FPGA Car Control

Instructor: Prof. Kunle Olukotun, Chris Copeland(TA)
Students: Neel Parikh, Tian Zhao

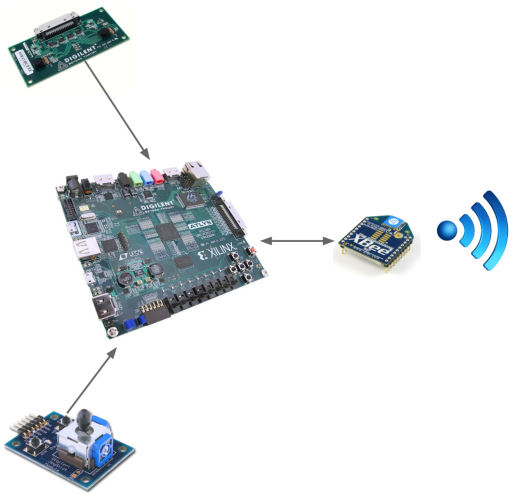
Project Overview:

We implemented a FPGA Controlled Vehicle using Digilent's Atlys board and a mobile platform from DFRobot. The system supports three modes: Joystick mode, command mode and visual control mode. User can switch between different modes using dip switches. In the joystick mode, we attach a joystick module to the Pmod connector and let the user control the car remotely by moving the joystick. In the command mode, the car moves along a predefined route that can be edited by the user. In the visual mode, the camera module (connected to the Atlys board via a Vmod connector) detects the position and the orientation of the car.

The Atlys board talks to the car via Xbee modules. We set up a bidirectional communication channel between the two so that FPGA can track the state of the car in real time. On the car, we use an arduino board to decode the commands from FPGA and to control each pair of motors. The direction of rotation is controlled by polarity of power supply and the magnitude of rotation is controlled by PWM (Pulse Width Modulation).

System Structure:

1. FPGA Side of System:



The Joystick module is a slave device that utilize SPI protocol to send data back to master device via IO buses. It updates readings at a very high frequency but we choose to use only part of data to ensure the speed of control.

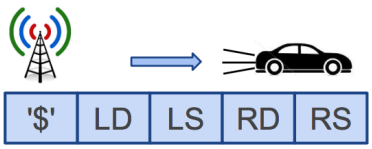
We only make use of one camera on the camera module. In order to reconstruct a whole

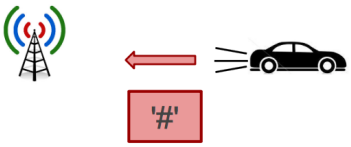
frame we use a FIFO to buffer rgb data and pixel information and send those to processor via DMA transfer.

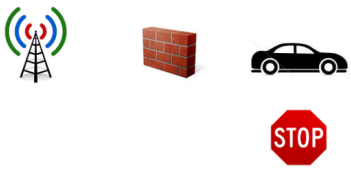
The FPGA board is connected with Xbee via a UART port and a computer. The UART port sends command that needs to be run on the car to a COM port via USB, and then we sync data received at that COM port with Xbee using VSPE (a serial ports emulator).

2. Transmission of Commands

Joystick Mode:

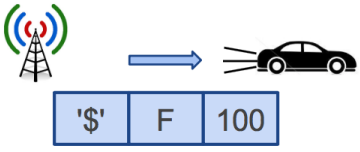


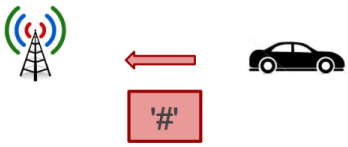




Each command has five fields: a header, direction for left motors, speed for left motors, direction for right motors, speed for right motors. A header is used so that the car can know if a valid command is received. After executing the command the car sends back a special command that tells the FPGA to issue a new command. When the car receives no command it pauses the motors (the system is still constantly polling information)

Command Mode

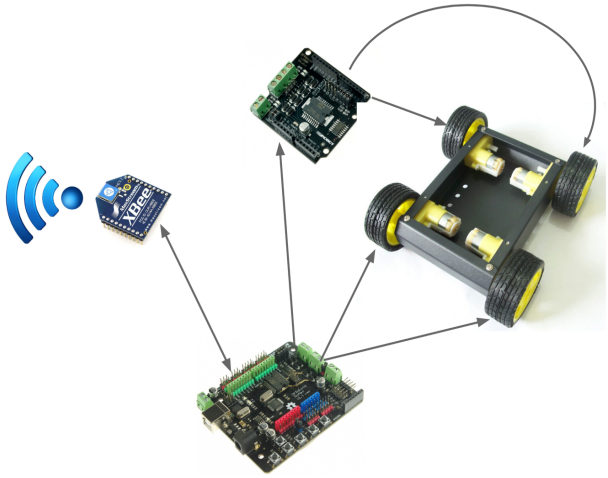




Each command has three fields: a header, direction, number of cycles that motor should turn. Similar to joystick mode, in command mode the arduino board decodes the command and

controls the motors using the direction and cycle information. After execution, the car sends back a termination character.

3. Vehicle side of system:



We use Arduino Romeo v1 to decode commands and to control motors. The servos on Arduino are extended by a motor shield to power all the four motors at the same time. Arduino talks to the Xbee module via a Xbee extension board.

Hardware/Software Implementation:

Peripherals:

Camera Stream Peripheral

We utilize the camera stream peripheral from Lab2.

Pmod Joystick Peripheral

We created our own peripheral using XPS. We found an ISE project from Digilent that demonstrates how to read data from the joystick module. Based on that, we modified the default VHDL files created by XPS to include the logic of reading joystick data and transferring data to processor via IO buses. We modified .pao file to include extra libraries to support the major logic. In addition, we changed the default .mpd file to include the Pmod ports into our system. In order to map all the physical pins on FPGA to our ports, we modified the .mhs file and the .ucf file. The .mhs file does two layers of translation from port names to on-board pins. .mhs file first translates a port name to a temporary pin name; it then maps the temporary pin name to an actual physical pin listed in the .ucf file.

User Logic:

Pmod Joystick Logic

Essentially we set up a state machine that handles data transfer using SPI protocol. We instantiate a SPI clock such that at every SPI clock cycle, the Master sends one bit data to the Slave (Pmod Joystick Module), and the Slave sends back one bit to the Master.

Control Interface

In the default user_logic.v file generated by XPS, we instantiate two registers to store and transfer the x,y values read from joystick. These data are stored linearly in the memory. We modified the top level vhdl file to connect the ports in joystick module and the IO buses.

Software Logic:

In software we set up four different modes: joystick mode, command mode, visual mode, and a standby mode. The standby mode is used to tell the car not to move around but to stay and wait for a command.

Every time a change in dip switches is detected, our software switches the system into one of the four modes and asks the car to decode the command in the way corresponding to different modes.

In Joystick mode, the software system keeps reading data from the two registers that store x,y values of joystick. It first look at the most significant bits of the x,y values and then adjust the direction and value of acceleration that the car should travel at (For example, if MSB of x is 1, then we know that the user pushes the joystick forward and the car accelerates proportionally to the value of lower bits of x . If MSB is a one, then we know that the joystick is pushed back; the car should slow down proportionally to the value read from lower bits of x). Then we encode this information and print the command to the terminal for transmission.

In command mode, we program a path for the car by using the SendCommand() function, which takes in a letter that maps to the direction and number of cycles that the motors should rotate.

In visual mode, we implemented a series of color filters (Adaptive filter and Spatial filter). The program first detects the positions of markers that we install on the car. After the first stage of filtering, our program then computes what the shape of the filtered result should be and then supplement the missing part.

Design Trade-offs and Problems Encountered

Building the Car and Markers; Configuring the Arduino Board with Other Modules

The car frame we got from DFRobot did not quite meet our requirements. It does not have necessary holes to host the standoffs needed for separating different platform. Also we added extra switches and battery packs to power the Arduino boards and motors. We had to go to the mechanic shop to drill holes on the platform and to build extra platforms to put on the switches.

Sending Data From FPGA to XBee

The shield for XBee uses a mini USB port and the UART port on FPGA board is micro USB. We initially planned to connect XBee directly to the UART port using a cable (Mini USB to Micro USB) built by ourselves, then we realized that the UART port is a non-standard USB port since it does not have power lines. Also we noticed that if we connect them directly, we would have to implement some functions at the end of XBee to initiate the USB transfer.

In order to power the UART port and to avoid implementing the USB protocol ourselves, we connected the UART port to a COM port on a computer and XBee to another COM port, and then used a COM port emulator to sync data between the two COM port. This way the computer automatically initiate USB transfer for us once the driver of COM port is installed, and it powers the UART port nicely.

Connecting Pmod Joystick Module to FPGA Board

Unfortunately we were not able to find a complete tutorial online about how to create a peripheral from scratch, and as a result we did a lot of work to figure out how to generate peripherals and drivers for softwares. We walked through almost every single function of XPS and studied how it generates files and modifies project. In the end we realized that .mpd, mhs and .pao files are crucial to generation of a peripheral:

.pao: Specifies all the libraries needed to support the user_logic.v files.
.mpd: Defines the name of ports, data width, type (Clock? Reset?)
.mhs: Gives the mapping from user defined ports to a physical pin name defined in .ucf

There are three tricky parts:
1. XPS ver13.4 is BUGGY. Although user can specify the type of port in the peripheral import wizard (Say, if we want to use a SPI clock we need to define it as a CLK in .mpd file), the wizard actually does not overwrite the type field in .mpd file. Therefore users

have to manually define the type of port in .mpd file.

2. The naming convention of ports / pins in XPS is overwhelming. It is not a good idea to manually modify .mhs file as it is tricky to follow the naming rules. Therefore using the peripheral import wizard to modify .mhs is always a good idea

3. There are two layers of translation in .mhs file. It first maps a user port to a temporary name and then maps it to a pin name specified in the .ucf file.

Detecting Faults and Recovering From Fault State

Our original version of car control does not use a header. As a result whenever our car drove out of range (The range of XBee is quite wide; however the complicated environment in the lab reduces the working range significantly) it falls into a fault state and cannot recover from that since the data in the Arduino buffer is no more aligned.

In order to solve that we add a header to every command sent so that the car can always pick up the right command by checking if the header is present. We choose a special character to be the header so that even the header is corrupted and is changed into a valid command, out program on the car still will not pick it up.