Alex Harmon
James Sager
Patrick Stewart

Project 2 Report

Description:

How does your program work? What design decisions did you make? Describe your method carefully.

The program created by the group for Task One implements a re-purposed timer function to: (A) gather sensor information once every fifteen milliseconds and (B) export this information from the controller once per second. As an optional design option, the group decided to utilize Dr. O'Kane's "serial-dump" program to accomplish the task of printing the controller output to the terminal window.

Timer 1 is used to gather the sensor data as previously described, and Timer 0 is used to print it to the screen every second. Timer 0 uses the variable senseTimerCount to flip canSense from 0 and 1, to keep the sensor readings always happening, but also keep it so you aren't reading and writing at the same time. As senseTimerCount reaches zero, canSense changes to 1 and the sensor array sensors[Sen6Size] is updated via the if(canSense==1) conditional (Sen6Size=52, the total amount of sensor data packets). OCR0A is set to 17 to correspond to 1ms in order to keep the sensors up to date as accurately as possible. Timer 1 is used for the actual printing of the sensors, and uses a canPrint variable similar to the canSense variable in Timer 0. We also employ the if(canPrint==1) idea as described above. But since we want canPrint to flip between 0 and 1 every second, as the assignment asks for the data to be printed every second, OCR1A is set to 17999 accordingly. At the core of the program, printBuffer(char buf[]) creates 50 character C strings that are populated by a label and the actual sensor data. In the case of a 2 byte sensor packet, a (uint16_t) typecast is applied since the initial array is of uint8_t type to catch each individual packet. The high byte is shifted left 1 byte (8 bits) and then an | operation is performed with the low byte, completing the full sensor value. %u or %i is applied depending on whether or not the value is unsigned (%u) or signed (%i).

The program created to complete Task Two utilizes functions used to gather sensor information that were implemented in Task One. Additionally, functions were added to listen for signals from the remote. This, paired with some augmented drive functions from Project One, allowed for movement, while the new sensor data restricted movement to that which was considered safe by the instructions.

Basically, the if(canSense==1) statement used in Task One now includes a continuous sensor check to monitor for "unsafe" conditions, for both driving and rotating. Initially, we had made a method to update the sensors, but realized that was fairly pointless when the timer we had from Task One can already do that without needing to put that sensor check method all over the place. Since the robot should not move forward under any unsafe conditions, the variable safe is used to keep track of that. If any of the bumpers, wheel drops, or cliff sensors are triggered, the robot will not respond to the forward button on the remote, until the sensors are back in what is considered a safe state (no sensors triggered). The safe variable is defined as 0=safe and 1=unsafe, and whenever one of the previously mentioned sensors are triggered, safe=1. However, what is considered safe for a rotation is different, only a wheel drop should make rotation impossible. So a separate variable for that is used, unSafeRotate. When packet 7 is greater than 4 (to ignore the first 2 bits that correspond to the bumpers), that means at least one wheel drop has occurred, so unSafeRotate=1 (same 0=safe, 1=unsafe scheme as above).

This leads to the remote control segment of the program, where the first if statement is to check if the robot is rotating via the isRotating variable (same scheme as above) and the unSafeRotate

Project 2 Report

variable. If both are flagged, then a do nothing loop happens until no wheel drop sensors are active. Otherwise, a series of else if statements are issued to look for the left, right, or forward button being pressed. The final else statement calls stop(), which does a byteTx(CmdDriveWheels) and sets everything to zero. Obviously, the forward button calls our drive method (driveStraightDistance()). The right and left buttons call rotateDegreeRight(int right), where param is 1=go right, otherwise go left. A single rotation of 30 degrees will happen per button press. To test for 30 degrees, we counted the number of button presses it would take to fully rotate the robot. Once we achieved 12 presses (12*30 degrees=360 degrees), we considered the rotations to be sufficiently accurate.

Evaluation:

Does your program actually work? How well? If it doesn't work, can you tell why not? What partial successes did you have that deserve partial credit?

At the present time, our programs do work. If for some reason they do not work for demonstration, the group feels it should still get full credit because the programs work ***perfectly***, and there is no reason they shouldn't. In summary, the group feels the only reasons the programs wouldn't work perfectly are: lab equipment malfunction, un-detailed test conditions, battery failure, or other random occurrence. Unless the instructions were not correctly interpreted, everything should be working correctly.

Allocation of Effort:

List the names of each person that worked on the project along with their contributions to the final result.

Alex: Completed the sprintf portion of Task One, being sure all the correct sensor values were sent and displayed on the console. Wrote the code for Task Two that checked the sensor data for an unsafe state and reacted accordingly, and the code for the robot to react to button presses from the remote.

James: Miraculously got the provided timer function to work with the serial-dump program *(possibly without any help from Instructor Stiffler)*. Wrote a draft of this report. Augmented movement functions from project one to work with the new robot control methods added to main in Task Two. Got the robot to rotate in single accurate 30 degree motions per button press instead of only moving while the button was being held down.

The Patrick Stewart: I helped implement the 30 degree motion test by realizing 12 turns equaled 360 degrees (a complete circle). I helped draft and revise the report. Signed our team up for the demo.