Lab 4: Interrupts and Realtime using H-Bridge and DC Motors

Irina Tolkova # 1240609
Samuel Scherer # 1435365
Tremaine Ng # 1433161

Overview

In this lab we implemented movement using an h-bridge connected to DC motors and a distance sensor system that utilized timer-based interrupts. In addition, we used a systemd based startup script to make the board run our code automatically after power is connected. To use these functionalities, we also created a search and move program. Timer interrupts fed into the main code, reading their current values and allowing the main receiver code to make decisions from that information.

Hardware

H-Bridge

The h-bridge served as a way to control the motor's external power supply using the board's output pins. The h-bridge's logic inputs set the motion to one of 6 modes: Free, forward, reverse, brake, left, and right. The pin outputs were given by setting the GPIOs to the corresponding bits of an integer value. The following numbers are the movement codes for the device.

6: forwards
9: backwards
10: Turn left
5: Turn right

SystemD Startup Script

We directed the OS to run a script in a location on startup using a .service file located in etc/systemd/system as well as in lib/systemd/system. We did this by creating a symbolic link between the two locations using bash command ln. startupTest.service, contained the information below.

```
[Unit]
After=mysql.service

[Service]
ExecStart=/usr/bin/startupTest.sh

[Install]
WantedBy=default.target
```

We then placed the desired startup script named startupTest.sh (as specified in service parameter) into the specified location (/usr/bin/startupTest.sh). We also had to make the script executable using “chmod +x startupTest.sh”. This script compiled our two executable programs, ran the listener, then ran the interrupt timer with the process ID passed as an argument.

Next we had to enable the service we created using these commands
systemctl daemon-reload
systemctl enable startupTest.service

To test the service without rebooting we used “systemctl start startupTest.service” and “systemctl stop startupTest.service”.

Timer and Interrupts

The timer and interrupts generated by that timer are generated in simple_timer.c and received by simple_listener.c. The timer program calls a function timer_Init that runs a timer 100 times per second. Each time the timer finishes it calls the function timer_handler which sends a signal to PID which is defined to be 22.

The program simple_listener.c has function sig_handler which is called each time a signal, SIG_USER, defined as 22 is received. Inside the sig_handler function the proximity sensors are read and certain averages are calculated. Each 10 samples are averaged, giving an effective sampling rate of 10 samples per second from each sensor, but much less noisy samples than without the averaging.

Search and Move Program

To utilize the sensors and motors, we designed a program that would move in front of the nearest object it finds. It is designed as a finite state machine with the states SEARCH, SPIN, GO, and DONE. The sensors are constantly reading distance values 100 times per second, averaging every 10 samples. The car begins in SEARCH, where it spins for 5 seconds and remembers the closest object it sees (high sensor input). It then enters SPIN, where it continues to rotate until the sensor sees an object as close as or closer than that object (with some tolerance). It will then enter GO, moving forwards until it sees that it is about to collide with the object (sensor input > 950). The car then enters the DONE state, stopping and terminating the program.

On every interrupt, the files holding the analog in readings are opened, scanned into a char pointer, saved, then closed. A global counter variable increments every time the interrupt is called. Every 10 counts the average value of those counts are compared to a maxValue global variable (initialized to 0, so the value is always replaced on first iteration).

Discussion

Various improvements and expansions could be made to this system. The program that we chose to make may not have been the best use for the sensors we had. Our sensors were best suited for close object detection, e.g. for collision avoidance. We instead chose to use them for sweeping the area for objects but wasn't able to see very far reliably. Better sensors would improve our program's functionality significantly.

Adding the ability to physically switch off our program would also help. We had to constantly unplug our board to make sure the previous running version of the program was off and wouldn't affect our tests in some way. Adding a button-based interrupt that terminates the program would remove the need for constant restarts.

A PWM-h-bridge input would allow us to adjust the speed of the motors. We did not find this functionality useful since the motors usually struggled to move at all, and used the constant outputs from GPIOs instead.

Conclusion

Interrupts give us access to real-time operation, allowing us to count on a program to execute on time and meet its deadlines. In this case, the program was a timer that ensured our sensors would feed values in 100 times every second. This avoids the scheduler problems that would crop up from running the sensor code normally. If another program occupied the portion of time when the car was about to hit an object, the sensors wouldn't respond until it was too late.