ESE 519 RoboSoccer

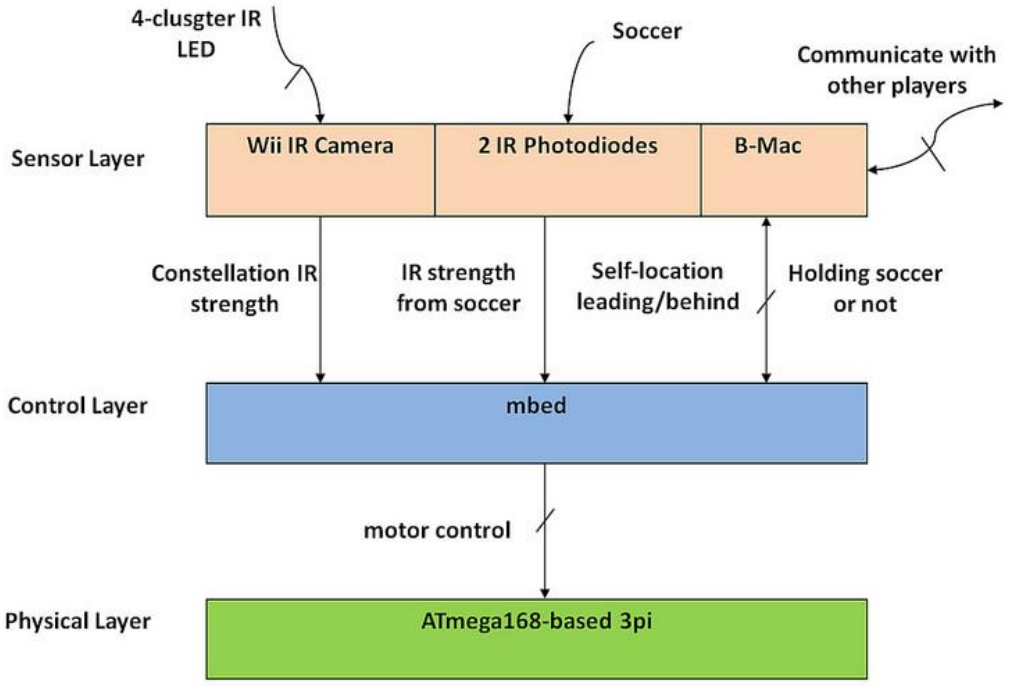
Final Report

Group #6

Yufei Ma, Zhu Cai, Su Gu

RoboSoccer Configuration

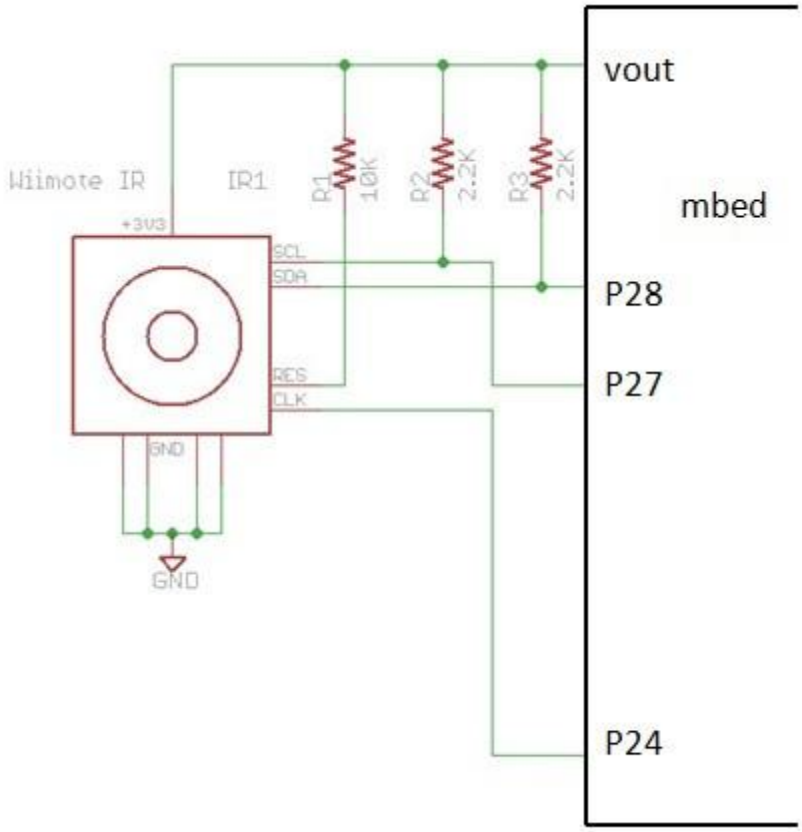
The hardware architecture of RoboSoccer is composed of the sensor layer, the control layer and the physical layer:



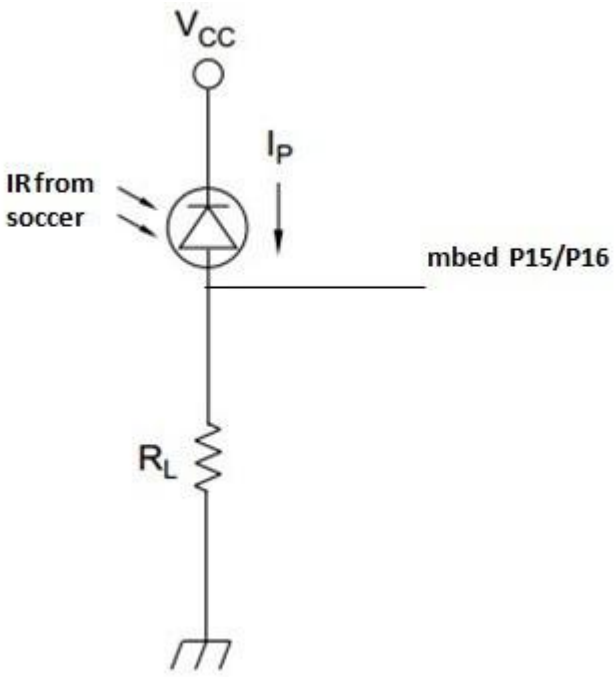
1. Sensor Layer

The sensor layer hardware configuration includes the Wii IR camera, the IR photodiodes and the B-Mac. The Wii IR camera is deployed to receive the 4-cluster IR streams from the ceiling constellation to self-localize each play. Each RoboSoccer also has two IR photodiodes (left and right) to track the the IR strength transmitted from the soccer ball, which essentially tells the robot where the relative position of the soccer. The B-Mac is used to communicate with other players as well as the goal controller. The communication

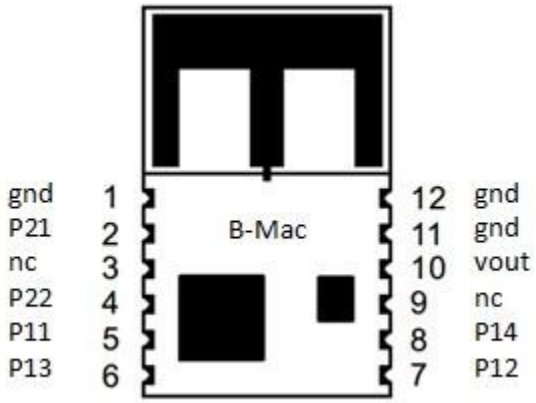
information includes: a) self-position of robot during the ball-passing situation b) current leading/behind of the team and c) who is holding the soccer currently.



The IR photodiode circuit is shown as the following. As the resistance of photodiode changes with the IR strength, the regulated output voltage also changes and is converted into digital inputs to mbed. The mbed then compares the left (P15) and right (P16) values to track the soccer ball.



The B-Mac is interfaced with mbed to implement communication among the robots as well as with the goal controller.



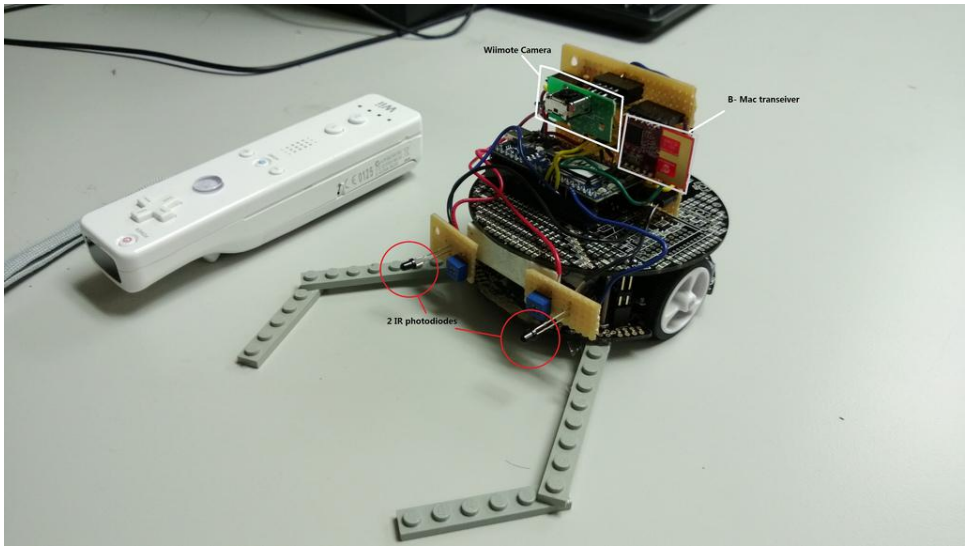
2. Control Layer

The m3pi consists of an mbed microcontroller running the high level functions and control algorithms, implemented by API calls to 3pi hardware. The mbed receives the 4-channel IR strength data from Wii IR camera and computes the self-location. It uses the information of current game (leading/behind, holding the soccer or not and self-position) to make the movement decision and drives left and right motors.

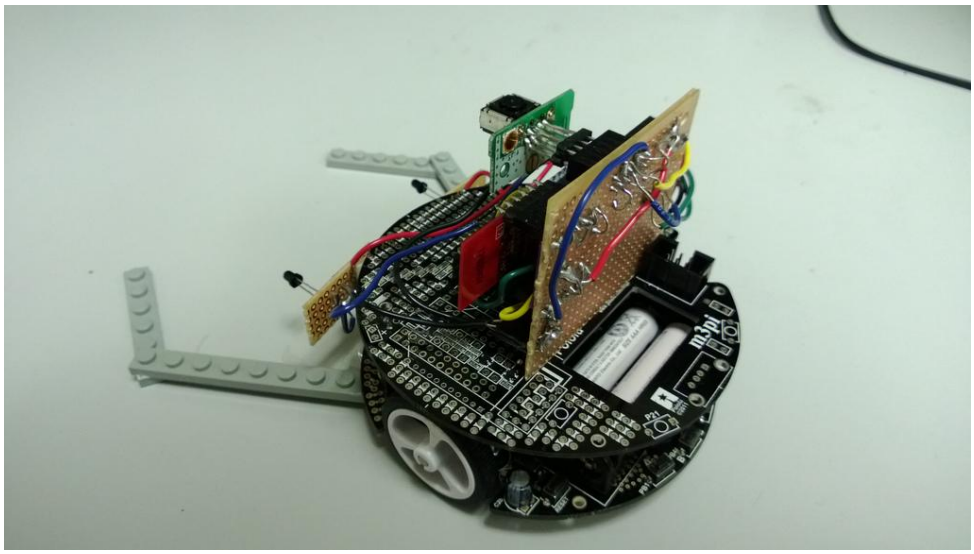
3. Physical Layer

The ATmega 168 microcontroller of 3pi robot is controlled by mbed to drive the two motors. The motors are driven in a differential style, which means the robot makes a turning by running the two motors at different PWM duty cycles. A spin is achieved by driving one motor forward and the other one backward.

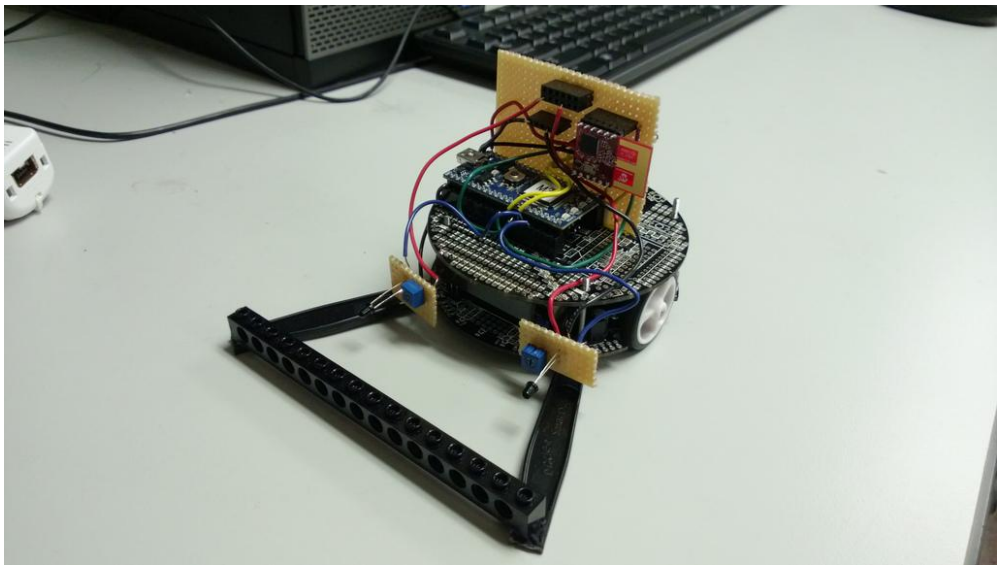
Pololu 3pi robot with mbed, wiimote camera, B-MAC antenna, and IR photodiodes integrated.



Back of gray team player.

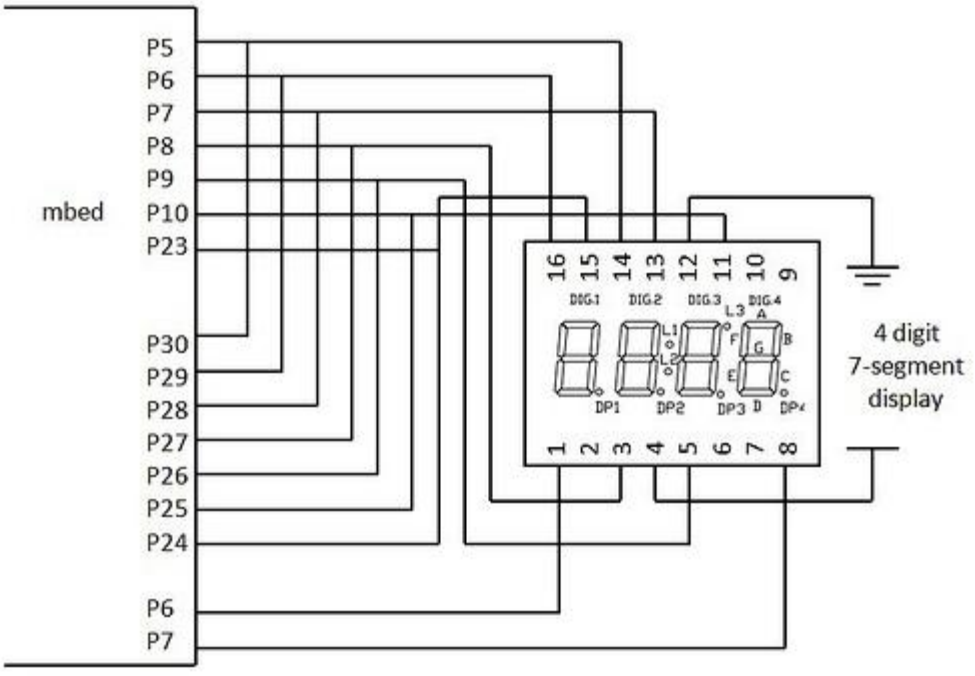


Black team goalkeeper.

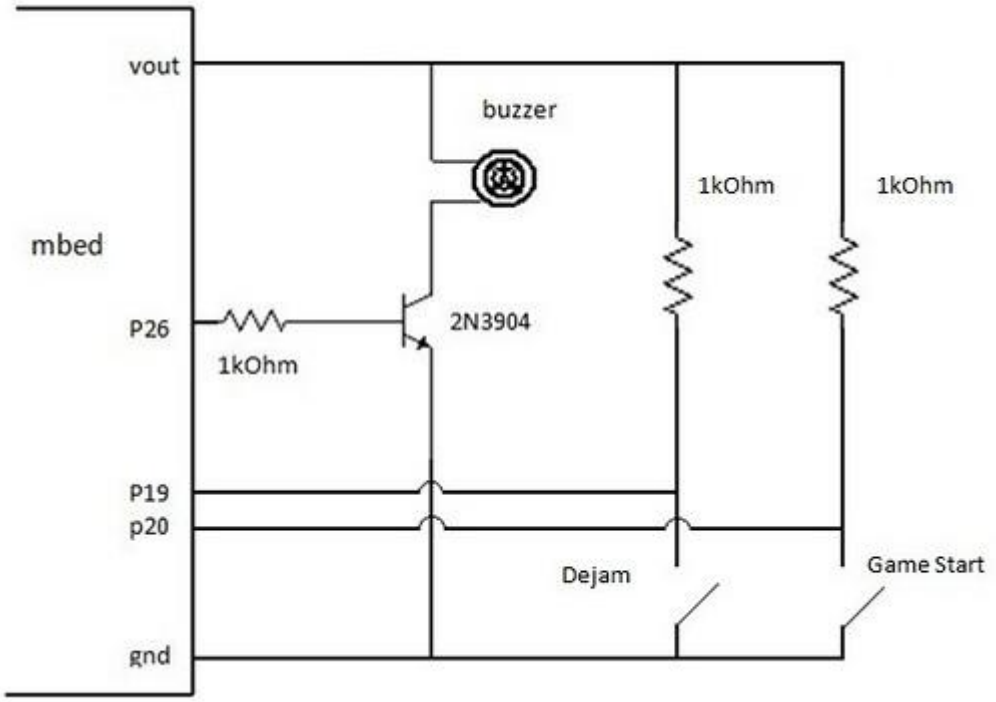


Goal Controller

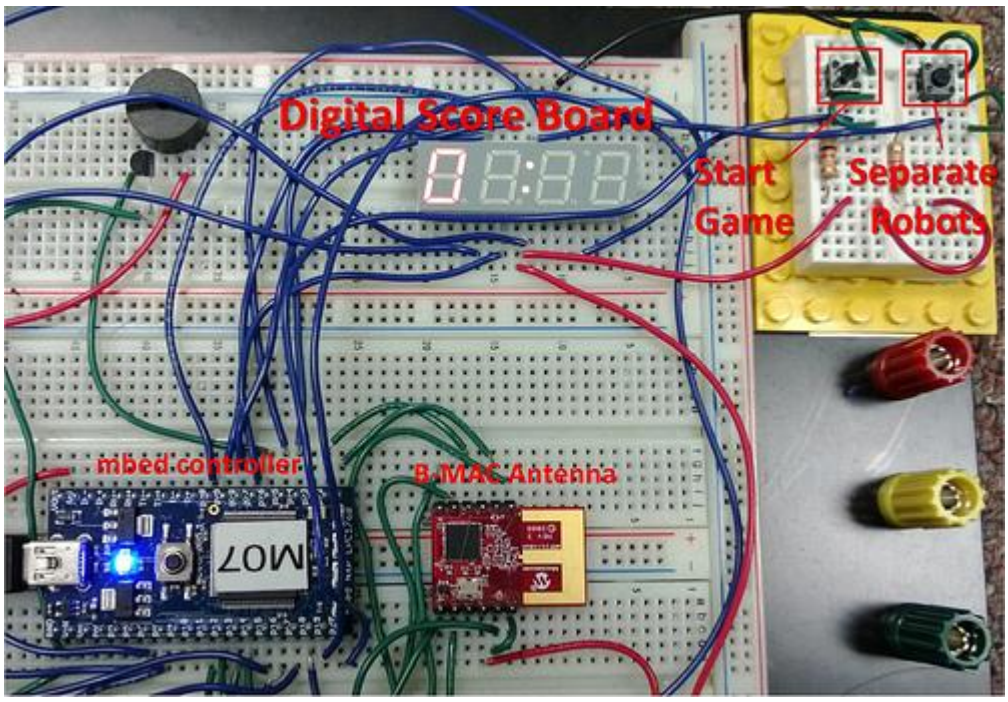
The goal controller serves as the referee in the game. It keeps track of the score and display it on a 7-segment LED scoreboard and it also plays a melody indicating that a goal is made. In addition, it sends the "game start" signal to the players at the beginning of each turn. When the players are jammed with each other, the goal controller also sends the "dejam" signal to resolve the jamming. The 7-segment scoreboard configuration is shown as the following schematic. Two digits are used to represent the score of the game. To save the mbed pin, segments of the 2 digits share 1 pin on the mbed. At any time during the game, only one digit is selected by the control output of mbed (P6/P7) so that we can update the scoreboard dynamically.



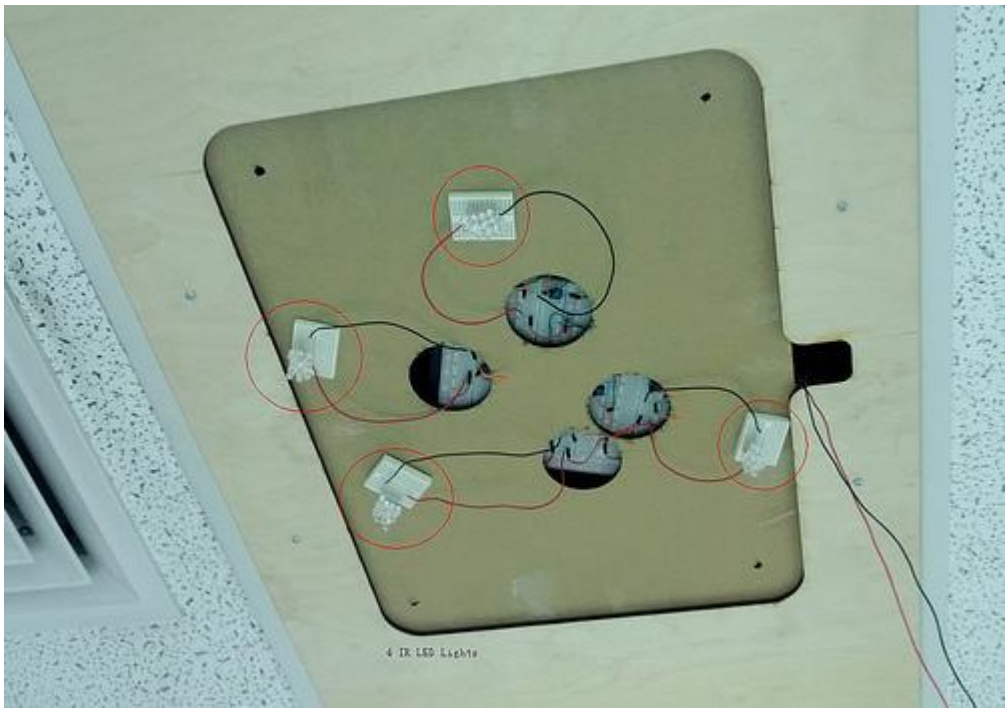
A piezo buzzer is added to display a piece of cheering melody to remind the audience that a goal is made and that the scoreboard is updated at the same time. The buzzer circuit is shown as the following. P26 from the mbed is used as the PWM output to generate different music scale.



The Digital Score Board and Control Part (Start Game & Dejam).

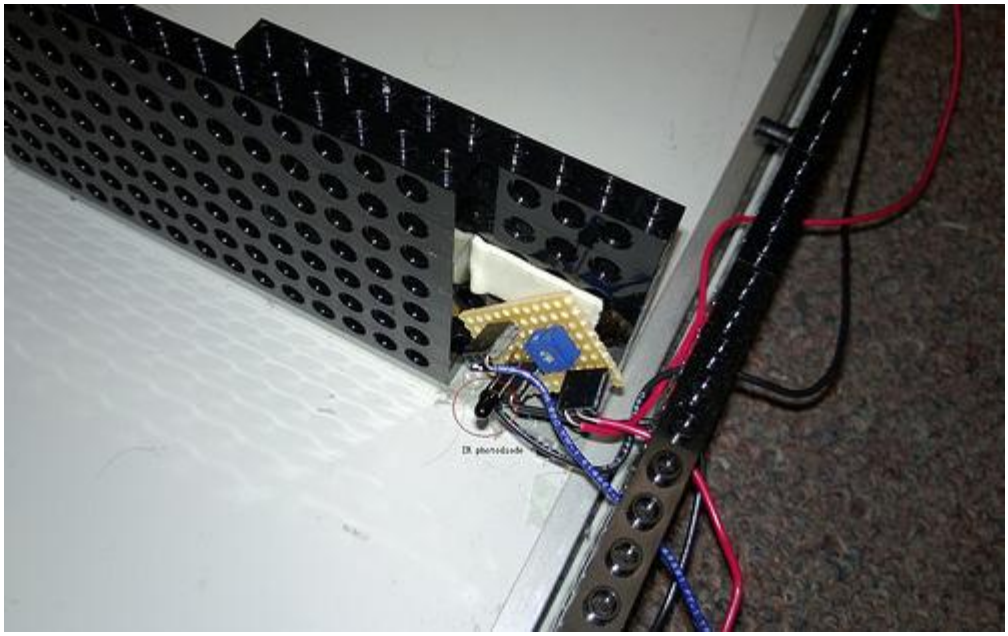


The 4-cluster IR constellation LED is shown as the following.



Ball with 5mm IR LEDs emits IR light supplied by 9V battery.

Goal with IR photodiode senses the ball and scores.



Software Description

This part describes the software implementation along with the attack/defense strategy. The description will be elaborated as sub parts describing the features of the whole software.

Software Platform: Keil uvision4
Hardware Platform: mbed

1. API Based Implementation

As you can see from the hardware part, each robot has been built as an integrated system combining different functional parts. Our first job is to enable each part and make them collaborate successfully. So we implement different API for each part:

1) wiicamera.h: contains function “read_camera(blob)” which returns the coordinates of 4 ir led clusters. This API requires PWMout as clock and I2C connections. Pay attention to the pins which you want to assign, maybe you need to modify uart_library if these pins have been previously occupied.

2) position.h: contains function “track(blob)” which inputs blob coordinates and returns robots global coordinates and orientations. The algorithm could be generally described as finding the longest line from the clusters and calculate the angel between the local frame and global frame. In the end, scale the coordinates by the ratio of global frame to cluster frame.

3) goto.h: contains “approach_net_()” which is crucial in finding the angel between robots current position and another fixed point. This function is used whenever you want the robots to go to a fixed point, say the goal or home.

4) bmac.h: responsible for transmitting and receiving packet.

5) adc_driver.h & adc_driver.cpp: responsible for reads the adc data from the photodiodes. set_adc_chan() and then get_adc_val() enables reading adc data from any of the 6 channels.

6) m3pi.h: crucial API for controlling 3pi bots by mbed. Require I2C connections so pay attention to any previously occupied pins.

2. Task-based Software Framework

Our implementation in 3pi robots programming is highly task based and module based. The whole main.cpp is derived from former lab task basic_bmac. We use the basic framework and schedule the whole program into 2 tasks: rx_task() and tx_task().

Rx_task(): responsible for receiving packets and process different packet types.
Tx_task(): responsible for state machine routine and transmitting packet.

Functions of packets: a) collaborate with teammates. b) receive signals from referee so to start a game or dejam, also choosing strategy based on received score changes.

Challenges: Since we use B-Mac as our network protocol, we’ve been experiencing packets loss and lags. The more packets on fly, the more chance the robots miss a packet. We have 6 robots and a controller playing, the throughput of packets is huge and it’s very easy that robots miss some crucial packets. So we must do some optimization to packet tx/rx mechanisms.

Optimization of Tx/Rx mechanism: The main reason we see a loss of packet is that the original tx_task() would transmit a packet every time it invoked, regardless of whether we need to send a packet or not. And since we use state machine here, we have set the tx_task() period as 5ms, you can imagine the amount of redundant packets. So, to optimize the tx/rx mechanism, we’ve modified the tx_task(), we move all the transmitting codes into an module “transmit(int flag)”, and we call this function just when we need it. This modification has two merits: 1. largely reduces redundant packets. 2. Simplified the process of transmitting packets carrying different messages.

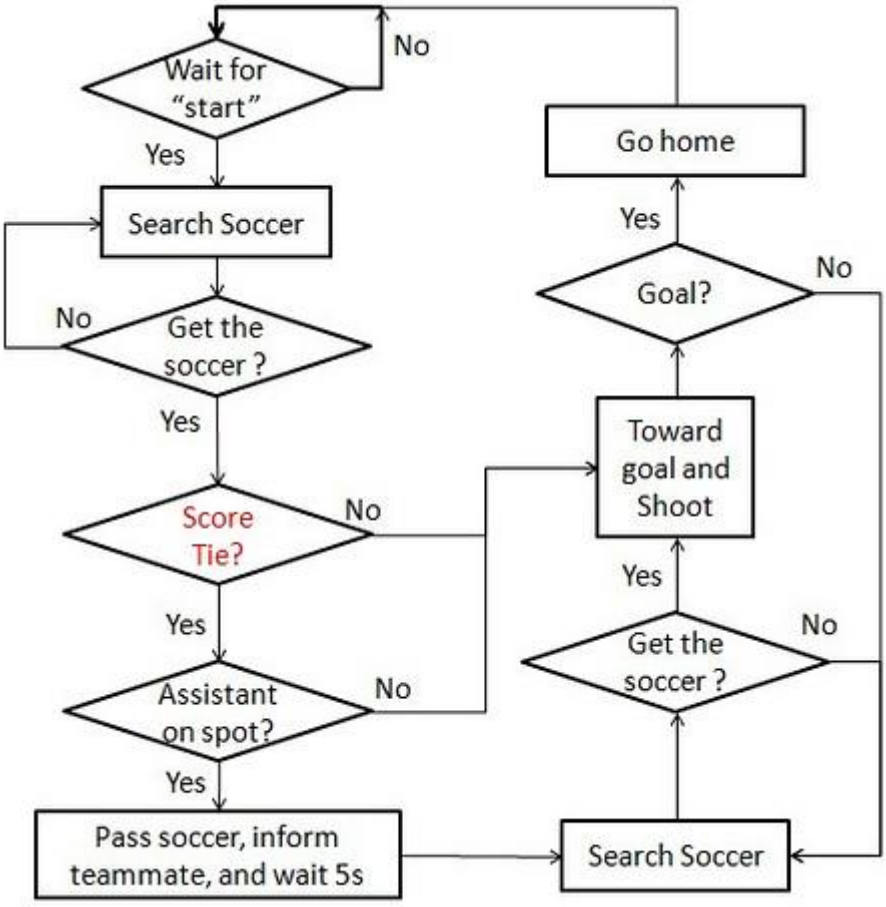
Minor considerations: Since it is of utmost importance that each robots receive the packet, we’ve set the priority of rx_task() higher than tx_task().

3. Module Based Robots Assignment

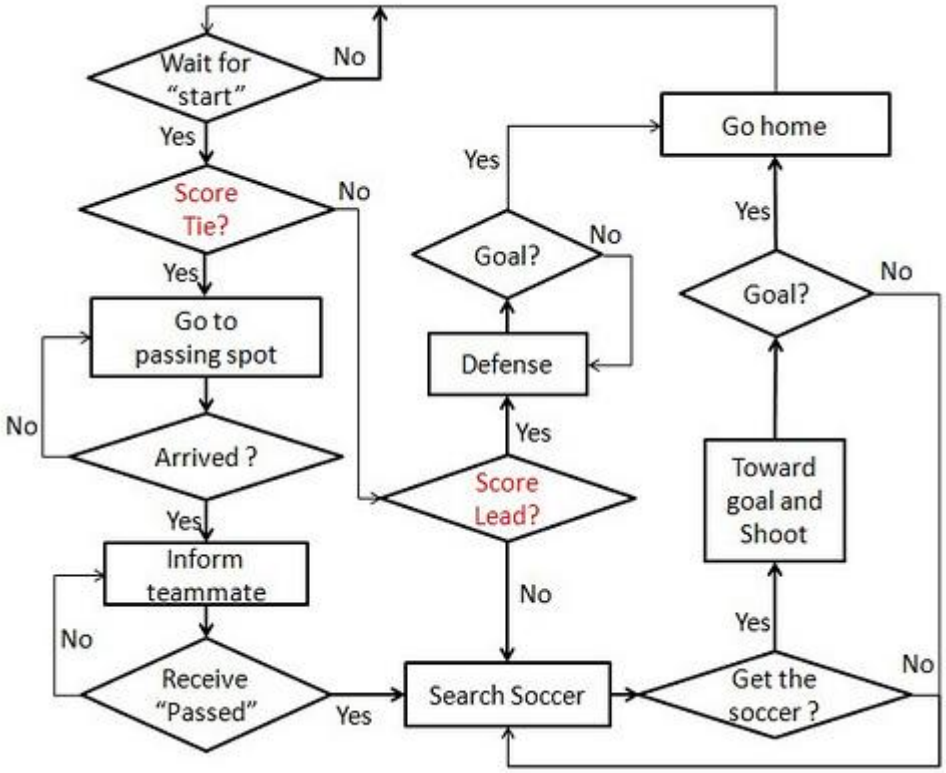
As we know that there are 6 robots here, and each robots have their own copy of coordinates and strategies. Do we need to write 6 different copies of programs so that each is unique to one robot? No, we don't. We want to do it with more feasibility. So we've done module programs with the 4 most important robots—robots which are attackers or defenders. All 4 robots can be simply programmed by the same project, "sniper". All we have to change is the id of the robot. 1 and 2 are for team A, 3 and 4 are for team B. The program can simply assign different strategies and important coordinates according to robot ids.

4. State Machine Based Strategy Routine

This is the core part of our program. The state machine is a comprehensive, close-looped state machine, so that the robot can play fully autonomous, without human involved.The simplest description of our robots can be like: start->play->goal->go home->start again. So the loop can go infinitely as long as batteries live.The state machine is different across different robots, one robot of each team serves as pure attacker and passer; the other robot of each team serves as assistant and defender. A simplified state machine for the former type is:



State machine for the latter type is:



This is a simplified state machine, it doesn't include attack/defense changes within one stage. Actually in our real strategy, when one robot get the ball, the other team are all going to defend; and when one robot finishes shooting (meaning the ball is free now), the other team is going to search the ball again.

To collaborate with rx_task(), the state is a global object, so it can be changed whenever a new packet arrives, telling the newest information. You can simply imagine the rx_task() as sort of real-time interrupts.

Locks Implementation: If you play with the robot, you can find it's important to make the state machine flow. If the state machine stuck at one point, it's going to ruin the match. But one problem is, there can be more than one packet telling the same thing. So do we need to change our state every time we receive this? No, that is going to stick our state machine. For example: every time the robot receives a "dejam" packet, the robot backoff. But what if the controller sends a thousand packets telling "dejam"? If the robot has no lock mechanism, it's going to backoff forever! So, we implement a lock every time the state changes by receiving a packet. There are three kind of locks:

1. actFlag: That's for general packets send by robots, whenever a state change happens, the actFlag is goint to block the state from changing by such a packet again.

2. Packet serial number: That's for "dejam" packet sent by controller, we don't want the robot to just backoff once and forever either. So we count the serial number of each received "dejam" packet, if the packet is newer, the robot is going to backoff, else, nothing.

3. Inited: that's for "start" packet sent by controller, we just want the packet once at the beginning of each stage. So when inited=0&&packet received, initialize and start the game, also set inited to 1. So no more trouble.

Latest Youtube Video Demo Link:

http://www.youtube.com/watch?v=IvAT3iL1Sts