Interrupts and Real Time

EE 474 Introduction to Embedded Systems
Lab 4

Daniel Park
Samuel Johnson
Reilly Mulligan

Introduction

The purpose of this lab was to use the Beaglebone with the H-bridge and distance sensors to manipulate the two motors beneath the Beaglebone. To maneuver the vehicle in relation to its surroundings, interrupts were used to control the movement of the drive frame. These interrupts were controlled by a distance sensor located at the front of the vehicle. The vehicle is able to boot itself, with the ability to turn on and off multiple times via a switch located on the top of the vehicle.

Hardware Connections

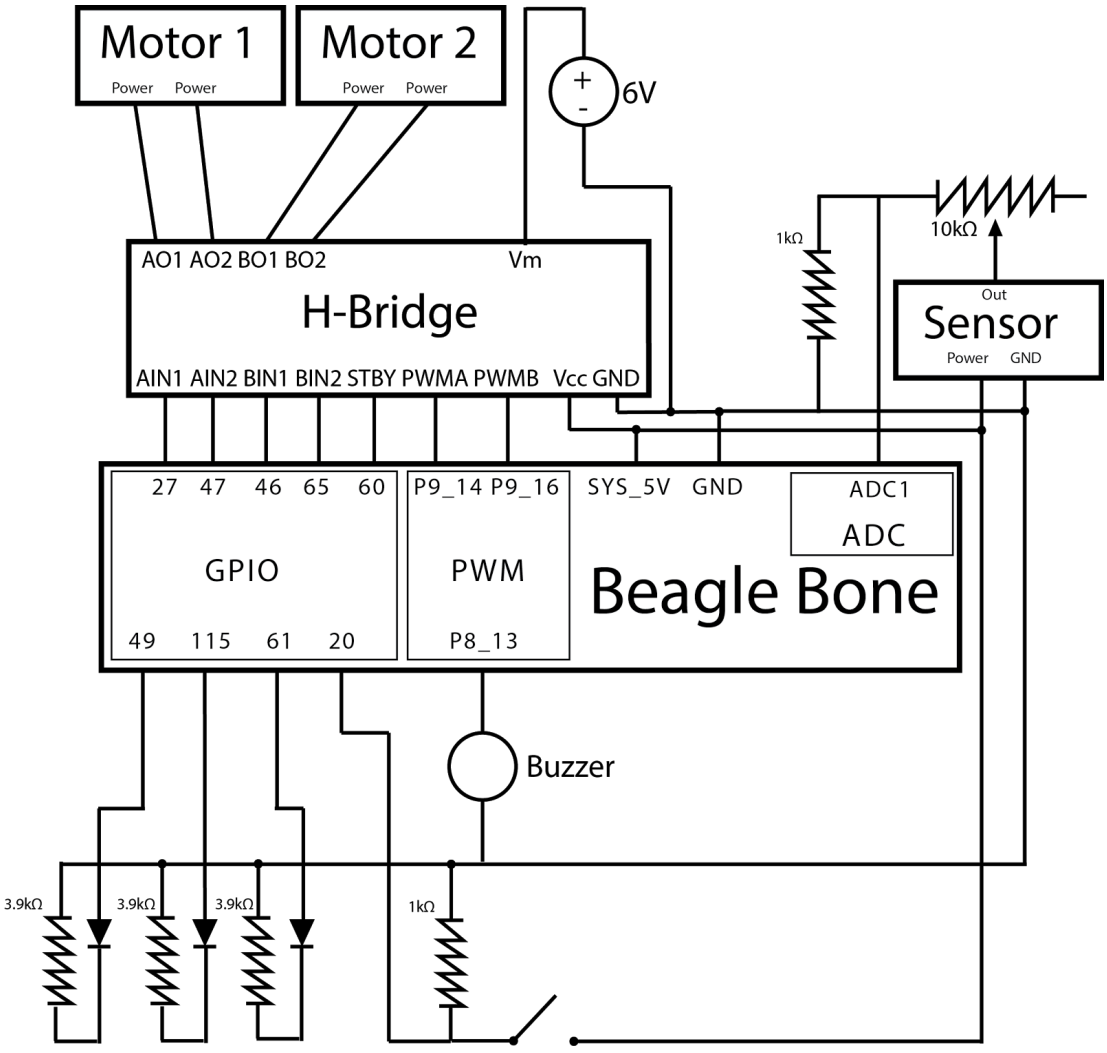

Figure 1: Hardware connections on the Beagle Bone.

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, the ADC on the Beaglebone is attached to the output of the distance sensor so that the distance to an object in front 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 are several GPIO pins and one PWM pin that react to different events in the processes on the Beaglebone. There are three LEDs hooked to GPIO pins. One LED serves the important function of notifying the user when the Beaglebone has successfully booted and is ready to be switched on to drive itself. The other two red LEDs are used as brake lights to signal when the vehicle is backing up (so that accidents don't happen - bonus points for safety?). Additionally, the PWM is hooked to a buzzer so that the vehicle makes a noise when backing up (again, for safety purposes).

Finally, the switch is wired to the Beaglebone so that the driving function can be turned on and off once the vehicle has booted. This switch can be used an unlimited number of times to switch the vehicle on or off, once it has been booted.

Software Overview

Our software consists of three scripts that control the Beaglebone and its self-driving capabilities. The first script is called tank_entry.exe which is the process that initializes the self-driving capabilities of the vehicle. Tank.exe controls the driving functions of the tank, while adc_listener.exe controls the distance sensor and detects objects in front of the tank. These processes work in parallel, and communicate through the use of signals to drive the vehicle. The specifics of the three scripts will be discussed more extensively in the following sections.

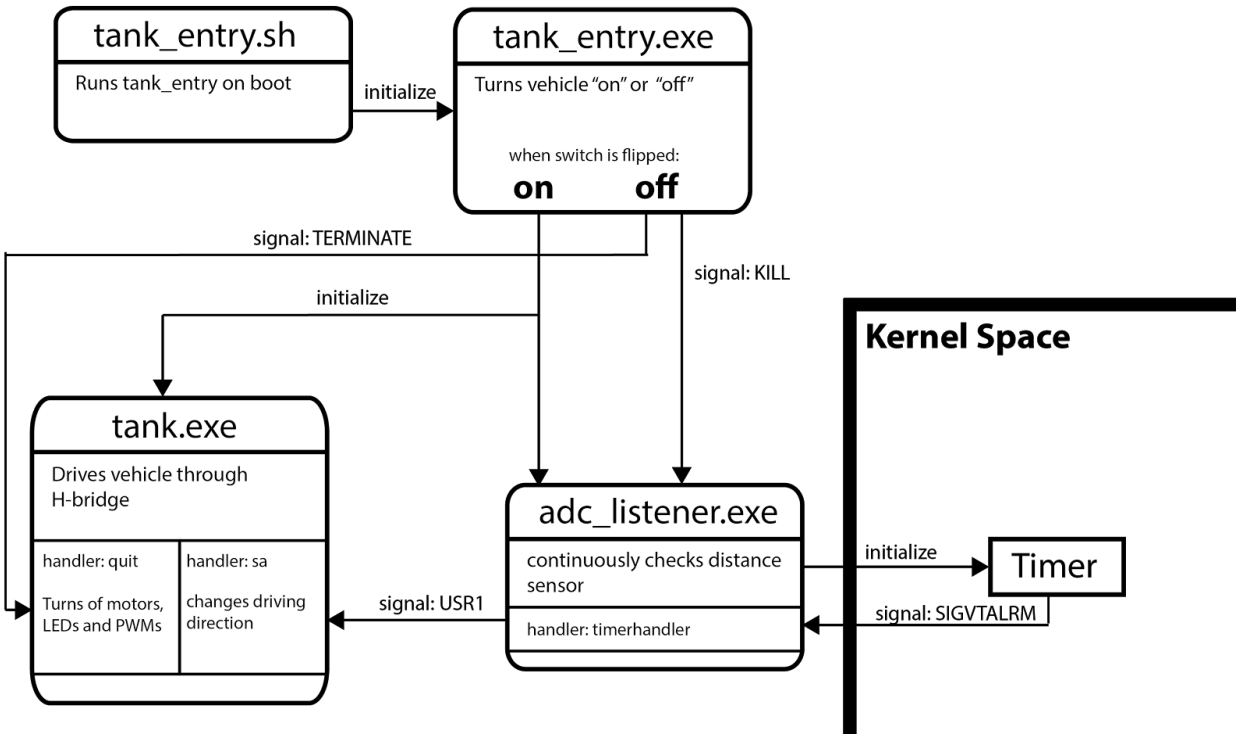

Figure 2: Diagram showing software interactions

tank_entry.exe

This is the master script that initializes the vehicle to self-drive mode and runs tank.exe and adc.listener.exe. The script takes an active-high switch as input to control whether the vehicle is in self-drive mode or not. When the switch is on, the script creates two child processes, one for running tank.exe and one for running acd_listener.exe. The process will then continue to poll the value of the switch. If the switch is ever turned off, 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, buzzers and LEDs until the switch is turned on 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 "master 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. This script will continue the drive the vehicle forward until it receives an interrupt signal from adc_listener, telling it that it is getting too close to an object in the front. Once this signal is received, the handler function will be executed by tank to drive the vehicle backward, and turn to avoid bumping into the object that is too close. During this interrupt process, the buzzer and Red LEDs will also be turned on to notify surrounding vehicles of the safety hazard. After the interrupt, the tank will continue to drive the vehicle forward until another interrupt is passed.

Tank also has a second 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 distance sensor. 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 the distance sensor and stores it. Then, to cut out noise, 20 samples are averaged together to obtain one value every 10th of a second. If this average value indicates a distance that is too close to the sensor, adc_listener will send an interrupt signal to tank that will change the driving behaviour of the vehicle. Once this signal is sent, adc_listener will sleep for a specified amount of time in order to avoid sending any unnecessary interrupts to tank.

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.

Conclusion

Ultimately, by completing this lab, we learned about interprocess communication, peripherals that use the ADC pin and how to utilize them, and about how to use the H-bridge to control a drive base. With a lot of time and effort, we were able to make a fully-functioning system comprised of 3 concurrently-running processes.

The biggest difficulties that we ran into had to do with timing; depending on the order of certain operations, we would get different outcomes, and due to the nondeterministic nature of parallel processes, we ran into problems. By making sure that certain commands happened in the right order, we were able to fix these.

Moving forward we can use the knowledge we gained to make even more elaborate systems, with a multitude of processes and peripherals.