on command is received, the script creates two child processes, one for running tank.exe and one for running acd_listener.exe. The process will then continue to run. If an off command is ever received, tank_entry will send a kill signal to its child processes, adc_listener and

tank, to kill them individually. These signals will then be caught by adc_listener and tank, respectively, to exit their processes by first switching off any active processes to return to idle mode. This includes switching off motors, and LEDs until an on command is received again

This process was a challenge to implement because of the subtle quirks involved with the beaglebone naming scheme for PWMs and ADCs. We initially had tank and adc_listener both initialize the PWMs and ADCs that their scripts required. However, this methods doesn't work when tank_entry calls tank and adc_listener because the processes are called in parallel. When they are called in parallel we couldn't guarantee which process would run first, so sometimes the dev files would be create in the right order and our script would work, while other times the dev files would be created in the wrong order, and our scripts would cease to work. We solved this problem by initializing the required pins in tank_listener itself.

Another challenge of implementing this script was creating the child processes and communicating with them successfully. A problem we had was that our original implementation of this script would create zombie processes of its children instead of completely killing them. This meant that the second time the switch was flipped, the script wouldn't run because signals were being sent to the wrong scripts. Getting this process to fork, and successfully kill its children each time was difficult to implement correctly.

tank.exe

This script is the script which drives the vehicle. It does this by interfacing with the H-bridge using the PWMs and GPIOs. The functionalities for driving forward, turning, and self driving are contained in this script. The process has two driving modes, which are controlled through the android app via bt_listener.exe: manual and self-driving mode. When a command to turn on self-drive mode is received from bt_listener.exe via a signal, the signal is handled by an interrupt which sets a flag to begin self-driving. The vehicle will return to manual drive mode when a self-drive off command is received from bt_listener.exe via a signal.

During the manual drive mode, this executable continually receives commands from the android app via bt_listener.exe. The tank will then drive the motors based on the command that it receives. The form factor of the 2 byte command is described in Table

1. When self-drive mode is enabled, this process sends a signal to adc_listener.exe to notify it of this change. Adc_listener will then begin sending the distance sensor values via a signal to tank.exe. This signal is caught and the distance values are stored in a global variable to self drive the vehicle. Self-drive will continue to drive the vehicle forward until iit sees that it is getting too close to an object in the front based on the data continually being received from adc_listener.exe. Once this happens, the script will enter a portion of the code that blocks the signals being sent from adc_listener.exe, in order to prevent sleep commands from being interrupted and prematurely terminated. Once in this state, the tank will reverse for a short amount of time and then turn either left or right, depending on the values of the distance sensors from either side to avoid bumping into the object that is too close. Also, if at any point during self-driving there is an object close behind the vehicle, it will increase its speed to run away! After the beaglebone changes direction it will continue to drive forward until it senses an object too close in the front.

Tank also has an interrupt handler which catches a terminate signal. This handles ensures that whenever tank is terminated, the motors, LEDs, and buzzer are turned off when the process exits. This is a very important feature to implement, because otherwise the vehicle would continue to drive forward for all of eternity (or just till the batteries run out). Implementing this is also key for tank_entry to have full control over this process.

adc_listener.exe

The adc_listener.c is responsible for continuously checking the output value of the four distance sensors, which are attached to the front, left, right, and rear of the drive base. To do this, the script initializes a timer which continuously sends a signal to adc_listener to obtain the value on the distance sensor. This is done at a specified frequency, specifically at 200 Hz. Each time the process receives this signal, it checks the output of each distance sensor and stores it, averaging the values that are received every 20 samples to cut out noise. This creates usable data at a rate of 10Hz which is continuously sent to bt_listener.exe, which sends the data to the android app. If the self-drive flag has been set by a self drive signal sent from the android app via bt_listener.exe and tank.exe, then the distance sensor information will also be sent to tank.exe via a signal. This is so that tank.exe can determine how to drive the tank in self drive mode.

Untethered Mode (tank_entry.sh and tank_entry.service)

The vehicle is also capable of booting itself, and running untethered. To do this, we created tank_entry.sh, a bash script, which runs tank_entry on boot of the system, and tank_entry.service, a file that sets up a service to run our executable. tank_entry.sh was placed in /usr/bin and given proper permissions, and tank_entry.service was placed in /lib/systemd. A symbolic link was created in /etc/systemd/system to let the system know the location of the service. After following these steps, tank_entry.exe runs on boot and allows the it to run untethered.

TankDrive (Android App)

TankDrive is the Android app that we created to control the tank using a Bluetooth connection. The app has functionality that allows the phone to turn the tank on and off, make the tank drive autonomously, control the tank's speed, and it allows the user to drive the tank in 2 different modes: one that uses 4 arrow keys and one that is controlled based on the rotation of the phone.

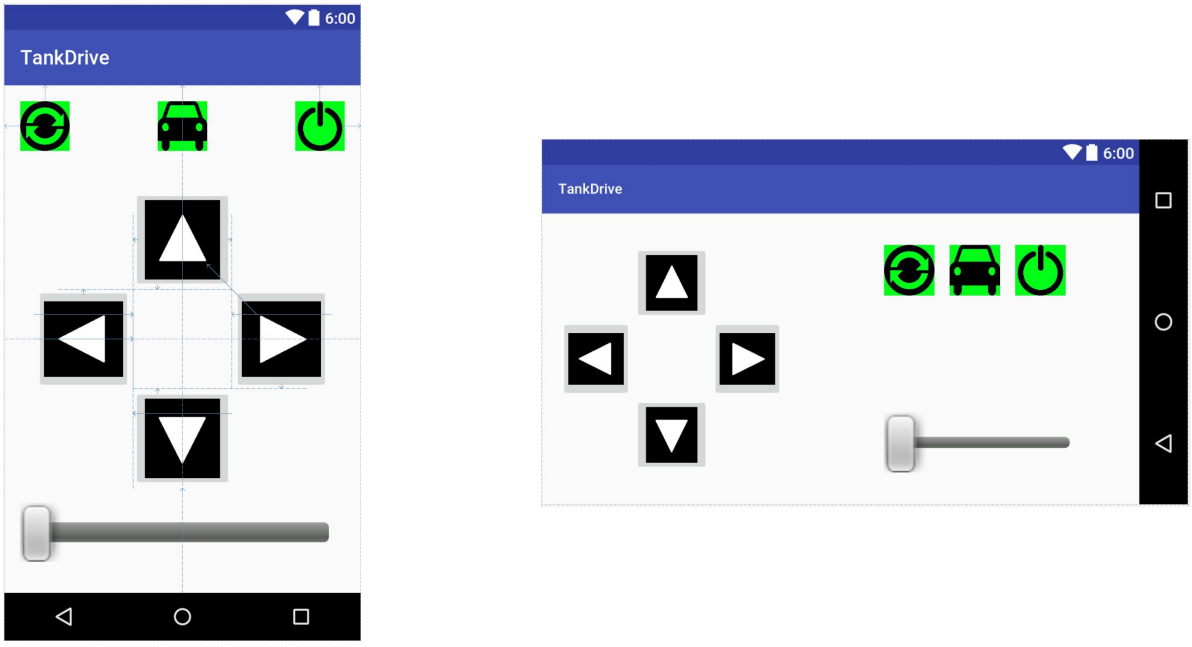
Bluetooth Connection

When the user first opens the phone, the app will prompt the user to enable bluetooth if it isn't already enabled. After enabling bluetooth, the phone will attempt to connect to the tank's module up to 4 times until the connection is established (the retry mechanism was included because of Reilly's ancient android phone's inconsistent bluetooth adapter; Daniel's much better phone (Samsung Galaxy S5) usually connects on the first try). After this point, the user can start to communicate with tank using the UI.

The connection is maintained through the ConnectionThread object, which follows the singleton design pattern, meaning that only one connection will exist at any time and this connection will remain outside of most of the lifecycle events of the app. We needed this because Android tends to destroy objects between switching activities and screen orientations, and we wanted the same connection to be visible to all activities. The phone creates this connection by looking through its list of paired devices and finding the tank, which it will then attempt to connect to in the ConnectionThread. The phone acts as a client of the bluetooth module on the tank rather than acting as a server.

Control Scheme

After the bluetooth connection has been established, the user can then send commands to the tank. The tank will not act on any commands until the power button has been pressed in the top right corner of the app. Once this button has been pressed, the tank will spawn processes that respond to user input. The directional arrows in the middle of the screen tell the tank to move forward and backward, or to turn left or right. The slider along the bottom tells the tank how fast to move; the further to the right the slider goes, the faster the tank will move. The button with a picture of a car on it above the directional controls will put the tank into auto-drive mode. To the left of auto-drive is a button that will put the tank into tilt-driven-steering mode.



Figures 3a and 3b: UI layouts in portrait and landscape

Driving the Tank

Each of the four directional buttons send 4 bytes of data to the tank when pressed, the first two of which represent the motion of the motors. The last 2 bytes are included as a termination sequence for the tank. When the motors are released, 2 0x00 bytes are

sent, which signals the tank to stop moving. If multiple direction buttons are pressed at once, the button that was pressed last will determine the action of the tank.

The speed of the tank is determined by the duty as a percentage of the period of the PWMs. This value is determined by the slider bar, and each button reads the progress of the slider bar on press and factors it into its calculation for duty. This speed is only used when the user is controlling the tank manually; autodrive has a set speed.

Auto-drive

When the user presses the button with a picture of a car on it, the tank will enter auto-drive mode until the button is pressed again. Once in this mode, the user cannot send input to the tank and the tank will follow routines contained in the system itself. Once auto-drive is cancelled by pressing the button once again, manual control can resume.

Tilt-Driven Steering

When the button in the top left is pressed, the user enters tilt-driven steering mode. In this mode, when the screen is rotated more than 25 degrees clockwise, the phone will turn right, and when the screen is rotated more than 25 degrees counterclockwise, the phone will turn left. This is determined relative to the phone's landscape orientation.

In order to use tilt-driven steering, the phone must have both an accelerometer and a geomagnetic sensor. The rotation of the phone is determined by the SettingsManager object after retrieving data from these 2 sensors. Behavior is undefined for phones that do not have both of these sensors.

Given more time, we would have fleshed out the functionality of tilt-driven steering mode. As it stands, the tank can rotate based on the orientation of the screen, but it can't be driven from this view; the directional buttons do nothing when pressed.

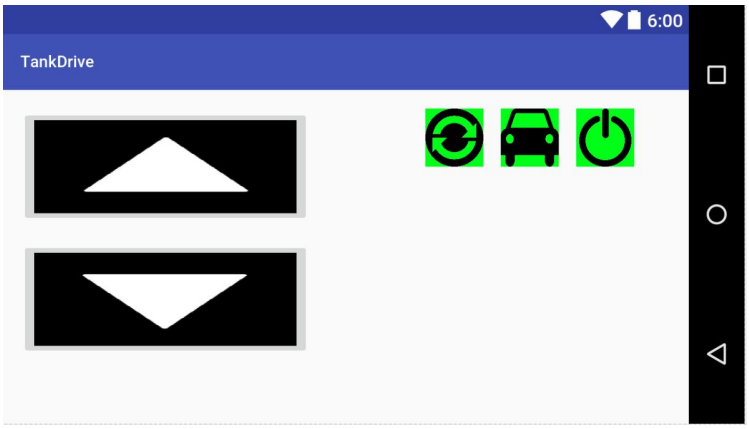


Figure 4: UI layout for tilt-driven steering mode

Conclusion

This final project was both a culmination of the skill we've gained from the previous 4 labs and a tremendous challenge as we delved into topics our group had little experience in. Before this project, none of us had any skill with Android development or Bluetooth communication, so the first few days of work were daunting. Learning an entirely new framework and a somewhat convoluted protocol in a matter of days was no small feat, but we managed to create a successful final product and learned along the way. We chose to not utilize any APIs other than the default Android APIs in order to learn as much as we could about the low-level work that needs to go into creating an end-to-end system. In the end, we may not have had the flashiest project, but we learned much more than if we had simply outsourced the more challenging parts (e.g. bluetooth) to existing libraries. We found that our decision to dive deep into the complicated details of embedded systems left us feeling more confident in our abilities as system engineers.