5-3-2012

Low-Cost Autonomous Multi-Robot Network

Michael DeMange
Rodrigo Tamashiro
William Westrick

Follow this and additional works at: http://opus.ipfw.edu/etcs_seniorproj_engineering

Part of the Engineering Commons

Opus Citation

This Senior Design Project is brought to you for free and open access by the School of Engineering, Technology and Computer Science Design Projects at Opus: Research & Creativity at IPFW. It has been accepted for inclusion in Engineering Senior Design Projects by an authorized administrator of Opus: Research & Creativity at IPFW. For more information, please contact admin@lib.ipfw.edu.
Project Title: Low-Cost Autonomous Multi-Robot Network

Team Members: Michael DeMange
Rodrigo Tamashiro
William Westrick

Faculty Advisor: Yanfei Liu, Ph.D

Date: 5/3/2012
# TABLE OF CONTENTS

**TABLE OF FIGURES** ........................................................................................................................................ 7  
**ACKNOWLEDGEMENTS** ................................................................................................................................. 9  
**ABSTRACT** .................................................................................................................................................... 10  
**SECTION 1: DESIGN DESCRIPTION** ............................................................................................................... 11  
  1.1 REQUIREMENTS AND SPECIFICATIONS ............................................................................................. 12  
  1.2 GIVEN PARAMETERS .......................................................................................................................... 12  
    1.2.1 Hardware ................................................................................................................................... 12  
    1.2.2 Operating Standards .................................................................................................................. 12  
  1.3 DESIGN VARIABLES ............................................................................................................................ 12  
    1.3.1 Hardware ................................................................................................................................... 12  
    1.3.2 Software ..................................................................................................................................... 13  
    1.3.3 Operating Conditions ................................................................................................................. 13  
    1.3.4 Limitations and Constraints ....................................................................................................... 13  
    1.3.5 Safety, Environmental, and other Considerations ..................................................................... 13  
  1.4 ORIGINAL CONCEPTUAL DESIGN ....................................................................................................... 14  
    1.4.1 Overview Diagram ...................................................................................................................... 14  
    1.4.2 Functional Description Overview ............................................................................................... 15  
    1.4.3 Controller Module ...................................................................................................................... 15  
      1.4.3.1 Overview ................................................................................................................................. 15  
      1.4.3.2 Components - Interconnections ............................................................................................. 15  
      1.4.3.3 Functional Description ............................................................................................................ 16  
      1.4.3.4 Programming .......................................................................................................................... 16  
      1.4.3.5 Schematic ................................................................................................................................ 17  
    1.4.4 Locomotion Module ................................................................................................................... 17  
      1.4.4.1 Overview ................................................................................................................................. 17  
      1.4.4.2 Components - Interconnections ............................................................................................. 18  
      1.4.4.3 Functional Description ............................................................................................................ 18  
      1.4.4.4 Programming .......................................................................................................................... 18  
      1.4.4.5 Schematic ................................................................................................................................ 19  
    1.4.5 Power Module ............................................................................................................................ 19  
      1.4.5.1 Overview ................................................................................................................................. 19  
      1.4.5.2 Components - Interconnections ............................................................................................. 20  
      1.4.5.3 Functional Description ............................................................................................................ 20  
      1.4.5.4 Power Calculation ................................................................................................................... 21
TABLE OF FIGURES

Figure 1: Overall Design Diagram................................................................. 14
Figure 2: Schematic for the Controller Module............................................... 17
Figure 3: Schematic for the locomotion module............................................... 19
Table 1: Power requirements used to determine battery needs....................... 21
Figure 4: Schematic for the power module...................................................... 22
Figure 5: Schematic for the sensor/feedback module..................................... 24
Figure 6: 3D view of the overall design of the robot...................................... 25
Figure 7: The general overview of the design of a robot.................................. 26
Figure 8: Overall Design Diagram................................................................. 27
Figure 9: Schematic for the CPU Module....................................................... 30
Figure 10: Board Layout for the CPU module................................................ 31
Figure 11: Schematic for the locomotion module............................................ 33
Figure 12: Board layout for the locomotion module........................................ 34
Table 2: Power requirements used to determine battery needs...................... 36
Figure 13: Schematic for the power module.................................................... 37
Figure 14: Board layout for the power module............................................... 38
Figure 15: Schematic for the RangeFinder/Servo Module............................... 40
Figure 16: Diagram for the PSoC Connections of the RangeFinder/Servo Module 41
Figure 17: Board layout for the RangeFinder/Servo Module............................ 42
Figure 18: Schematic for the Sensor Module.................................................. 44
Figure 19: Diagram for the PSoC Connections of the Sensor Module.............. 45
Figure 20: Board layout for the Sensor Module.............................................. 46
Figure 21: Placement of motors and encoders................................................ 47
Figure 22: Battery Placement......................................................................... 48
Figure 23: Placement of the rangefinders....................................................... 49
Figure 24: Placement of the 3 amp fuse......................................................... 50
Figure 25: Placement of one circuit board....................................................... 51
Figure 26: Second Circuit Board Placement................................................... 52
Figure 27: Placement of the box that protects the circuit boards..................... 53
Figure 28: Placement of the Pan and Tilt servos.......................................... 54
Figure 29: Placement of the uCam visual sensor............................................ 55
Figure 30: Location for the LCD..................................................................... 56
Figure 31: Locations for the 2 RGB LEDs..................................................... 57
Figure 32: Wireless module placement......................................................... 58
Figure 33: Final Robot Design Picture............................................................ 59
Figure 34: Testing the camera servos on a breadboard................................... 61
Figure 35: 3 amp fuse module........................................................................ 62
Figure 36: Final design of Board #2............................................................. 63
Figure 37: Final design of board #1............................................................... 64
Figure 38: A look inside the box without boards......................................... 65
Figure 39: The top platform with fuse module, servos and camera module attached.............................................. 66
Figure 40: View of the bottom of the frame with motors attached.................. 67
Figure 41: Testing the rangefinders............................................................... 72
Figure 42: Testing the motor, motor drivers, and PSoC used to drive the locomotion module................................. 73
Figure 43: Pulse width at 10% duty cycle...................................................... 74
Figure 44: Pulse width at 20% duty cycle...................................................... 74
Figure 45: Pulse width at 60% duty cycle...................................................... 75
ACKNOWLEDGEMENTS

We would like to express our appreciation to our advisor Dr. Yanfei Liu for making this project possible. We would also like to thank Dr. Liu for all of her input and guidance throughout this project. Without her knowledge and support this project would not have come together.

We would also like to thank the Institute of Electrical and Electronics Engineers Standards Education Committee for their generosity with regards to funding our project.
ABSTRACT

The goal of the Low-Cost Autonomous Multi-Robot Network project is to create two robots that can communicate wirelessly with each other, that can avoid obstacles, and be built for less than $1000 dollars each. The robots are to use various sensors for obstacle avoidance and communication. All programs will be written in the same high level language, which has been chosen to be C, to maintain a cohesive language set across all aspects of each robot. Each robot must utilize a visual sensor, which has been selected as the CMUCam3, which is used in communicating the existence of another robot in the network. Even though the CMUCam3 was selected for this project, it was discontinued and unavailable for the build phase of the project, so the uCAM-TTL will be used instead. This camera is a serial JPEG camera module which uses a serial TTL interface for communication. The mechanical frames of the robots are built from commercially available kits. Another part of the project is to make the robot network easily expandable for use in other projects. This aspect led to developing a modularly built robot. Modularity allows for easy exchanging of core components and sensors. It also allows for easy upgrades to the robots for future use.
SECTION 1: DESIGN DESCRIPTION
1.1 REQUIREMENTS AND SPECIFICATIONS

The following requirements and specifications will provide the basic structure needed for the design and the evaluation of different designs. It is important to determine the requirements and specifications of the system so as to have a focus for the project.

- Mobility – The robots must be mobile and able to move forward at a rate of at least 1 feet/s and rotate at a rate of at least 90 degrees/s. In addition the robots must turn to the right/left and also travel backward if needed.
- Visual Sensors – uCam is required to get visual data from the point of view of the robots.
- Motion Sensors – Must be able to use various sensors for obstacle avoidance and telemetry.
- Power – The robots must be capable of recharge from an external charging unit. On top of that the units should display a warning when the power is less than the 20% of battery.
- Flags – Sound and light must be generated to indicate the status of the robot.
- Interface – The ability for a human user to turn the unit on and off.
- Controls – The device must be controlled by a microcontroller that can be used for a lengthy period of time to accommodate future robot additions.

1.2 GIVEN PARAMETERS

The following given parameters are specified and cannot be altered during the design of the robots.

1.2.1 Hardware

- Wireless range – Robots should communicate in a range of 10 feet.
- Visual Sensor – uCam.
- Budget – Each robot should be able to be built with $1,000.00 or less in cost of parts.

1.2.2 Operating Standards

- The external device to charge the robot must be under American Federal Safety Law.

1.3 DESIGN VARIABLES

The following design variables are aspects of the project that the team has the freedom to plan.

1.3.1 Hardware

- Geometry – The geometry of the robots must be capable of at least 2 degrees of movement and be able to maneuver over at flat surface.
- Dimensions – The dimensions of the robots must be in a range of 1 ft$^3$.
- Weight – The robots must have a weight within an acceptable range to determine the speed.
- Light Emission – The robot must have LEDs to warning for low level of battery.
- Speakers – The robot must have speakers to determine that the task is done.
- Operating Time - Robots should be capable of 30 minutes of activity on a single charge before needing recharged.
• Sensors - The robots should be able to detect protruding obstacles in front of the robot within 2 feet.
• Communications - Robots will have a wireless communication system allowing them to transmit and receive data up to 10 feet away.
• Mobility – Omni-directional wheels will be considered for the design.
• LED array – Consideration will be given to a visual LED array mounted on the robot such that patterns could be seen from other robots using the vision sensor.

1.3.2 Software
• Signal Processing – A programming language will be selected according to the microprocessor chosen. Codes will be created to get data from the sensors and make the robots functional. Codes should be easy to understand for future development of the robot.
• Driver – A driver will be developed to recognize and control hardware and peripheral systems.
• Image Interpretation – The robot must be able to interpret the captured images by the uCam and send it to the other robot.
• Sensors Interpretation – The robot must be able to interpret each sensor Data to avoid obstacles along the way.
• Communications Framework - A normalized communications framework needs to be created to allow for standardized communications for CPU to peripheral and inter-robot communication.
• Programming Language – The programming language should be a high level language such as C.

1.3.3 Operating Conditions
• Environmental Conditions – The unit is expected to operate indoors.

1.3.4 Limitations and Constraints
• Material – The frame material used on the robot must be strong enough to hold the weight from the sensors and non-toxic for humans and the environment.
• Durability – The robot must be capable of hitting an obstacle that may appear suddenly in the robot’s path without inflicting damage upon the robot.
• Microcontroller – For future compatibility the source needs to be able to work on other hardware.

1.3.5 Safety, Environmental, and other Considerations
• Excess capacity – Extra memory must be easy to install if the microprocessor user does not have enough memory to the code being loaded.
• Robot frame – The frame of the robot must be easy to install new sensors or any device to develop the original design.
• Battery – The battery from the robot must be easy to recharge and change whenever the battery gets old.
1.4 ORIGINAL CONCEPTUAL DESIGN

1.4.1 Overview Diagram

Figure 1 is the original overall design diagram that was to be the basis of our final design. The controller module’s main components were to be the NetBurner MOD5270, NBWIFIUG-100CR WIFI Adaptor,
DLP-USB232R USB to Serial, and CMUCam3. The locomotion module’s main components were to be the Cypress CY8C29466 Mixed Signal MCU, the 5V DC motors with rotary encoders, and the 2 TB6612FNG dual motor drivers. The power module’s main components were to be the 7.2V 2200 mAh NiMH battery, the 5V switching voltage regulator, the 3.3V switching voltage regulator, the power switch, and the LM393 comparator IC. The Sensor/Feedback module’s main components were to be the Cypress CY8C29466 Mixed Signal MCU, the 3 Maxbotix MB1300 ultrasonic rangefinders, the DFR0032 buzzer, and 2 LEDs. All of these modules were to communicate with each other through an I2C bus and supplied with either 3.3V or 5V from the power module.

1.4.2 Functional Description Overview
The Clickybot Rover will use a NetBurner MOD5270 module as its main CPU. The camera and wireless communications modules will be tightly bound to this main controller module. This breaks our design philosophy somewhat but was needed in this case due to the high bandwidth and tight integration that these components need. Future Clickybot designs may choose to decouple these. The power subsystem is designed to accommodate multiple modules with varying power requirements. For this first design, these were 5 volts and 3.3 volts, with one component (The CMUCam3) needing 6 or more volts to operate. Other modules will be connected to the CPU/Controller module through a 3.3V I2C bus mastered by the controller module. We choose 3.3V as it would then be compatible with modules based on 3.3 or 5 volt logic. For our initial design, we add two modules onto the bus. A locomotion module to handle the motor drivers and rotary encoders and a Sensor/Feedback module to handle an array of ultrasonic rangefinders as well as status LEDs and a buzzer.

1.4.3 Controller Module

1.4.3.1 Overview
The Controller Module is the heart of the Clickybot Rover. It serves as the primary CPU with its NetBurner MOD5270. It also has a USB port and Wi-Fi module along with the Ethernet adapter built into the NetBurner. The Controller module will have a connector for serial communications to the CMUCam3 module mounted on top of the rover. It also serves as the I2C bus master for communications to the peripheral modules like the Locomotion module and Range Finder module.

1.4.3.2 Components - Interconnections
- NetBurner MOD5270 CPU module (1)
- NBWIFIUG-100CR WIFI Adaptor Module (1)
- DLP-USB232R USB to Serial module (1)
- CMUCam3 camera module (1)
- 4-pin I2C connector (1)
- 5-pin CMUCam3 connector (1)
- 2-pin power connector (1)
- Resistors (4)
  - Two 4.7kΩ I2C pullup resistors
  - A 5.6kΩ and 10kΩ for 5V to 3.3V level shifter.
- 22µF Capacitor for power input decouple (1)
1.4.3.3 Functional Description

The NetBurner MOD5270 will serve as the primary CPU and controller for the Clickybot rover. It uses a 32-bit 147Mhz Freescale Coldfusion processor chip. The NetBurner module also supports the µC/OS real-time operating system and has a full IP stack with many standard protocols such as low level TCP and UDP to higher level protocols like HTTP, FTP, Telnet, and NTP. These higher level protocols will be the basis for communications over the Wi-Fi link when the unit is operating and the Ethernet link on the bench for debugging and development.

The rover will accept commands from a central monitoring station via Wi-Fi and will communicate to other robots and the monitoring station in this way. Commands will be filtered through this module and sent on to the other peripherals. Data from the CMUCam3 and other devices will be broadcast from the rover so that other rover can know its status or monitor its functions for coordination.

One example will be the preparation of the images from the CMUCam3 for broadcast back to the monitor station. Raw images will be JPEG compressed on the NetBurner as it has more processing power than the processor on the CMUCam3. These images then can be processed used by the monitor station as a display or further processing can be done.

1.4.3.4 Programming

The Controller Module will be programmed in C/C++. The compiled code will be loaded onto the module using NetBurner over Wi-Fi or Ethernet. There is a low level boot loader that can be accessed via the USB port in case normal functionality is compromised by a software error during development. The CMUCam3 is also capable of being programmed. This will be done through its own RS232 serial port when needed. In this way, image processing functions can be effectively shared between the CMUCam3 processor and the 32-bit Freescale processor.
1.4.3.5 Schematic

Figure 2: Schematic for the Controller Module.

The original wiring schematic for the controller module is shown in figure 2. This schematic was to be used to aid the team in wiring the circuit boards for the final robot design.

1.4.4 Locomotion Module

1.4.4.1 Overview

The Clickybot Locomotion Module consists of a Cypress PSoC MCU connected to the four motors of the DFRobot 4WD platform through two TB6612FNG Dual Motor Drivers. Feedback to the MCU will be through two rotary encoders on one motor on each side (left and right) of the 4WD platform. The MCU connects to the Clickybot main CPU via the central I2C control bus. Here it will receive direct commands and relay its status data for further processing by the CPU.
1.4.4.2 Components - Interconnections

- Cypress CY8C29466 Mixed Signal MCU (1)
- DFRobot 4WD platform (1)
  - 5V DC motors (4)
  - Rotary Encoders (2)
- TB6612FNG Dual Motor Drivers (2)
- 4-pin power/I^2C connector (1)
- 5-pin PSoC programming port (1)
- 2-pin connector for DC motors (4)
- 2-pin connector for rotary encoders (2)
- 22µF Capacitor for power input decouple (1)

1.4.4.3 Functional Description

The Locomotion Array Module will connect to four 5-volt DC motors and the two rotary encoders using 2-pin screw-type header terminals. The motors will be driven by the two dual H-bridge motor drive modules mounted on the Locomotion module PCB. These H-bridge modules (one for the two front motors and one for the rear motors) are controlled in tandem by the PWM outputs of the MCU. There are separate control lines for the Left and Right motors. Feedback to the MCU will be through two rotary encoders located on the front two motor shafts. The MCU will receive commands from the Clickybot Controller module CPU via the I^2C bus. It will also report status back to the CPU on this bus when prompted. The MCU will be responsible for executing commands given to it by the CPU. The commands will be high level directives rather than low level instructions like move forward at 0.5 m/s or arc left at 0.3 m/s radius 1m. The MCU will interpret these commands and act on them accordingly. It will also interpret the encoders to give feedback to the CPU describing speed, acceleration, and rotation.

1.4.4.4 Programming

The Locomotion Module will be programmed in C. The compiled code will be loaded onto the module using the 5-pin programming port using the I^2C standard.
1.4.4.5 Schematic

Figure 3: Schematic for the locomotion module.

The original wiring schematic for the locomotion module is shown in figure 3. This schematic was to be used to aid the team in wiring the circuit boards for the final robot design.

1.4.5 Power Module

1.4.5.1 Overview

The Clickybot Power Module consists of a 7.2V 2200 mAh NiMH Battery and 3.3V and 5V switchable voltage regulators. The battery will be located between the motors and underneath the main module housing. This connects to the power regulators and distribution of the Power Module.
1.4.5.2 Components - Interconnections

- 7.2V 2200 mAh NiMH Battery (1)
- Switching Voltage Regulators (2)
  - DE-SW050 5V regulator
  - DE-SW033 3.3V regulator
- Power switch (1)
- LM393 Comparator IC (1)
- 2-pin connector for LEDs (2)
- 300Ω resistors (2)
- 10kΩ Trim Potentiometer (1)
- 2-pin battery connector (1)
- 2-pin power distribution connectors (9)
- 47µF Capacitor for power filtering (2)
- 100µF Capacitor for power filtering (2)

1.4.5.3 Functional Description

The power module will regulate and condition power for the other modules in the system. It will have 3 connectors each for 7.2V (from the battery) 5V and 3.3V. Power coming in from the battery will be filtered with a 100µF Capacitor and the output from each regulator will be filtered with a 47µF capacitor.

The power module also has a Low Voltage indicator LED. The threshold for the low voltage will be set by the 10k trim potentiometer. The tap of the pot is compared to the 3.3V of the regulator via the LN393 comparator IC. The 3.3 volt regulator is used since this will be the last power supply affected as the battery discharges. The output from the comparator should light the LED when the battery voltage falls below the desired threshold. We will test different threshold voltages to get an adequate warning level for our application.
1.4.5.4 Power Calculation

Table 1: Power requirements used to determine battery needs.

<table>
<thead>
<tr>
<th>Part</th>
<th>Pieces</th>
<th>Volts</th>
<th>mA peak</th>
<th>Estimated Duty %</th>
<th>mW</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC motors</td>
<td>4</td>
<td>5</td>
<td>200</td>
<td>75%</td>
<td>3000</td>
<td>30</td>
</tr>
<tr>
<td>NetBurner</td>
<td>1</td>
<td>3.3</td>
<td>250</td>
<td>100%</td>
<td>825</td>
<td>8.25</td>
</tr>
<tr>
<td>Wi-Fi adapter</td>
<td>1</td>
<td>3.3</td>
<td>160</td>
<td>100%</td>
<td>528</td>
<td>5.28</td>
</tr>
<tr>
<td>CMUcam3</td>
<td>1</td>
<td>7.2</td>
<td>130</td>
<td>100%</td>
<td>936</td>
<td>9.36</td>
</tr>
<tr>
<td>Ultrasonic rangefinders</td>
<td>3</td>
<td>5</td>
<td>100</td>
<td>4%</td>
<td>60</td>
<td>0.6</td>
</tr>
<tr>
<td>PSoC</td>
<td>2</td>
<td>5</td>
<td>25</td>
<td>100%</td>
<td>250</td>
<td>2.5</td>
</tr>
<tr>
<td>Circuit board</td>
<td>1</td>
<td>5</td>
<td>50</td>
<td>100%</td>
<td>250</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total (mW)</td>
<td></td>
<td></td>
<td></td>
<td>Total (W)</td>
<td>5849</td>
<td>58.49</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Total (mA)</td>
<td>812.36</td>
<td>8.12</td>
</tr>
<tr>
<td>Regulator 80% efficiency</td>
<td></td>
<td></td>
<td></td>
<td>80% efficiency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1015.45 mA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10.15 A</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 was used to determine the battery we needed. This was done by multiplying the milliamps by the volts to determine the milliwatts. This number was then multiplied by the number of pieces needed. After adding all the milliwatts, for each part, together this number was divided by the 7.2V supplied by the battery to determine the overall milliamps. Last this number was divided by the efficiency of the voltage regulators. The regulators we chose have a worst-case efficiency rating of 80%. The final number is the milliamps that the battery needs to supply. As seen in the table the milliamps needed is 1015.45. We want the robot to operate for at least an hour so the battery should be rated at least at 1015.45 milliamp-hours. We wanted to allow for expansion for future projects so we doubled this so we need a battery that is rated at 2030.90 milliamp-hours. After looking at some batteries it was determined that the closest standard milliamp-hours that a battery would be rated at is 2200 milliamp-hours, so we chose a battery that supplied 7.2V at 2200mAh.
1.4.5 Schematic

The original wiring schematic for the power module is shown in figure 4. This schematic was to be used to aid the team in wiring the circuit boards for the final robot design.

1.4.6 Sensor/Feedback Module

1.4.6.1 Overview
The Clickybot Sensor/Feedback Module consists of an array of Maxbotix ultrasonic range finders controlled by a Cypress PSoC MCU. Along with this are connectors for 3 status LEDs and a buzzer module to provide feedback. The MCU connects to the Clickybot main CPU via the central I²C control bus. Here it will receive direct commands and relay its range data for further processing by the CPU.
1.4.6.2 Components - Interconnections

- Cypress CY8C29466 Mixed Signal MCU (1)
- Maxbotix MB1300 Ultrasonic Rangefinders (3)
- 2-pin buzzer connector for a DFR0032 buzzer module (1)
- 2-pin LED connectors (3)
- 200Ω resistors (3)
- 4-pin power/I²C connector (1)
- 5-pin PSoC programming port (1)
- 7-pin connector for MB1360 cables (3)

1.4.6.3 Functional Description

The Sensor/Feedback Module will connect to 3 Maxbotix MB1300 modules located at the front of the Clickybot Rover. These will be controlled by the Cypress PSoC (CY8C29466) MCU. The MCU will be responsible for “pinging” the sensors during predetermined time-slices. The time-slices will be sent as a configuration command from the CPU and will be coordinated with other Clickybots in the area so that sensor pings from other robots will not conflict and give false reading. The MCU will then coordinate the pinging of each of its sensors during the time-slice to avoid interference. The readings from each sensor will be processed by the PSoC and they resulting range and status data will be sent back to the CPU via the I²C bus.

The module will also take commands to light the various status LEDs and make beep noises. The meanings for these will be application specific and will be determined by later programming but the idea here is to allow audible and visual feedback to indicate that the robot is engaged in or completed various tasks.

1.4.6.4 Programming

The Sensor/Feedback Module will be programmed in C. The compiled code will be loaded onto the module using the 5-pin programming port using the I²C standard.
The original wiring schematic for the sensor/feedback module is shown in figure 5. This schematic was to be used to aid the team in wiring the circuit boards for the final robot design.
Figure 6: 3D view of the overall design of the robot.

Figure 6 shows the original 3D mock-up of the overall design of the robots.
1.5 FINAL DESIGN OVERVIEW

1.5.1 Electrical Scheme
The following figure is a general overview of the approach the team is taking in designing the robots.

![Diagram of robot design](image)

Figure 7: The general overview of the design of a robot.

1.5.2 Design Philosophy
For the design of the multi-robot network, we were given several requirements and design constraints. The overall theme of these was to devise a system and design that could be used into the future to empower other research projects. As such, the individual robot rovers needed to be relatively low cost and fairly easy to construct and expand to meet future needs. To this end a system was designed that was modular and relied on several well established standards. The thought process here is that if various parts and functions of the robot could be replaced and expanded, then future projects would be easier to implement. Also, if any of the explicit components that we choose for the original design become obsolete, the particular module could be redesigned with modern components. In keeping with the modular approach, each module is designed with the same PCB footprint and with a standardized mounting-hole scheme. In this way the units could be mounted together with standoffs and fit nicely into an enclosure.

1.5.3 Architectural Overview
Our basic design consists of a primary processor unit connected via a bus architecture to several coprocessor units. In this way the primary processor can focus on higher level functions such as TCP communications, image processing, and command and control. This way each coprocessor can concentrate on a particular function such as sensor import, motor control, and other tasks.
1.5.4 Overview Diagram

Figure 8 above shows the overall design diagram. This diagram conveys the general flow of communications, connections, and power routing throughout the robots.

1.5.5 Functional Description Overview

The Clickybot, the name we have given the robots, Rover will use a NetBurner MOD5270 module as its main CPU. The camera and wireless communications modules will be tightly bound to this main controller module. This breaks our design philosophy somewhat but was needed in this case due to the high bandwidth and tight integration that these components needed. Future Clickybot designs may choose to decouple these.

The power subsystem is designed to accommodate multiple modules with varying power requirements. For this first design, these were 5 volts and 3.3 volts, with one component needing 6 or more volts to operate.
Other modules will be connected to the CPU module through a 3.3V I\(^2\)C bus mastered by the controller module. We choose 3.3V as it would then be compatible with modules based on 3.3 or 5 volt logic.

1.5.6 Standards Used
Several Engineering standards were used in our design. We will briefly discuss two of these, namely the IEEE 802.11g Wireless communications standard and the I\(^2\)C two-wire bus standard. Other standards were used such as 5V and 3.3V TTL logic, serial communications, USB and Ethernet, but these are not the highlights of this design.

1.5.6.1 IEEE 802.11 (Wi-Fi)
We choose the IEEE 802.11 standard for our wireless communications. We had also considered IEEE 802.15.14 (ZigBee) as a possibility but ultimately choose Wi-Fi for its higher bandwidth, and integration with the TCP/IP standard packet protocols. Another factor included the availability of integrated Wi-Fi hardware with our choice of the NetBurner as our CPU platform. The NetBurner OS has many standard Internet protocols available that can better be utilized with Wi-Fi hardware rather than ZigBee. It should be noted however that with our modular design, future implementations might choose a different CPU and use ZigBee or another wireless standard instead.

1.5.6.2 I\(^2\)C
The choice of the I\(^2\)C protocol in our design for the bus was another decision point. We had also considered the SPI standard. We felt that the I\(^2\)C standard was better accepted and required fewer signal lines than SPI. Also, I\(^2\)C better supported 3.3 and 5 volt logic. This allows our system to be compatible with modules that are based on either standard. Also, we felt that the need of a chip select line for SPI would complicate the design. I\(^2\)C uses an addressing scheme so separate select signals are not needed.

1.5.7 CPU Module

1.5.7.1 Overview
The CPU Module is the heart of the Clickybot Rover. It serves as the primary CPU with its NetBurner MOD5270. It also has a USB port and Wi-Fi module along with the Ethernet adapter built into the NetBurner.

The CPU module will have a connector for serial communications to the uCam module mounted on 2 servos on top of the rover. It also serves as the I\(^2\)C bus master for communications to the peripheral modules like the RangeFinder/Servo module.

1.5.7.2 Components - Interconnections
- NetBurner MOD5270 CPU module (1)
- IOGEAR Universal Wi-Fi N Adapter [Ethernet to Wi-Fi] (1)
- DLP-USB232R USB to Serial module (1)
- uCam camera module (1)
- 4-pin I\(^2\)C connector (1)
- 4-pin uCam connector (1)
- 1.8kΩ Resistors (2)
- 22µF Capacitor for power input decouple (1)
1.5.7.3 Functional Description

The NetBurner MOD5270 will serve as the primary CPU and controller for the Clickybot rover. It uses a 32-bit 147Mhz Freescale Coldfusion processor chip. The NetBurner module also supports the µC/OS real-time operating system and has a full IP stack with many standard protocols such as low level TCP and UDP to higher level protocols like HTTP, FTP, Telnet, and NTP. These higher level protocols will be the basis for communications over the Wi-Fi link when the unit is operating and the Ethernet link on the bench for debugging and development.

The rover will accept commands from a central monitoring station via Wi-Fi and will communicate to other robots and the monitoring station in this way. Commands will be filtered through this module and sent on to the other peripherals. Data from the uCam and other devices will be broadcasted from the rover so that other rover can know its status or monitor its functions for coordination. One example will be the preparation of the images from the uCam for broadcast back to the monitor station. Raw images will be JPEG compressed on the NetBurner as it has more processing power than the processor on the uCam. These images can then be processed and used by the monitor station as a display or further processing can be done.

1.5.7.4 Programming

The CPU Module will be programmed in C/C++. The compiled code will be loaded onto the module using NetBurner over Wi-Fi or Ethernet. There is a low level boot loader that can be accessed via the USB port in case normal functionality is compromised by a software error during development.
The wiring schematic for the CPU module is shown in figure 9. This schematic was used to aid the team in wiring the circuit boards for the final robot design.
1.5.7.6 Board Layout

The red square in figure 10 shows the location of the main components in the CPU module on the circuit boards designed for use in the final robot design. The main components in the red square are the 2 NetBurner and the USB to serial module.

1.5.8 Locomotion Module

1.5.8.1 Overview
The Clickybot Locomotion Module consists of a Cypress PSoC MCU connected to the four motors of the DFRobot 4WD platform through two TB6612FNG Dual Motor Drivers. Feedback to the MCU will be through two rotary encoders on one motor on each side (left and right) of the 4WD platform. The MCU connects to the Clickybot main CPU via the central I²C control bus. Here it will receive direct commands and relay its status data for further processing by the CPU.
1.5.8.2 Components - Interconnections

- Cypress CY8C29466 Mixed Signal MCU (1)
- DFRobot 4WD platform (1)
  - 5V DC motors (4)
  - Rotary Encoders (2)
- TB6612FNG Dual Motor Drivers (2)
- 4-pin power/I²C connector (1)
- 5-pin PSoC programming port (1)
- 2-pin connector for DC motors (4)
- 2-pin connector for rotary encoders (2)
- 22µF Capacitor for power input decouple (1)

1.5.8.3 Functional Description

The Locomotion Array Module will connect to four 5-volt DC motors and the two rotary encoders using 2-pin screw-type header terminals. The motors will be driven by the two dual H-bridge motor drive modules. These H-bridge modules (one for the two left-side motors and one for the right-side motors) are controlled in tandem by the PWM outputs of the MCU. There are separate control lines for the Left and Right motors. Feedback to the MCU will be through two rotary encoders located on the front two motor shafts.

The MCU will receive commands from the Clickybot CPU module CPU via the I²C bus. It will also report status back to the CPU on this bus when prompted. The MCU will be responsible for executing commands given to it by the CPU. The commands will be high level directives rather than low level instructions like move forward at 0.5 m/s or arc left at 0.3 m/s radius 1m. The MCU will interpret these commands and act on them accordingly. It will also interpret the encoders to give feedback to the CPU describing speed, acceleration, and rotation.

1.5.8.4 Programming

The Locomotion Module will be programmed in C. The compiled code will be loaded onto the module using the 5-pin programming port using the I²C standard.
The wiring schematic for the locomotion module is shown in figure 11. This schematic was used to aid the team in wiring the circuit boards for the final robot design.
1.5.8.6 Board Layout

The red square in figure 12 shows the location of the main components in the locomotion module on the circuit boards designed for use in the final robot design. The main components in the red square are the 2 motor drivers and the PSoC.
1.5.9 Power Module

1.5.9.1 Overview
The Clickybot Power Module consists of a 7.2V 3000 mAh NiMH Battery, 3.3V and 5V switchable voltage regulators, 2 adjustable voltage regulators, a 3 amp fuse, and an AttoPilot 13.6V/45A Voltage/Current sensor. The battery will be located between the motors and underneath the main module housing. This connects to the power regulators and distribution of the Power Module. The AttoPilot 13.6V/45A Voltage/Current sensor allows us to monitor the voltage and current of the battery in real time. The 2 adjustable voltage regulators allow each motor to be driven up to 500mA without overloading the switchable 5V regulator.

1.5.9.2 Components - Interconnections
- 7.2V 3000 mAh NiMH Battery (1)
- Switching Voltage Regulators (2)
  - DE-SW050 5V regulator
  - DE-SW033 3.3V regulator
- Adjustable Voltage Regulators (2)
- 3A fuse
- Power switch (1)
- 120Ω resistors (2)
- 470Ω resistors (2)
- 2-pin battery connector (1)
- 220µF Capacitor for power filtering (4)
- AttoPilot 13.6V/45A Voltage/Current sensor (1)

1.5.9.3 Functional Description
The power module will regulate and condition power for the other modules in the system. It will have 3 connectors each for 6V, 5V and 3.3V. Power coming in from the battery will be filtered with a 220µF Capacitor and the output from each regulator will be filtered with a 220µF capacitor.
### 1.5.9.4 Power Calculation

Table 2: Power requirements used to determine battery needs.

<table>
<thead>
<tr>
<th>Part</th>
<th>Pieces</th>
<th>Volts</th>
<th>mA peak</th>
<th>Estimated Duty %</th>
<th>mW</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>DC motors</td>
<td>4</td>
<td>5</td>
<td>200</td>
<td>75%</td>
<td>3000</td>
<td>3</td>
</tr>
<tr>
<td>NetBurner</td>
<td>1</td>
<td>3.3</td>
<td>250</td>
<td>100%</td>
<td>825</td>
<td>0.825</td>
</tr>
<tr>
<td>Wi-Fi adapter</td>
<td>1</td>
<td>3.3</td>
<td>160</td>
<td>100%</td>
<td>528</td>
<td>0.528</td>
</tr>
<tr>
<td>uCam</td>
<td>1</td>
<td>7.2</td>
<td>130</td>
<td>100%</td>
<td>936</td>
<td>0.936</td>
</tr>
<tr>
<td>Ultrasonic rangefinders</td>
<td>3</td>
<td>5</td>
<td>100</td>
<td>4%</td>
<td>60</td>
<td>0.06</td>
</tr>
<tr>
<td>PSoC</td>
<td>2</td>
<td>5</td>
<td>25</td>
<td>100%</td>
<td>250</td>
<td>0.25</td>
</tr>
<tr>
<td>LEDs</td>
<td>2</td>
<td>5</td>
<td>30</td>
<td>100%</td>
<td>300</td>
<td>0.3</td>
</tr>
<tr>
<td>LCD</td>
<td>1</td>
<td>3.3</td>
<td>60</td>
<td>100%</td>
<td>198</td>
<td>0.198</td>
</tr>
<tr>
<td>Servo</td>
<td>2</td>
<td>3.3</td>
<td>450</td>
<td>100%</td>
<td>2970</td>
<td>2.97</td>
</tr>
<tr>
<td>Compass</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>100%</td>
<td>5</td>
<td>0.005</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>100%</td>
<td>20</td>
<td>0.02</td>
</tr>
<tr>
<td>Speaker</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>100%</td>
<td>10</td>
<td>0.01</td>
</tr>
<tr>
<td>Wireless</td>
<td>1</td>
<td>5</td>
<td>380</td>
<td>100%</td>
<td>1900</td>
<td>1.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total (mW)</th>
<th>Total (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11002</td>
<td>11.002</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total (mA)</th>
<th>Total (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1528.06</td>
<td>1.528</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Regulator</th>
<th>80% efficiency (mA)</th>
<th>80% efficiency (A)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1910.07</td>
<td>1.910</td>
</tr>
</tbody>
</table>

Table 2 was used to determine the battery we needed. This was done by multiplying the milliamps by the volts to determine the milliwatts. This number was then multiplied by the number of pieces needed. After adding all the milliwatts, for each part, together this number was divided by the 7.2V supplied by the battery to determine the overall milliamps. Last this number was divided by the efficiency of the voltage regulators. The regulators we chose have a worst-case efficiency rating of 80%. The final number is the milliamps that the battery needs to supply. As seen in the table the milliamps needed is 1528.06. We want the robot to operate for at least an hour so the battery should be rated at least at 1528.06 milliamp-hours. After adding to the power requirements we chose a battery that supplied 7.2V at 3000mAh to allow for some expansion.
The wiring schematic for the power module is shown in figure 13. This schematic was used to aid the team in wiring the circuit boards for the final robot design.
1.5.9.6 Board Layout

The red square in figure 14 shows the location of the main components in the power module on the circuit boards designed for use in the final robot design. The main components in the red square are the 2 switching voltage regulators, the 2 adjustable voltage regulators and the AttoPilot voltage/current sensor.
1.5.10 Rangefinder/Servo Module

1.5.10.1 Overview
This module is responsible for reading the three range finders and controlling the servos for the pan tilt of the camera. During testing, we discovered that we got better accuracy and less overlap with the ultrasonic sensors running them at 3.3V rather than 5.0V. This was due in part because the sensors were mounted so low and we were getting echoes from the floor in front of the robots.

1.5.10.2 Functional Description
The RangeFinder/Servo Module will connect to 3 Maxbotix MB1300 modules located at the front of the Clickybot Rover. These will be controlled by the Cypress PSoC (CY8C29466) MCU. The MCU will be responsible for “pinging” the sensors during predetermined time-slices. The time-slices will be sent as a configuration command from the CPU and will be coordinated with other Clickybots in the area so that sensor pings from other robots will not conflict and give false reading. The MCU will then coordinate the pinging of each of its sensors during the time-slice to avoid interference. The readings from each sensor will be processed by the PSoC and the resulting range and status data will be sent back to the CPU via the I²C bus.

The module will also take commands to operate the 2 servos that manipulate the pan/tilt for the camera.

1.5.10.3 Programming
The RangeFinder/Servo Module will be programmed in C. The compiled code will be loaded onto the module using the 5-pin programming port using the I²C standard.
The wiring schematic for the RangeFinder/Servo module is shown in figure 15. This schematic was used to aid the team in wiring the circuit boards for the final robot design.
1.5.10.5 PSoC Connections

Figure 16 shows the labeled connections for the RangeFinder/Servo module PSoC. This diagram aided the team in wiring the circuit boards by showing which component is connected to the PSoC and which pin that component utilizes on the PSoC.
1.5.10.6 Board Layout

The red square in figure 17 shows the location of the main components in the RangeFinder/Servo module on the circuit boards designed for use in the final robot design. The main components in the red square are the PSoC and servo communication pins.
1.5.11 Sensor Module

1.5.11.1 Overview
The sensor module reads the 2-axis accelerometer and drives the piezo speaker and two RGB LEDs for output. The speaker and LEDs will provide a way of alerting the user to the different statuses of the robot. The module also hosts the digital compass that connects directly to the I²C bus.

1.5.11.2 Components - Interconnections
- Cypress CY8C29466 Mixed Signal MCU (1)
- RGB LEDs (2)
- Piezo Speaker (1)
- 2-axis Accelerometer (1)
- Digital Compass (1)

1.5.11.3 Programming
The Sensor Module will be programmed in C. The compiled code will be loaded onto the module using the 5-pin programming port using the I²C standard.
Figure 18: Schematic for the Sensor Module.

The wiring schematic for the sensor module is shown in figure 18. This schematic was used to aid the team in wiring the circuit boards for the final robot design.
Figure 19 shows the labeled connections for the sensor module PSoC. This diagram aided the team in wiring the circuit boards by showing which component is connected to the PSoC and which pin that component utilizes on the PSoC.
The red square in figure 20 shows the location of the main components in the sensor module on the circuit boards designed for use in the final robot design. The main components in the red square are the speaker, the accelerometer, the digital compass, and the PSoC.
Figure 21 shows where the motors are located on the frame of the robot. This figure also shows where the encoders are located. The motors are the yellow L-shaped blocks and the encoders are the light gray squares on the motors.
1.5.12.2 Battery

Figure 22: Battery Placement.

Figure 22 shows where the battery is located on the frame of the robot. The battery is the long gray block with black caps.
1.5.12.3 Rangefinders

Figure 23 shows where the ultrasonic sensors are located on the frame of the robot. This figure also shows the top platform. The ultrasonic sensors are the 3 black cylinders in the front.
1.5.12.4 Fuse

Figure 24: Placement of the 3 amp fuse.

Figure 24 shows where the fuse module is located on the frame of the robot. The fuse module is the green block labeled FUSE located in the back.
1.5.12.5 Circuit Board #1

Figure 25 shows where the first circuit board is located on the robot. Circuit board #1 is the green block labeled PsOC located in the back. This circuit board contains the power module, the locomotion module, and the RangeFinder/Servo module.
1.5.12.6 Circuit Board #2

Figure 26: Second Circuit Board Placement.

Figure 26 shows where the second circuit board is located on the robot. Circuit board #2 is the green block labeled NETBURNER located in the back. This circuit board contains the CPU module and the sensor module.
1.5.12.7 Enclosure

Figure 27: Placement of the box that protects the circuit boards.

Figure 27 shows where the enclosure is located on the robot. The enclosure is the black box that contains the 2 circuit boards.
1.5.12.8 Servos

Figure 28 shows where the 2 servos are located on the robot. The servos are the black blocks located in front of the enclosure on the brass colored standoffs.
1.5.12.9 Camera

Figure 29: Placement of the uCam visual sensor.

Figure 29 shows where the uCam is located on the robot. The uCam is the green and black square on the servo stack.
Figure 30 shows where the LCD is located on the robot. The LCD is the red black and green block on the side of the enclosure.
1.5.12.11 LEDs

Figure 31: Locations for the 2 RGB LEDs.

Figure 31 shows where the 2 RGB LEDs are located on the robot. The LEDs are the 2 blue dots on the front of the enclosure on either side of the servo stack.
1.5.12.12 Wireless Module

Figure 32 shows where the wireless module is located on the robot. The wireless module is the black block on the top of the enclosure. This figure is also the final mock-up of the final robot design.
Figure 33 shows a picture of the final robot. Comparing figures 26 and 27 shows that our final design is extremely close to the final mock-up.
SECTION 2: BUILDING PROCESS
2.1 OVERVIEW OF THE BUILDING PROCESS

The building process started by getting to know the programs needed to program the PSoCs to do the tasks assigned to them. After a few simple tests were performed, like making LEDs blink, in order to make sure that the PSoC was being programmed correctly, our group started testing different components of the robot. Since the design was modular each module could be built and tested before integrating them into a complete design. After each module was built and tested on breadboards, we began to wire up the boards with the tested designs. After each module was wired onto one of the boards we then tested them again to verify everything was wired correctly. After all boards were wired we then began writing the overall Netburner program that would communicate with each module through an I²C bus. After the program was complete, the robot as a whole was tested and from these tests the Netburner and PSoC programs were revised until the robots were working how we wanted them to.

2.2 BUILDING OF THE MODULES

Each module was built by first building and testing on breadboards, as shown in figure 34, and then transferring the final design to the boards and wiring them. After each module was wired on the board then that module was tested to make sure that it was wired correctly.

Figure 34: Testing the camera servos on a breadboard.
2.3 BUILDING OF THE CIRCUIT BOARDS

Each of the two circuit boards and the fuse module were built after testing the design on a breadboard. The designs on the breadboards were transferred to the circuit boards where we drilled, wired, and soldered wires and components together to achieve the final circuit.

The fuse module was wired on a circuit board for easy replacement. Figure 35 shows the fuse module in development.

Figure 35: 3 amp fuse module.
Figure 36 shows the final design of circuit board #2. This board has the CPU module and the sensor module wired on it.
Figure 37: Final design of board #1.

Figure 37 shows the final design of circuit board #2. This board has the locomotion module, the RangeFinder/Servo module, and the power module wired on it.
2.4 BUILDING THE FRAME OF THE ROBOTS

The frames, DFRobot 4WD, of the robots were bought as kits from Robotshop.com. These kits provided the basic structure for the robot and came with 4 motors and 2 encoders. The platform was built to provide the base for the robots and a box, shown in figure 38, was purchased to hold the circuit boards and to provide some protection. This box was attached to the top platform of the frame. To attach the servos for the camera we purchased standoffs to obtain the proper clearance needed. The 3 ultrasonic rangefinders were attached to the front by brackets that came with the frame. The battery is attached underneath the main platform by Velcro straps. The speaker and wireless module are attached to the box by Velcro.

Figure 38: A look inside the box without boards.
Figure 39: The top platform with fuse module, servos and camera module attached.

Figure 39 shows the top platform of the robot with the fuse module, the servo stack, and the uCam attached to it. The ultrasonic sensors are not attached to the platform at this point in the build process.
Figure 40 shows the frame of the robot with motors and encoders already attached to it. None of the wires for the motors or the encoders have been attached at this point.

2.5 CHANGES AND ADDITIONS

2.5.1 LCD
An LCD was added to the side of the box that houses the circuit boards for general information about the robot’s status and for debug information.

2.5.2 Servos
Two servos were added to the top platform to be used for camera pan and tilt capabilities.

2.5.3 Three Amp Fuse
A 3 amp fuse was added to protect the components from becoming damaged due to too much current.
2.5.4 Accelerometer
An accelerometer was added for determining the speed the robot is travelling.

2.5.5 Digital Compass
A digital compass was added for determining the direction the robot is facing.

2.5.6 Camera
A different camera was used, uCAM-TTL camera module, due to the CMUCam3 being discontinued, therefore unavailable to purchase.

2.5.7 Wireless Module
A different wireless module was used because we were unable to get the NBWIFIUG-100CR WIFI Adaptor Module working. The IOGear Universal Wi-Fi N Adapter was used instead.

2.5.8 Adjustable Voltage Regulators
Two adjustable voltage regulators were added for motor voltage regulation because one 5V switching regulator could not handle the amps drawn by the motors.

2.5.9 Battery
A different battery was used due to the addition of extra components like the servos. A 7.2V 3000 mAh battery was used instead of a 7.2V 2200 mAh one.

2.5.10 Added Voltage and Current Sensor
An AttoPilot 13.6V/45A Voltage/Current sensor was added to aid in determining the voltage and current of the battery.

2.6 DIFFICULTIES ENCOUNTERED

2.6.1 Visual Sensor
Initially had some issues with getting the CMUCam3 to communicate with the NetBurner, but then found out that this product was discontinued. Our team only had one CMUCam3 and we needed 2 so we had to find another visual sensor. The visual sensor that we decided on using was the uCam. Again, initially had some issues with getting the uCam to communicate with the NetBurner. These issues were overcome by selecting a different baud rate for the uCam.
2.6.2 Motors
The motors gave us some issues, in the fact, that they drew more current than what they were rated for when we checked the datasheet. One motor was supposed to draw 200mA which would be 800mA for all 4 motors, but when we measured the current draw all 4 motors were drawing about 2.1A. We original had a 5V switching regulator supplying the motors. The problem was that this regulator could only supply 5V at 1A, so we decided to use an adjustable voltage regulator for the motor voltage supply. We used one regulator to supply 2 motors thus cutting the current drawn through each regulator to 1.05 A, which solved the issue.

2.6.3 Wireless module
We had initially selected the NBWIFIUG-100CR WIFI, which, no matter what we tried, we could not get to connect to the wireless router that we were using. We decided to try a different wireless module, the IOGear Universal Wi-Fi N Adapter, which worked straight out of the box. No problems with getting it to connect right away with the wireless router.
SECTION 3: TESTING
3.1 OVERALL TESTING PROCEDURE
For our design we tested each module separately on breadboards and after each module was working on the breadboards we then transferred them to circuit boards that we then wired and soldered. After each module was transferred to the circuit boards then a multimeter was used to verify that the connections were working and correctly wired. After all modules were verified correct and soldered on the circuit boards then the system was tested as a whole.

3.2 REQUIREMENTS, SPECIFICATIONS, AND PARAMETERS
- **Mobility** – The robots must be mobile and able to move forward at a rate of at least 1 feet/s. In addition the robots must turn to the right/left and also travel backward if needed.
- **Visual Sensors** – uCam is required to get visual data from the point of view of the robots.
- **Motion Sensors** – Must be able to use various sensors for obstacle avoidance.
- **Power** – The robots must be capable of recharge from an external charging unit.
- **Flags** – Sound and light must be generated to indicate the status of the robot.
- **Interface** – The ability for a human user to turn the unit on and off.
- **Controls** – The device must be controlled by a microcontroller that can be used for a lengthy period of time to accommodate future robot additions.
- **Wireless range** – Robots should communicate in a range of 10 feet.
- **Budget** – Each robot should be able to be built with $1,000.00 or less in cost of parts.
- **Geometry** – The geometry of the robots must be capable of at least 2 degrees of movement and be able to maneuver over a flat surface.
- **Dimensions** – The dimensions of the robots must be in a range of 1 ft³.
- **Operating Time** – Robots should be capable of 30 minutes of activity on a single charge before needing recharged.
- **Sensors** - The robots should be able to detect protruding obstacles in front of the robot within 2 feet.
- **Signal Processing** – A programming language will be selected according to the microprocessor chosen. Codes will be created to get data from the sensors and make the robots functional. Codes should be easy to understand for future development of the robot.
- **Driver** – A driver will be developed to recognize and control hardware and peripheral systems.
- **Communications Framework** - A normalized communications framework needs to be created to allow for standardized communications for CPU to peripheral and inter-robot communication.
- **Programming Language** – The programming language should be a high level language such as C.
- **Environmental Conditions** – The unit is expected to operate indoors.

3.3 SENSOR TEST
To test the ultrasonic sensors we took a plastic sleeve that contained the PSoCs and measured the length of it. This sleeve measured 42 cm long. Next we took the robot into the hallway and placed the sleeve between the robot and the wall. Figure 41 shows this test setup. Next we ran the sensor program and the number we received back from the middle sensor was 44. This number should correspond to the distance, in centimeters, an object is from the sensor. We received 44 cm and the actual distance was 42 cm. This is an error of approximately 4.5%, which is acceptable. Also, 42 cm equals 1.38 feet which is less than the required 2 feet in which the robot is supposed to detect an obstacle.
3.4 DIMENSIONS TEST

To check the dimensions of the robot we used a ruler to measure the length, height and width of the completed robot. The height measured 8 inches. The length measured 8.5 inches. The width measured 6.75 inches. This results in a volume of 459 in$^3$, which is approximately 0.27 ft$^3$. This is significantly smaller than the 1 ft$^3$ specified.
3.5 MOBILITY TEST

First we needed to make sure that the motors and motor drivers worked properly, so we tested them on a breadboard. Figure 42 shows the breadboard and motors during a test.

We also verified the PWM was a nice square wave by testing the output from the motor drivers on an oscilloscope. Figures 43 through 46 show the results from the oscilloscope of the pulse width at different duty cycles.
Figure 43: Pulse width at 10% duty cycle.

Figure 44: Pulse width at 20% duty cycle.
Figure 45: Pulse width at 60% duty cycle.

Figure 46: Pulse width at 75% duty cycle.
To test for mobility we used a ruler to measure a known distance, 1 foot, and then ran the motors at full speed, which has been limited to 75% of full power, and the robot traveled the 2 feet in 1.5 seconds. This means that the robot can travel at 1.3 feet per second, which satisfies the requirement of 1 foot per second. Also, the current drawn when all 4 motors are running at full speed, 75%, is 2.1 A. The switching regulators are rated at 1 A continuous so we switched to an adjustable regulator that could handle the current draw. We used one regulator for the left side motors and one for the right side motors.

3.6 VISUAL SENSOR TEST

Figure 47: uCam and the servos that control pan/tilt.

To test that the visual sensor, shown in figure 47 already attached to the servo stack, was receiving an image we set up a test showing both robots uCam image displayed on a laptop screen.
As seen in figures 48 and 49 one can see that the uCam is picking up an image and these images are displayed from each robot's uCam onto the laptop screen.
3.7 FLAGS TEST

At start-up the robots right LED flashes a sequence of colors, green to red to blue, and the speaker plays a few notes to tell the user that start-up has completed. Also when the ‘link’ command is sent from one robot to the other the left LED turns green and when the ‘unlink’ command is sent the left LED turns red thus indicating to the user the different statuses of the robots. This was tested by turning the robot on and by sending both the ‘link’ and ‘unlink’ commands and observing that the expected results did in fact happen. Also an LCD displays general information about the robot and continuously cycles through them. Some of the information that is displayed is the IP address, the battery voltage, the number of ticks of the encoders, and debug information.

Figure 50: Right after start-up completes.

Figure 50 shows the end of the start-up sequence.
3.8 INTERFACE TEST

An on/off switch, shown in figure 51, was added to the back of the robot so that the user could easily turn the robots on and off. This was tested by flipping the switch from the off position to the on position and observing that the robot did in fact turn on. Then the switch was turned from on to off and was in fact observed to shut off the robot.

Figure 51: On/Off switch in the back of the robot in the off position.
3.9 WIRELESS RANGE TEST

To test the wireless range of the robot network we placed one 10 feet away from the access point on one side and the other 10 feet away on the other side of the access point. We then sent the ‘link’ command from one robot to the other and they did indeed become linked as shown by the LED becoming green indicating a successful link.

![Figure 52: Tape measure indicating 13 feet between the robots.](image)

Figure 52 shows that the robots can communicate at a distance of 13 feet away, which is more than the required 10 feet.

3.10 OPERATING TIME TEST

To test that the operating time we ran all four motors at full speed until they stopped due to the battery not being able to supply enough power to run them. This test yielded a time limit of 17 minutes at full speed. We also ran a test where we had both LEDs rapid cycling a color sequence, both servos moving the camera continuously, the cameras automatically updating every 2 seconds, and the rangefinders pinging continuously. This test was run until a full battery was depleted enough that it could no longer provide enough power to run everything but the motors. This test yielded an operating time of 2 and a half hours. Our goal was to have the robot run for 30 minutes. From the two tests that we performed we concluded that this was met because the motors will never be running at full speed continuously but instead move the robot a short distance and usually not at full speed.

3.11 POWER TEST

To test the batteries ability to be recharged, we recharged them after performing the operating time tests. Both batteries recharged in 2 to 3 hours, therefore the rechargeable batteries are in fact rechargeable.
### 3.12 BUDGET REQUIREMENT

Table 3: Budget Calculation

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Part Number</th>
<th>Description</th>
<th>Quantity</th>
<th>Price (Single)</th>
<th>Total Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxBotix</td>
<td>MB1300</td>
<td>XL-MaxSonar-AE0, ultrasonic sensor</td>
<td>3</td>
<td>$44.95</td>
<td>$134.85</td>
</tr>
<tr>
<td>Digi-Key</td>
<td>428-1586-ND</td>
<td>IC MCU 32K FLASH 2K SRAM 28-DIP</td>
<td>3</td>
<td>$9.24</td>
<td>$27.72</td>
</tr>
<tr>
<td>Dimension Engineering</td>
<td>DE-SW033</td>
<td>3.3V 1A Switching voltage regulator</td>
<td>1</td>
<td>$15.00</td>
<td>$15.00</td>
</tr>
<tr>
<td>Dimension Engineering</td>
<td>DE-SW050</td>
<td>5V 1A Switching voltage regulator</td>
<td>1</td>
<td>$15.00</td>
<td>$15.00</td>
</tr>
<tr>
<td>All-Battery</td>
<td>11204-01</td>
<td>Tenergy 7.2V 3000mAh Flat High Power NiMH Battery Packs w/ Tamiya Connectors for RC Cars</td>
<td>1</td>
<td>$19.99</td>
<td>$19.99</td>
</tr>
<tr>
<td>NetBurner</td>
<td>MOD5270</td>
<td>MOD5270 LC Development Kit</td>
<td>1</td>
<td>$99.00</td>
<td>$99.00</td>
</tr>
<tr>
<td>Robotshop</td>
<td>RB-Dfr-67</td>
<td>DFRobot 4WD w/ encoders</td>
<td>1</td>
<td>$71.67</td>
<td>$71.67</td>
</tr>
<tr>
<td>Pololu</td>
<td>713</td>
<td>TB6612FNG Dual Motor Driver Carrier</td>
<td>2</td>
<td>$8.45</td>
<td>$16.90</td>
</tr>
<tr>
<td>RadioShack</td>
<td>271-1317</td>
<td>470 Ω carbon-film resistors</td>
<td>2</td>
<td>$0.25</td>
<td>$0.50</td>
</tr>
<tr>
<td>RadioShack</td>
<td>272-1027</td>
<td>22μF Capacitor</td>
<td>3</td>
<td>$1.49</td>
<td>$4.47</td>
</tr>
<tr>
<td>RadioShack</td>
<td>271-1126</td>
<td>10K Ω carbon-film resistor</td>
<td>1</td>
<td>$0.25</td>
<td>$0.25</td>
</tr>
<tr>
<td>RadioShack</td>
<td>272-1044</td>
<td>220μF Capacitor</td>
<td>4</td>
<td>$1.49</td>
<td>$5.96</td>
</tr>
<tr>
<td>West Florida Components</td>
<td>WW459PD</td>
<td>120 Ω Resistor</td>
<td>4</td>
<td>$0.38</td>
<td>$1.52</td>
</tr>
<tr>
<td>West Florida Components</td>
<td>WW288</td>
<td>82 Ω Resistor</td>
<td>4</td>
<td>$0.38</td>
<td>$1.52</td>
</tr>
<tr>
<td>West Florida Components</td>
<td>CF161PD</td>
<td>1/4W 4.7K Ω 5% Carbon Film Resistor</td>
<td>2</td>
<td>$0.10</td>
<td>$0.20</td>
</tr>
<tr>
<td>West Florida Components</td>
<td>CF163PD</td>
<td>1/4W 5.6K Ω 5% Carbon Film Resistor</td>
<td>1</td>
<td>$0.10</td>
<td>$0.10</td>
</tr>
<tr>
<td>RadioShack</td>
<td>270-1009</td>
<td>3 A Fuse</td>
<td>1</td>
<td>$0.80</td>
<td>$0.80</td>
</tr>
<tr>
<td>AllBattery</td>
<td>80185</td>
<td>2 Pin Battery Side Connector</td>
<td>11</td>
<td>$1.50</td>
<td>$16.50</td>
</tr>
<tr>
<td>Digi-Key</td>
<td>813-1024-ND</td>
<td>MODULE USB-TO-SRL UART 18-DIP</td>
<td>1</td>
<td>$18.00</td>
<td>$18.00</td>
</tr>
<tr>
<td>Amazon</td>
<td>B004UAKCS6</td>
<td>IOGEAR Universal Wi-Fi N AdapterGWU627 (Black)</td>
<td>1</td>
<td>$23.95</td>
<td>$23.95</td>
</tr>
<tr>
<td>RobotShop</td>
<td>RB-Inn-10</td>
<td>Vex Hardware and Metal Kit</td>
<td>1</td>
<td>$79.99</td>
<td>$79.99</td>
</tr>
<tr>
<td>RobotShop</td>
<td>RB-Fds-15</td>
<td>4D Systems microCAM Serial JPEG Camera Module - TTL</td>
<td>1</td>
<td>$66.02</td>
<td>$66.02</td>
</tr>
<tr>
<td>RobotShop</td>
<td>RB-Lyn-77</td>
<td>Lynxmotion Pan and Tilt Kit / Aluminium (No Servos)</td>
<td>2</td>
<td>$9.95</td>
<td>$19.90</td>
</tr>
<tr>
<td>ServoDatabase</td>
<td>HS-300</td>
<td>Hitec HS-300 - Standard Servo</td>
<td>2</td>
<td>$14.95</td>
<td>$29.90</td>
</tr>
<tr>
<td>Digi-Key</td>
<td>S7093-ND</td>
<td>CONN HEADER FMAL 50PS .1&quot; DL TIN</td>
<td>3</td>
<td>$2.77</td>
<td>$8.31</td>
</tr>
</tbody>
</table>
Table 3: Budget Calculation (Continued)

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Part Number</th>
<th>Description</th>
<th>Quantity</th>
<th>Price (Single)</th>
<th>Total Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digi-Key</td>
<td>S7078-ND</td>
<td>CONN HEADER FMAL 20PS .1&quot; DL TIN</td>
<td>3</td>
<td>$1.53</td>
<td>$4.59</td>
</tr>
<tr>
<td>Digi-Key</td>
<td>438-1019-ND</td>
<td>PC BOARD 4&quot;X5&quot; UNCLAD 0.1&quot;</td>
<td>3</td>
<td>$4.68</td>
<td>$14.04</td>
</tr>
<tr>
<td>Polycase</td>
<td>DC-46PMBYT</td>
<td>DC-46P DC Series Heavy Duty Electronics Enclosure</td>
<td>1</td>
<td>$7.61</td>
<td>$7.61</td>
</tr>
<tr>
<td>All-Battery</td>
<td>1025</td>
<td>Tenergy Smart Universal Charger, NiMH/NiCD Battery Packs: 6v-12v</td>
<td>1</td>
<td>$21.99</td>
<td>$21.99</td>
</tr>
<tr>
<td>RadioShack</td>
<td>270-739</td>
<td>Chassis-Type Fuse Holder</td>
<td>1</td>
<td>$1.10</td>
<td>$1.10</td>
</tr>
<tr>
<td>Digi-Key</td>
<td>LT1085CT#PBF-ND</td>
<td>LT1083-85 Adjustable Voltage Regulator</td>
<td>2</td>
<td>$6.78</td>
<td>$13.56</td>
</tr>
<tr>
<td>Sparkfun</td>
<td>COM-00105</td>
<td>LED - RGB Clear Common Cathode</td>
<td>2</td>
<td>$1.95</td>
<td>$3.90</td>
</tr>
<tr>
<td>Sparkfun</td>
<td>LCD-09066</td>
<td>Serial Enabled 16x2 LCD - Black on Green 3.3V</td>
<td>1</td>
<td>$24.99</td>
<td>$24.99</td>
</tr>
<tr>
<td>Sparkfun</td>
<td>SEN-10643</td>
<td>AttoPilot Voltage and Current Sense Breakout - 45A</td>
<td>1</td>
<td>$19.95</td>
<td>$19.95</td>
</tr>
<tr>
<td>RadioShack</td>
<td>273-073</td>
<td>Piezo Speaker</td>
<td>1</td>
<td>$2.19</td>
<td>$2.19</td>
</tr>
<tr>
<td>Parallax</td>
<td>28017</td>
<td>Memsic 2125 Dual-axis Accelerometer</td>
<td>1</td>
<td>$29.99</td>
<td>$29.99</td>
</tr>
<tr>
<td>Sparkfun</td>
<td>SEN-07915</td>
<td>Compass Module - HMC6352</td>
<td>1</td>
<td>$34.95</td>
<td>$34.95</td>
</tr>
</tbody>
</table>

| Total Cost   |                   |                                                       |          |                | $856.88     |

From Table 3 the final total cost of each robot is $856.88 which is less than the $1,000 goal.
SECTION 4: EVALUATION AND RECOMMENDATIONS
4.1 RECOMMENDATIONS

4.1.1 PSoC Programming
After wiring the two circuit boards we realized that in order to reprogram any of the PSoCs that one has to disassemble the robot. The disassembly process consists of removing the box from the robot and detaching the first circuit board to get to the PSoC programming connection. If one would just add another PSoC programming connection to the first board and add a jumper cable to it, then one could program the PSoC by only detaching the box.

4.1.2 Camera
The 4D Systems uCAM Serial JPEG Camera Module that we decided to use for the visual sensor could have been improved upon and became closer to the original CMUCam3 that was to be used by using the SRV-1 Blackfin camera. The Blackfin camera is a closer match to the CMUCam3 that we were originally required to use but due to time constraints and the unavailability of the CMUCam3 we used the uCam instead.

Another alternative to the uCam that was used is the CMUCam4. The CMUCam3 was discontinued and the CMUCam4 replaced it. The CMUCam4 was not available until rather late in the build process therefore we were unable to utilize this camera.

4.1.3 Motors and Motor Drivers
The motors or motor drivers could be the source of a problem that arose late in the build. It seems that after running the motors to propel the robot that it causes a loss of communication and the robot locks up. We tried to determine the exact reason why this happens but were unable to due to the fact that the issue did not arise until late into the build. More testing and research needs to be put forth in this area to try and rectify the problem.

4.1.4 I^2C Bus
The I^2C communication drops out and causes the robots to lock up and not respond to any commands issued to it resulting in a manual reboot required to get the robot back up and running. We tried to troubleshoot as to why this was happening but did not resolve the problem in the end. One solution we thought might help was to rewrite some of the code to detect that the I^2C communication is not working and to do a soft reset of the I^2C bus to get communications going again. Also, some work needs to be done on the I^2C communication code to make it less prone to this problem.

One reason we thought this issue could be happening is because of the motor issue again. The motor drivers could be causing electromagnetic interference which is disrupting the I^2C communications. Therefore it might help to develop some way of shielding the rest of the system from the motor drivers.
4.1.5 Additional Power Requirements
After all the additions that were made to the robots, the power requirements are greater than before. The team estimated that the both LEDs would add 30mA each, the LCD would add 60mA, the 2 servos would add 450mA each, the compass would add 1mA, the accelerometer would add 4mA, the speaker would add 2mA, and the wireless module would add 380mA. The LCD and both servos are supplied 3.3 V. The LEDs, compass, accelerometer, speaker, and wireless module are all supplied 5V. If one assumes the worst case scenario of all additions having a 100% duty cycle then 5403mW are added to the previous total, which in turn leads to the robot requiring 1910.07mA. If one needs to run everything for an hour that leads to needing a battery rated at 1910.07mAh. With the current 3000mAh battery everything should be fine, but, as we found out, the motors draw more current than what they were rated at. With this being said a higher mAh battery would probably be better especially if more sensors, LEDs, etc. are added.
5.1 PROGRAMMING
The Clickybot architecture currently consists of one Netburner MOD5270 and three PSoC CY8C29466 processors. The Netburner module is programmed with the NBEclipse development environment. And the PSoC are programmed with the PSoC Designer 5.1 environment.

5.1.1 NetBurner Programming
Run the NBEclipse program and select the C:\_proj\sd\nburn\ClickyBot workspace. (To send new code to the robots, be sure to run the NBEclipse program as Administrator.) This should display the project for editing. Make any changes and save the source file. This will automatically compile the code and display any error if Build Automatically is selected from the project menu.

Once the program is compiled it can be downloaded to the robots via WiFi, Ethernet, or serial port. If the robots are connected WiFi or Ethernet, simply “Run” the application. Under “run configurations” you can change the IP address where the program will download. This will need to be adjusted to send the program to the correct robot. Sometimes during development an error in the program can cause the Netburner to “freeze” and not accept a new program over TCP/IP. In this case you will need to use the MTTY program that comes with the Netburner development environment and connect the robot to the computer through USB. Start the MTTY terminal program and power on the robot. You should see instructions to press the “A” key within 2 seconds. This will put the robot into a low level monitor mode. To send the new program in this mode, type FLA<enter> then choose transfer file option form the menu. Choose the C:\_proj\sd\nburn\ClickyBot\ClickyBot\Release\ClickyBot_APP.s19 file for upload.

5.1.2 PSoC Programming
Run the PSoC Designer 5.1 program and open the TiltSensor, Motors, or Servo project. Here you can make changes to the main.cpp for the chosen module or adjust the peripherals configured in the “chip” editor. Save the files you changed and choose Generate from the Build menu. If the changes compiled correctly you can send the new firmware to the PSoC chip. To do this you will need to attach the PSoC MiniProg to the PC with the USB cable and then to the appropriate programming port on the robot. Each of the 3 PSoC on the robot has a separate port. Two of these are on the first board that is accessible by lifting the case. The third, for the TiltSensor, is located on the board with the netburner module. Select Program Part from the Program menu. Change the Acquire Mode to Reset. Be certain that you DO NOT press the power button on this screen. The chip must be powered by the robots power supply and not the programmer. Powering from the programmer could damage the components on the robot! Press the program button at the bottom of this screen when the connected indicator is green and the new firmware program should download.
5.2 SOFTWARE OVERVIEW
We used the Netburner module as our master CPU. This processor uses the µC/OS RTOS (Real Time Operating System.) We’ve broken our code into 8 separate tasks or threads that run within the µC/OS environmment.

- **Main/Message Loop task**
  - Handles the display of status information on the robot’s LCD display.
- **I2CMonitor Task**
  - Handle communication to all I²C devices.
- **HTTP Task**
  - Handles requests from web browsers for image and sensor data.
- **Command Task**
  - A command line interface accessible via telnet or directly through the serial link.
- **Camera Task**
  - Handle communication with the uCam camera and downloads images when available.
- **Sequencer Task**
  - Runs a higher level robot program sequence.
- **TCPServer Task**
  - Receives command and communicates data to other robots or PC programs
- **Link Task**
  - Establishes communication to other robots and issues direct commands to them based on the current sequencer program or via proxy commands.

5.2.1 Main/Message Loop task
The program starts at the UserMain function. This first initializes the robot and sets up the other tasks. After initialization, this becomes the message loop task. Every 3 seconds this task increments the message number calls the GetMessage() function to get two lines of text for that message ID. These are then sent to the Serial LCD of the robot.

5.2.2 I2CMonitor Task
This CPU communicates to and controls the PSoC co-processors via a common I²C bus architecture. Each device has been configured with a unique 7-bit address. This task periodically (usually 20 times a second) addresses each device and downloads the latest register file from each. Desired changes are then made to this copy of the register file and specific portions are then uploaded back to the device if needed. In this way there is conceptually a small area of shared memory between the main CPU and each PSoC device.

Certain registers can be manipulated directly in this way like the current colors for the LEDs or the desired position of the servos. The PSoCs then monitor changes in their own register files and affect the changes as they come in. An example of this would be the servo controller which simply reads the servo position value from its register file, translates this into an appropriate value for the PWM peripheral with the chip, and updates the duty cycle for the desired position.
This method does not work well for all communication, so we have also included a command injection system within the registry file for each PSoC. This consists of an optional 4-byte parameter array followed by command id byte. It is important that the command byte comes after the parameters to ensure that the parameters are written to the file first. This way, when the command is read by the PSoC code there is no chance that the parameter have not yet been written for this command. The PSoC communicates that the command has been received by setting the command byte back to zero which is the idle state. Figure 53 shows an example of the process on both sides of the I^2C link.

Figure 53: Example of the I^2C Link process from the master and slave.
5.2.3 HTTP Task
The HTTP task is set up using the SetNewGetHandler() function. This sets a hook in the operating system
to redirect HTTP GET messages to the MyDoGet() function. Here we have code to intercept incoming
requests for the cam.jpg file and send out the current jpeg image buffer. This way the images from the
robots can be easily viewed with just a simple web browser. There could also be specialized programs
written to further process the images on the PC or to send them directly to other robots.

5.2.4 Command Task
The Command task is initialized in our Cmd_Init() function. This sets up the Command Line interpreter
via another OS hook and connects it to both the physical USB serial port and virtually by listening to the
Telnet port 23. The Command task can handle one serial and up to 5 simultaneous connections via
telnet. Each command is directed to the Cmd_Execute() function where it is interpreted and executed.

5.2.5 Camera Task
The Camera is connected directly to the Netburner by a TTL serial link. The Camera task initializes the
camera and downloads images to one of two alternating buffers. One buffer is flagged as the current
buffer and this one is made available for further processing. The other buffer is set aside and used to
actively download the available image from the camera. Once the image has been completely
downloaded, this buffer is flagged as the current buffer and future request for image data will get the
freshly downloaded image.

5.2.6 Sequencer Task
The Sequencer is used to control overall robot behavior. Here is where higher level processing should
occur. The sequencer issues commands and interprets the data made available by the other tasks. In
our example, we use the sequencer to change the color of one of the LEDs. More complex sequences
can be encoded here for future projects. Ideally, this would be a higher level language interpreter that
would abstract much of the robots internal workings.

5.2.7 TCPServer Task
This serves as a command interpreter for inter-robot communications. A listener is set on TCP/IP port
6502 and other robots or specialized PC programs can connect to this port and send commands or read
sensor buffers directly.

5.2.8 Link Task
The Link task works with the TCPServer task of another robot to open a communications channel
between the two.
CONCLUSIONS

In conclusion, we have performed several steps in the development of the Low-Cost Autonomous Multi-Robot Network project. We have defined the requirements, brainstormed possibilities for meeting those requirements, analyzed each of the modules, and chose the final design for the project. We then looked at the final design and analyzed it in a meticulous manner. Next, we proceeded to build the design chosen to fulfill the project requirements. After starting the testing and building of the final design we realized some things that would not accomplish the task we had set forth so changes and additions were made to fulfill the requirements. After each robot was built and operating the way in which we designed, we then demonstrated that each requirement was met. Finally, we suggested some improvements and some further testing and research for anyone who will be working on the project in the future.
REFERENCES


APPENDIX

A.1 CODE FOR NETBURNER

A.1.1 Main.cpp
/* Main.cpp
* ECE 405 - 406 Senior Design Project
* Team Members: Michael DeMange
* Rodrigo Tamashiro
* William Westrick
*/

#include "predef.h"
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpclient.h>
#include <smarttrap.h>
#include <taskmon.h>
#include <NetworkDebug.h>
#include <serial.h>
#include <i2cmaster.h>
#include <ntime.h>
#include <netinterface.h>
#include <dns.h>
#include <ethernet.h>
#include <ucos.h>
// #include <ftp.h>
#include <tcp.h>
#include <bsp.h>

#include "cmd.h"  // command processor
#include "Camera.h" // camera task

#define CLEARSCREEN putchar(0xFE);putchar(0x01)
#define LINE1 putchar(0xFE);putchar(0x80)
#define LINE2 putchar(0xFE);putchar(0xC0)
#define ROM 0x40
#define COMPASS 0x21
#define TILTSENSOR 0x01
//#define MOTOR 0x02
#define MOTOR 0x0D
// #define RANGESERVO 0x03
#define RANGESERVO 0x1B
#define NUNCHUCK 0x52

/*/CameraCommands...
#define CAM_INIT 0x01
enum TiltCommands {tilt_cmdIdle = 0, tilt_cmdPlayNote = 'P'};
enum Notes {noteRest = 0, noteC = 1, noteCS = 2, noteD = 3, noteDS = 4,
noteE = 5, noteF = 6, noteFS = 7, noteG = 8, noteGS = 9, noteA = 10, noteAS = 11, noteB = 12};
BYTE LED[2][3] = { {0,0,0},{0,0,0} }; // led colors.
BYTE Song[100][3] = { {noteG,5,16}, {noteC,5,8}, {noteE,5,8}, {noteE,5,8}, {0,0,0} }; // song buffer 100 note in note, octave, duration format.
int SongNote = 0; // current note to play.

extern "C" {
void UserMain(void * pd);
}

// Custom GET handler
int MyDoGet( int sock, PSTR url, PSTR rxBuffer );
static http_gethandler *oldhand; // stored old handler for chaining if we dont handle the request.
int GetMessage(int msg, char *line1, char *line2);

//enum msgtype
{msgTITLE=0, msgMAC, msgETHER, msgIP, msgMASK, msgDNS, msgGATEWAY, msgI2CTest, msgCOMPASS, msgLASTMSG,
 msgSERVER, msgTESTRESULT, msgTESTNEXTTIME, msgUPTIME, msgEXTERNALIP, msgHBLATENCY,
 msgWATCHDOG};
enum msgtype {msgTITLE=0, msgMAC, msgIP, msgCOMPASS, msgTILT, msgCAMERA,
 msgRANGE, msgMOTOR, msgBATTERY, msgLASTMSG,
 msgETHER, msgMASK, msgDNS, msgGATEWAY,
 msgSERVER, msgTESTRESULT, msgTESTNEXTTIME, msgUPTIME, msgEXTERNALIP, msgHBLATENCY,
 msgWATCHDOG, msgNOTE, msgI2CTest};

struct TiltRegStruct {  // Structure from PSoC
    BYTE bRamRegLength;
    BYTE bAddress;
    BYTE par[4];
    BYTE bCmd;
    BYTE bCmdID;
    BYTE bCompletedCmdID;
    BYTE bLed1_R;
    BYTE bLed1_G;
    BYTE bLed1_B;
    BYTE bLed2_R;
    BYTE bLed2_G;
    BYTE bLed2_B;
    WORD wXTilt;
    WORD wYTilt;
    BYTE bTiltCheck;
    BYTE bSpeakerStat;
    BYTE bxxx;
}__attribute__((packed)) TiltRegs; // note: we need the "packed" so that the
compiler will not pad the elements on WORD and DWORD boundaries. This make
the structure appear exactly as it does in the PSoC.

struct ServoRegStruct { // Structure from PSoC
    BYTE bAddress;
    BYTE bPar[4];
    BYTE bCmd;
    char bTopServo;
    char bBottomServo;
    WORD wSensorRight;
    WORD wSensorMiddle;
    WORD wSensorLeft;
}__attribute__((packed)) ServoRegs;

struct MotorRegStruct { // Structure from PSoC
    BYTE bAddress;
    BYTE bPar[4];
    BYTE bCmd;
    WORD wREncoder;
    WORD wLEncoder;
    BYTE bEncoderCheck;
    BYTE bRTick;
    BYTE bLTick;
    WORD wVoltage;
    WORD wCurrent;
}__attribute__((packed)) MotorRegs;
WORD RangeLeft = 0;
WORD RangeRight = 0;
WORD RangeMiddle = 0;

#define TCP_RX_BUFSIZE (4096)
#define TCP_PORT (6502)
char TCPRXBuffer[TCP_RX_BUFSIZE]; // receive buffer for TCP connection

int tilt_debug = 0;
int range_debug = 0;

// Tasks...
// Priority ranges from MAIN_PRIO-4 to MAIN_PRIO-1, MAIN_PRIO+1 to
// MAIN_PRIO+5, MAIN_PRIO+7 to MAIN_PRIO+12 (from highest to lowest)
#define Command_PRIO (MAIN_PRIO-4)
DWORD I2CMonitorTaskStack[USER_TASK_STK_SIZE];
void I2CMonitorTask(void *pdata);
#define I2CMonitorTask_PRIO (MAIN_PRIO-1)

DWORD CameraTaskStack[USER_TASK_STK_SIZE];
void CameraTask(void *pdata);
#define CameraTask_PRIO (MAIN_PRIO-2)

DWORD SequenceTaskStack[USER_TASK_STK_SIZE];
void SequenceTask(void *pdata);
#define SequenceTask_PRIO (MAIN_PRIO+1)

DWORD TCPServerTaskStack[USER_TASK_STK_SIZE];
void TCPServerTask(void *pdata);
#define TCPServerTask_PRIO (MAIN_PRIO-3)

DWORD LinkTaskStack[USER_TASK_STK_SIZE];
void LinkTask(void *pdata);
#define LinkTask_PRIO (MAIN_PRIO+2)

const char * AppName="ClickyBot v1.0a";

char I2CBuf[40];

char NunchuckBuf[20];

int CompassHeading = 1234;
int XTilt = -1;
int YTilt = -1;

void UserMain(void * pd) {
    int msg = msgTITLE; // the first message.
    char line1[21];
    char line2[21];

    InitializeStack();
    if (EthernetIP == 0) GetDHCPAddress();
OSChangePrio(MAIN_PRIO);
EnableAutoUpdate();

StartHTTP();
EnableTaskMonitor();

#ifndef _DEBUG
EnableSmartTraps();
#endif

#ifdef _DEBUG
InitializeNetworkGDB_and_Wait();
#endif

InterfaceBlock *ib = GetInterFaceBlock(0);
//DWORD UniqueID = *((DWORD*)((BYTE*)&ib->theMac)+2));
WORD UniqueID = ib->theMac.phywadr[2];
BYTE* ipb = (BYTE*)&other_robot;
ipb[0] = 192;
ipb[1] = 168;
ipb[2] = 0;
switch(UniqueID){
case 0x0C15:
    Robot_ID = 101;
    strcpy(Robot_Name,"Fred");
    ipb[3] = 102;
    break;
case 0xE15F:
    Robot_ID = 102;
    strcpy(Robot_Name,"Ginger");
    ipb[3] = 101;
    break;
case 0x150C:
    Robot_ID = 103;
    strcpy(Robot_Name,"Clyde");
    break;
}

range_debug = UniqueID;

// register our handler...
oldhand = SetNewGetHandler( MyDoGet );

// Setup Serial LCD as the main stdio output...
SerialClose(1);
int fd = OpenSerial(1,9600,1,8,eParityNone);
ReplaceStdio(1,fd); // direct stdio to LCD

OSTimeDly(20);

// init the command system.
Cmd_Init(Command_PRIO);

// Say Hello...
CLEARSCREEN;
LINE1; *printf("%s" Started:”, Robot_Name);
LINE2; *printf("Initializing");
for(int i=0; i<4; i++){
   *printf(".");
   OSTimeDly(TICKS_PER_SECOND*2/3);
}
// Start Tasks...
OSTaskCreate(CameraTask,
    NULL,
    *(void*)CameraTaskStack, USER_TASK_STK_SIZE, CameraTaskStack,
   CameraTask_PRIO);

OSTaskCreate(SequenceTask, 
    NULL,
    *(void*)SequenceTaskStack, USER_TASK_STK_SIZE, SequenceTaskStack,
    SequenceTask_PRIO);

OSTaskCreate(I2CMonitorTask,
    NULL,
    *(void*)I2CMonitorTaskStack, USER_TASK_STK_SIZE, I2CMonitorTaskStack,
    I2CMonitorTask_PRIO);

OSTaskCreate(TCPServerTask,
    NULL,
    *(void*)TCPServerTaskStack, USER_TASK_STK_SIZE, TCPServerTaskStack,
    TCPServerTask_PRIO);

OSTaskCreate(LinkTask,
    NULL,
    *(void*)LinkTaskStack, USER_TASK_STK_SIZE, LinkTaskStack,
    LinkTask_PRIO);

msg = msgMOTOR;
// Main Message Loop...
while (1) {
    GetMessage(msg, line1, line2);
    CLEARSCREEN;
    LINE1; *printf(line1);
    LINE2; *printf(line2);
    OSTimeDly(TICKS_PER_SECOND*2);
    msg++;
    //msg %= msgLASTMSG; // msgLASTMSG is a flag used to wrap back to the first valid message.
    if(msg==msgLASTMSG) msg = msgIP;
    // if(msg==msgCOMPASS) msg = msgIP;
    // if(msg==msgBATTERY) msg = msgIP;
}
/* Process our own Get Requests to respond to the GIF image request */

int MyDoGet( int sock, PSTR url, PSTR rxBuffer )
{
    static char buf[255];
    static char* message = "Hello Clickybot!";
    if(strlen(url)){
        if(httpstricmp(url,"TEST.TXT")){
            sprintf(buf,"HTTP/1.0 200 OK
Pragma: no-cache
MIME-version: 1.0
Content-Type: image/jpeg
Content-length: %d

",jpg_len );
            sprintf(buf,"HTTP/1.0 200 OK
Pragma: no-cache
MIME-version: 1.0
Content-Type: text/plain
Content-length: %d

",strlen(message) );
            writestring(sock, buf);
            //Now send the text image
            writeall(sock,message,strlen(message));

            return 1;
        }
        if(httpstricmp(url,"CAM.JPG")){
            // NOTE: this code is technically not safe. Normally we
            // would need a semaphore to prevent the camera task from overwriting the buffer
            // we are using while we are sending it out. With the timing we have however,
            // this should never occur.
            if(JPegBufferSize && JPegBuffer){
                sprintf(buf,"HTTP/1.0 200 OK
Pragma: no-cache
MIME-version: 1.0
Content-Type: image/jpeg
Content-length: %d

",JPegBufferSize );
                writestring(sock, buf);
                //Now send the JPG image
                writeall(sock,(char*)JPegBuffer,JPegBufferSize);
            }else{
                static char* error_message = "JPeg image not ready!";
                sprintf(buf,"HTTP/1.0 200 OK
Pragma: no-cache
MIME-version: 1.0
Content-Type: text/plain
Content-length: %d

",strlen(error_message) );
                writestring(sock, buf);
                //Now send the text image
                writeall(sock,error_message,strlen(error_message));

            }
            return 1;
        }
    }
    return ( *oldhand ) ( sock, url, rxBuffer );
}

//void RegisterPost()
//{
//   SetNewPostHandler( MyDoPost );
//   oldhand = SetNewGetHandler( MyDoGet );
//}
int GetMessage(int msg, char *line1, char *line2){
    int ret=0;
    switch(msg){
    case msgTITLE:
        {
            strcpy(line1,AppName);
            strcpy(line2,"IPFW Spring 2012");
        }
        break;
    case msgETHER:
        {
            strcpy(line1,"Ethernet Adapter");
            if(EtherLink()){
                strcpy(line2,"Connected 10");
                if(EtherSpeed100()) strcat(line2,"0");
                strcat(line2,"Mbps");
            }else{
                strcpy(line2,"Not Connected!");
                ret = -1;
            }
        }
        break;
    case msgIP:
        {
            InterfaceBlock *ib = GetInterFaceBlock(0);
            strcpy(line1,"IP Address:");
            sprintf(line2,"%d.%d.%d.%d",((unsigned char*)(ib->netIP))[0],((unsigned char*)(ib->netIP))[1],((unsigned char*)(ib->netIP))[2],((unsigned char*)(ib->netIP))[3]);
        }
        break;
    case msgMAC:
        {
            char tmp[8];
            InterfaceBlock *ib = GetInterFaceBlock(0);
            strcpy(line1,"MAC Address:");
            //ShowMac(&(ib->theMac));
            line2[0] = '\0';
            for(int n=0; n<6; n++){
                sprintf(tmp,"%02X",((BYTE*)(ib->theMac))[n]);
                strcat(line2,tmp);
            }
        }
        break;
    case msgMASK:
        {
            InterfaceBlock *ib = GetInterFaceBlock(0);
            strcpy(line1,"Address Mask:");
            sprintf(line2,"%d.%d.%d.%d",((unsigned char*)(ib->netIpMask))[0],((unsigned char*)(ib->netIpMask))[1],((unsigned char*)(ib->netIpMask))[2],((unsigned char*)(ib->netIpMask))[3]);
        }
        break;
    }
case msgDNS:
{
    InterfaceBlock *ib = GetInterFaceBlock(0);
    strcpy(line1, "Gateway:");
    sprintf(line2, "%d.%d.%d.%d", ((unsigned char*)&(ib->netIpGate))[0], ((unsigned char*)&(ib->netIpGate))[1], ((unsigned char*)&(ib->netIpGate))[2], ((unsigned char*)&(ib->netIpGate))[3]);
    break;
}

case msgGATEWAY:
{
    InterfaceBlock *ib = GetInterFaceBlock(0);
    strcpy(line1, "DNS:");
    sprintf(line2, "%d.%d.%d.%d", ((unsigned char*)&(ib->netDNS))[0], ((unsigned char*)&(ib->netDNS))[1], ((unsigned char*)&(ib->netDNS))[2], ((unsigned char*)&(ib->netDNS))[3]);
    break;
}

case msgI2CTEST:
{
    strcpy(line1, "I2C Message");
    strcpy(line2, I2CBuf);
    break;
}

case msgCOMPASS:
{
    strcpy(line1, "Compass Heading:");
    if (CompassHeading<0){
        sprintf(line2, "Unknown Error=%d", CompassHeading);
    } else{
        sprintf(line2, "%d.%d Degrees", CompassHeading/10, CompassHeading%10);
    }
    break;
}

case msgTILT:
{
    sprintf(line1, "X Tilt=%d", XTilt);
    sprintf(line2, "Y Tilt=%d", YTilt);
    break;
}

case msgCAMERA:
{
    sprintf(line1, "Camera:%s", strCameraStatus);
    sprintf(line2, "debug=%d", CameraSyncCount);
    break;
}

case msgNOTE:
{
    sprintf(line1, "Note:%d", SongNote);
    sscanf(line1,"Voltage:%4.2f V",(double)MotorRegs.wVoltage*0.0013+4.911);
sprintf(line2,"debug=%d",MotorRegs.wCurrent);
    break;

    //case msgNUNCHUCK:
    //  {
    //    sprintf(line1,"joy:%d,%d",joy_x_axis,joy_y_axis);
    //    sprintf(line2,"debug");
    //  }
    //  break;

    //case msgTESTRESULT:
    //  if(STLastTime>0){
    //    strcpy(line1,"Current Speed:");
    //    sprintf(line2,"%d kByte/s",(STLastSize)/(STLastTime));
    //    break;
    //  }
    //  }
    //  
    //case msgTESTNEXTTIME:
    //  {
    //    strcpy(line1,"Next Test:");
    //    if(STNext<=120){
    //      sprintf(line2,"%d sec",STNext);
    //    }else{
    //      sprintf(line2,"%d min",STNext/60);
    //    }
    //  }
    //  break;

    //case msgWATCHDOG:
    //  {
    //    strcpy(line1,"Watch Dog Timer:");
    //  }
    //  break;
if(WatchDogTimer<=120){
    sprintf(line2,"%d sec",WatchDogTimer);
} else{
    sprintf(line2,"%d min",WatchDogTimer/60);
}
break;

case msgSERVER:
{
    strcpy(line1,"ZOOM Server:");
    switch(dns_result){
    case DNS_OK:
        //ShowIP(WEB_Server_IP);
        sprintf(line2,"%.3rd.%.3rd.%.3rd",((unsigned char*)&(WEB_Server_IP))[0],((unsigned char*)&(WEB_Server_IP))[1],((unsigned char*)&(WEB_Server_IP))[2],((unsigned char*)&(WEB_Server_IP))[3]);
        break;
    case DNS_TIMEOUT:
        strcpy(line2,"DNS Timeout Err");
        break;
    case DNS_NOSUCHNAME:
        strcpy(line2,"DNS No Such Name");
        break;
    case DNS_ERR:
        default:
        strcpy(line2,"DNS Error");
        break;
    }
}
break;

case msgUPTIME:
{
    if(uptime>=0){
        strcpy(line1,"Connected for:");
        if(uptime<=120) sprintf(line2,"%d sec",uptime);
        else if(uptime<=60*60*2) sprintf(line2,"%d min",uptime/60);
        else if(uptime<=60*60*24*2) sprintf(line2,"%d hours",uptime/60/60);
        else sprintf(line2,"%d days",uptime/60/60/24);
    } else{
        strcpy(line1,"Not Connected!");
        strcpy(line2,"Last HB Failed!");
    }
}
break;

case msgEXTERNALIP:
{
    strcpy(line1,"External IP:");
    strcpy(line2,external_IP);
}
break;
case msgHBLATENCY:
    {
        strcpy(line1,"HB Latency:");
        if(HBLat>=0){
            sprintf(line2,"%d mSec",HBLat);
        }else{
            strcpy(line2,"???");
        }
    }
    break;

default:
    strcpy(line1,"Unknown Message:");
    sprintf(line2,"MSG #%d",msg);
    break;

    return ret;
}

int PSoc_Send(BYTE addr, BYTE* buffer, int len){
    int result = I2CStart(addr, I2C_START_WRITE, 1);
}

//TiltSensor, Leds and Speaker...
void ProcessTilt(){
    BYTE buf[sizeof(TiltRegStruct)+1];
    //BYTE cmd = 0;
    //buf[0] = &TiltRegs.bCmd - &TiltRegs; // register location of bCmd
    buf[0] = 0x00; // prepare to read the entire register
    I2CSendBuf(TILTSENSOR, buf, 1);
    tilt_debug = I2CReadBuf(TILTSENSOR, (BYTE*)&TiltRegs, sizeof(TiltRegStruct)); // Read registers...
    XTilt = (int)TiltRegs.wXTilt;
    YTilt = (int)TiltRegs.wYTilt;
    //TiltRegs.wXTilt = 123;
    // write LED values...
    buf[0] = &TiltRegs.blLed1_R - (BYTE*)&TiltRegs; // register location of blLed1_R
    for(int n=0; n<6; n++){
        buf[n+1] = ((BYTE*)LED)[n]; // copy values for LEDs.
    }
    I2CSendBuf( TILTSENSOR, buf, 7); // write LED values...

    if(TiltRegs.bCmd == tilt_cmdIdle && Song[SongNote][2] && SongNote>=0 &&
TiltRegs.bSpeakerStat==0){ // if the is no command in the buffer and there is
    a note to play...
        TiltRegs.bAddress = (BYTE*)&TiltRegs.par[0] - (BYTE*)&TiltRegs;
        TiltRegs.par[0] = Song[SongNote][0];
        TiltRegs.par[1] = Song[SongNote][1];
    }
TiltRegs.par[2] = Song[SongNote][2];
TiltRegs.bCmd = tilt_cmdPlayNote;
I2CSendBuf( TILTSENSOR, (BYTE*) &TiltRegs.bAddress, 6); // send the note
SongNote++; // prepare for next note.
// LED[1][1] = 0x40;

// if(TiltRegs.bSpeakerStat){
//   LED[1][0] = 0x00;
//   LED[1][1] = 0x40;
//   LED[1][2] = 0x00;
// }else{
//   LED[1][0] = 0x40;
//   LED[1][1] = 0x00;
//   LED[1][2] = 0x00;
// }
// SongNote = 0;
//}
// TiltRegs.bAddress = &TiltRegs.bCmd - &TiltRegs; // register location of bCmd
// I2CSendBuf( TILTSENSOR, (BYTE*) &TiltRegs.bAddress, 1);

int ServoTop = 0;
int ServoBottom = 0;

void ProcessRangeServo(){
  static int old_top = -999;
  static int old_bottom = -999;
  static int range_sequence = 0;

  BYTE buf[sizeof(ServoRegStruct)+1];
  //BYTE cmd = 0;
  // buf[0] = &TiltRegs.bCmd - &TiltRegs; // register location of bCmd
  buf[0] = 0x00; // prepare to read the entire register
  I2SendBuf(RANGESERVO, buf, 1);
  I2CReadBuf(RANGESERVO, (BYTE*) &ServoRegs, sizeof(ServoRegStruct)); // Read registers...

  // XTilt = (int) TiltRegs.wXTilt;
  // YTilt = (int) TiltRegs.wYTilt;
  if(ServoRegs.wSensorLeft) RangeLeft = ServoRegs.wSensorLeft;
  if(ServoRegs.wSensorRight) RangeRight = ServoRegs.wSensorRight;
  if(ServoRegs.wSensorMiddle) RangeMiddle = ServoRegs.wSensorMiddle;

  // if(ServoRegs.bCmd == 0 && old_bottom != ServoBottom){
  //   ServoRegs.bAddress = (BYTE*) &ServoRegs.bPar[0] -
  //   (BYTE*) &ServoRegs; // beggining of par/cmd buffer
  //   old_bottom = ServoBottom;
  //   ServoRegs.bCmd = 'S'; // servo command
  //   ServoRegs.bPar[0] = 'b'; // bottom servo
  //   ServoRegs.bPar[1] = (BYTE) (signed char) ServoBottom; // location
  // to move
if(old_bottom!=ServoBottom || old_top!=ServoTop){
    old_bottom=ServoBottom;
    old_top=ServoTop;
    buf[0] = (BYTE*)ServoRegs.bTopServo - (BYTE*)&ServoRegs;
    buf[1] = (BYTE)ServoTop;
    buf[2] = (BYTE)ServoBottom;
    I2CSendBuf( RANGESERVO, buf, 3); // send the command
}

ServoRegs.bAddress = (BYTE*)&ServoRegs.bPar[0] - (BYTE*)&ServoRegs; // Beginning of par/cmd buffer
ServoRegs.bPar[0] = 0;
switch(range_sequence){
    case 0:
        ServoRegs.bPar[0] = 'm'; // middle sensor
        break;
    case 1:
        ServoRegs.bPar[0] = 'r'; // right sensor
        break;
    case 2:
        ServoRegs.bPar[0] = 'l'; // left sensor
        break;
    default:
        break;
}

if(ServoRegs.bPar[0]){  
    ServoRegs.bCmd = 'R';
    I2CSendBuf( RANGESERVO, (BYTE*)&ServoRegs.bAddress, 6); // send the command
}

range_sequence++;
if(range_sequence>=10) range_sequence=0; // range every 1/2 second or so.

//ServoBottom+=2;
//if(ServoBottom>90) ServoBottom = -90;
//ServoTop++;
//if(ServoTop>90) ServoTop = -30;

char MotorLeft = 0;
char MotorRight = 0;
BYTE MotorDist = 0;

void ProcessMotor(void){
    static char old_MotorLeft = -123;
    static char old_MotorRight = -123;

    char cmd = 0;
    cmd = 0x00; // prepare to read the entire register
I2CSendBuf(MOTOR, (BYTE*)&cmd, 1);
I2CReadBuf(MOTOR, (BYTE*)&MotorRegs, sizeof(MotorRegStruct)); // Read registers...

//Copy sensor values here

if (MotorRegs.bCmd==0 && (old_MotorLeft!= MotorLeft || old_MotorRight != MotorRight)){
  old_MotorLeft = MotorLeft;
  old_MotorRight = MotorRight;
  MotorRegs.bAddress = (BYTE*)&MotorRegs.bPar[0] -
  (BYTE*)&MotorRegs; // Beginning of par/cmd buffer
  MotorRegs.bPar[0] = MotorLeft;
  MotorRegs.bCmd = 'M';
  I2CSendBuf( MOTOR, (BYTE*)&MotorRegs.bAddress, 6); // send the command
}

//MotorLeft+=1;
//if(MotorLeft>=100) MotorLeft=0;
}

void I2CMonitorTask(void *pdata)
{
  static char CompassBuff[6];
  I2CInit();

  OSTimeDly(20); // just in case. not sure these are needed.
  I2CBuf[0] = 0x00; // data address
  I2CSendBuf( TILTSENSOR|ROM, (BYTE*)I2CBuf, 1); // send address to psoc rom address
  OSTimeDly(1);
  I2CReadBuf( TILTSENSOR|ROM, (BYTE*)I2CBuf, 15);
  //I2CBuf[0] = 0x00; // data address
  //I2CSendBuf( TILTSENSOR, (BYTE*)I2CBuf, 1); // send address to psoc ram reg address
  //OSTimeDly(1);
  //I2CReadBuf( TILTSENSOR, (BYTE*)TiltRegs, 2);

  //CompassBuff[0] = 'A';
  //BYTE result =
  I2CSendBuf( COMPASS, (BYTE*)"A", 1);

  //CompassHeading = 0 - (int)result;

  while(1){
    OSTimeDly(1);
    I2CReadBuf( COMPASS, (BYTE*)CompassBuff, 2);
    CompassHeading = ((int)(BYTE)CompassBuff[0])*256 +
    (int)(BYTE)CompassBuff[1];
I2CSendBuf( COMPASS, (BYTE*)"A", 1);

ProcessTilt();

ProcessRangeServo();

ProcessMotor();

    //Nunchuck
    /*
    NunchuckBuf[0] = 0x40;
    NunchuckBuf[1] = 0x00;
    I2CSendBuf( NUNCHUCK/2, (BYTE*)NunchuckBuf, 2);
    OSTimeDly(2);
    I2CReadBuf( NUNCHUCK/2, (BYTE*)NunchuckBuf, 6);
    for(int n=0; n<6; n++){
        NunchuckBuf[n] ^= 0x17;
        NunchuckBuf[n] += 0x17;
    }
    */

    joy_x_axis = NunchuckBuf[0];
    joy_y_axis = NunchuckBuf[1];
    accel_x_axis = NunchuckBuf[2]*4; // * 2 * 2;
    accel_y_axis = NunchuckBuf[3]*4; // * 2 * 2;
    accel_z_axis = NunchuckBuf[4]*4; // * 2 * 2;
    z_button = (NunchuckBuf[5] >> 0) & 1;
    c_button = (NunchuckBuf[5] >> 1) & 1;
3) & 1);
5) & 1);
7) & 1);
    */

void SequenceTask(void *pdata) {
    OSTimeDly(20);

    int loop = 0;
    MotorLeft = 0;
    MotorRight = 0;

    while(1){
        OSTimeDly(5);
        LED[0][0] = 0x00;
        LED[0][1] = 0x40;
        LED[0][2] = 0x00;
        OSTimeDly(5);
        LED[0][0] = 0x40;
        LED[0][1] = 0x00;
void IPtoString(IPADDR ia, char *s)
{
    PBYTE ipb = (PBYTE)&ia;
    sprintf(s,
        "%d.%d.%d.%d", (int)ipb[0], (int)ipb[1], (int)ipb[2], (int)ipb[3]);
}

void Test_Execute(const char *command)
{
    char temp[20];
    if (command[0])
    {
        sprintf(fp, "# %s Sent Cmd[%s]", pData, command);
    }

    // Close the connection if we receive Logout
    if (stricmp(command, "logout") == 0)
    {
    }

    else if (stricmp(TCPRXBuffer, "reboot") == 0)
    {
        ForceReboot();
    }

    else if (strncmpi(TCPRXBuffer, "servo", 5) == 0)
    {
        int val1;
        int val2;
        sscanf(TCPRXBuffer, "%s %d %d", temp, &val1, &val2);
        ServoTop = val1;
        ServoBottom = val2;
    }

    else if (strncmpi(TCPRXBuffer, "motor", 5) == 0)
    {
int val1;
int val2;
sscanf(command, "%s %d %d", temp, &val1, &val2);
    if (val1 > 25) val1 = 25;
    if (val1 < -25) val1 = -25;
    if (val2 > 25) val2 = 25;
    if (val2 < -25) val2 = -25;
    MotorLeft = val1;
    MotorRight = val2;
}

/*-------------------------------------------------------------------
TCP Server Task
-------------------------------------------------------------------*/
void TCPServerTask(void * pd)
{
    //int ListenPort = (int) pd;
    int ListenPort = TCP_PORT;

    // Set up the listening TCP socket
    int fdListen = listen(INADDR_ANY, ListenPort, 5);
    if (fdListen > 0)
    {
        IPADDR client_addr;
        WORD port;

        while(1)
        {
            // The accept() function will block until a TCP client requests
            // a connection. Once a client connection is accepting, the
            // file descriptor fdnet is used to read/write to it.
            //iprintf("Waiting for connection on port %d...
            //ListenPort ");
            int fdnet = accept(fdListen, &client_addr, &port, 0);

            LED[1][0] = 0x00;
            LED[1][1] = 0x40;
            LED[1][2] = 0x00;

            //iprintf("Connected to: "); ShowIP(client_addr);
            //iprintf(":%d
", port);
            writestring(fdnet, "Welcome to ClickyBot ");
            writestring(fdnet, Robot_Name);
            writestring(fdnet, ":\r\n");
            char s[20];
            //IPtoString(EthernetIP, s);
            //siprintf(RXBuffer, "You are connected to IP Address %s,
            port %d\r\n", s, TCP_LISTEN_PORT);
            //writestring(fdnet, RXBuffer);
while (fdnet > 0)
{
    /* Loop while connection is valid. The read() function will return 0 or a negative number if the client closes the connection, so we test the return value in the loop. Note: you can also use ReadWithTimout() in place of read to enable the connection to terminate after a period of inactivity. */
    int n = 0;
    char inchar = 0;
    ServoBottom = -45;
    char temp[20];
    int cnt = 0;
    do {
        //n = read( fdnet, TCPRXBuffer, TCP_RX_BUFSIZE );
        //n = read( fdnet, TCPRXBuffer, 1 );
        cnt = read( fdnet, &inchar, 1 );
        if(inchar==-1){
            cnt = read( fdnet, &inchar, 1 );
            cnt = read( fdnet, &inchar, 1 );
            continue;
        }
        if(cnt && inchar!='\r'){
            TCPRXBuffer[n] = inchar;
            n++;
            TCPRXBuffer[n] = '\0';
            //siprintf(temp,"val = %d\r\n",inchar);
            //writestring(fdnet,temp);
        }
        else{
            if(inchar=='\r'){
                n = 0;
            }
        }
    }
    //ServoBottom = 0;
    //iprintf( "Read %d bytes: %s\n", n, RXBuffer );
    //siprintf(RXBuffer, "You are connected to IP Address %s, port %d\r\n", %s, TCP_LISTEN_PORT);
    //writestring(fdnet, TCPRXBuffer);
    // while ( cnt>0 );
    //writestring(fdnet, "Recieved:");
    //writestring(fdnet, TCPRXBuffer);
    //writestring(fdnet, "\r\n");
    //siprintf(temp,"count = %d\r\n",n);
    //writestring(fdnet,temp);
    ServoBottom = 45;
    //OSTimeDly(20);
void LinkTask(void * pd)
{
    //int servo_test = -25;
    int Init_compass = 0;
    int init_tilt = 0;
    char temp[30];

    while(1){
        OSTimeDly(20);
        if(Link_Mode==1 && Link_fd==0){
            Link_Mode = 0;
            Link_fd = connect(other_robot, 6505, 6502, 60);
            if(Link_fd<0) Link_fd = 0;
            init_compass = CompassHeading;
            init_tilt = (int)TiltRegs.wYTilt;
        }
        if(Link_Mode==-1 && Link_fd!=0){
            Link_Mode = 0;
            close(Link_fd);
            Link_fd = 0;
            LED[1][0] = 0x00;
            LED[1][1] = 0x00;
            LED[1][2] = 0x40;
        }
        if(Link_fd){
            //servo_test += 10;
            //if(servo_test>80) servo_test = -25;
            int top = ((int)TiltRegs.wYTilt - init_tilt)*90/1000;
            int bot = (init_compass - CompassHeading)/10;
            if(top>90) top = 90;
            if(top<-45) top = -45;
            if(bot>90) bot = 90;
            if(bot<-90) bot = -90;
            siprintf(temp,"servo %d %d\r",top,bot);
        }
    }
}
writestring(Link_fd,temp);
LED[1][0] = 0x00;
LED[1][1] = (BYTE)bot;
LED[1][2] = (BYTE)top;
A.1.2 cmd.h

/*
 * cmd.h
 * ECE 405 - 406 Senior Design Project
 * Team Members: Michael DeMange
 * Rodrigo Tamashiro
 * William Westrick
 * */

#ifndef CMD_H_
#define CMD_H_

int Cmd_Auth( const char *name, const char *pass);
int Cmd_Execute( const char *command, FILE *fp, void *pData);
void *Cmd_Connect( FILE *fp);
void Cmd_Prompt( FILE *fp, void *pData);
void Cmd_Disconnect( FILE *fp, int cause, void *pData);
void Cmd_Init(int prio);

extern int Robot_ID;
extern char Robot_Name[10];

extern int ServoTop;
extern int ServoBottom;

extern char MotorLeft;
extern char MotorRight;
extern BYTE MotorDist;

extern WORD RangeLeft;
extern WORD RangeRight;
extern WORD RangeMiddle;

extern int Link_Mode;

extern BYTE LED[2][3];

#endif /* CMD_H_ */
A.1.3 cmd.cpp

/*
 * cmd.cpp
 * ECE 405 - 406 Senior Design Project
 * Team Members: Michael DeMange
 * Rodrigo Tamashiro
 * William Westrick
 */

#include "predef.h"
#include <stdio.h>
#include <stdlib.h>
#include <constants.h>
#include <ctype.h>
#include <string.h>
#include <serial.h>
#include <iosys.h>
#include <utils.h>
#include <ip.h>
#include <tcp.h>
#include <bsp.h>

#include <command.h>

#include "cmd.h"

#define SERIALPORT_TO_USE (0) /* this is the usb port on the robot.
#define BAUDRATE_TO_USE (115200)
#define STOP_BITS (1)
#define DATA_BITS (8)
#define TCP_PORT_TO_USE (23) /* The Standard Telnet port

/* The User Authentication Callback */
int Cmd_Auth( const char *name, const char *pass )
{
    /* For testing reject the set if they are the same */
    if ( strcmp( name, pass ) == 0 )
    {
        return CMD_FAIL;
    }
    else
    {
        return CMD_OK;
    }
}

/* The command processing Callback */
int Cmd_Execute( const char *command, FILE *fp, void *pData )
{
    char temp[20];
    //if ( command[0] )
// Close the connection if we receive Logout
if (stricmp(command, "logout") == 0){
  return CMD_CLOSE;
}

// Reboot
else if (stricmp(command, "reboot") == 0){
  ForceReboot();
  return CMD_CLOSE;
}
else if (stricmp(command, "link") == 0){
  Link_Mode = 1;
}
else if (stricmp(command, "unlink") == 0){
  Link_Mode = -1;
}
else if (strncmpi(command, "servo", 5) == 0){
  int val1;
  int val2;
  sscanf(command, "%s %d %d", temp, &val1, &val2);
  ServoTop = val1;
  ServoBottom = val2;
}
else if (strncmpi(command, "led", 3) == 0){
  int val1;
  int val2;
  int val3;
  int val4;
  sscanf(command, "%s %d %d %d %d", temp, &val1, &val2, &val3, &val4);
  val1 %= 2;
  val2 &= 0xFF;
  val3 &= 0xFF;
  val4 &= 0xFF;
  LED[val1][0] = (BYTE)val2;
  LED[val1][1] = (BYTE)val3;
  LED[val1][2] = (BYTE)val4;
}
else if (strncmpi(command, "motor", 5) == 0){
  int val1;
  int val2;
  int val3;
  sscanf(command, "%s %d %d %d", temp, &val1, &val2, &val3);
  val1 *= 30;
  val1 /= 2;
  if (val1 > 80) val1 = 80;
  if (val1 < -80) val1 = -80;
  if (val2 > 80) val2 = 80;
  if (val2 < -80) val2 = -80;
  if (val3 <= 0) val3 = 1;
  if (val3 > 120) val3 = 120;
  MotorLeft = val1;
  MotorRight = val2;
  MotorDist = val3;
}
else if (strncmpi(command, "sen", 3) == 0) {
    if (fp) fprintf( fp, "Sensor [%d] [%d] [%d]", RangeLeft, RangeMiddle, RangeRight);
}
return CMD_OK;
}

void * Cmd_Connect( FILE *fp )
{
    //static int SessionNum;
    //const char *prompt;
    //fprintf( fp, "Hi" );

    fprintf( fp, "Welcome to the Clickybot OS Command Prompt\n\nYou are logged on to Robot #%d a.k.a \"%s\",
            Robot_ID, Robot_Name );

    fprintf( fp, \n\ntype HELP for a list of commands.\n\n" );
    //return ( void * ) prompt;
    return NULL;
}

void Cmd_Prompt( FILE *fp, void *pData )
{
    fprintf( fp, \n%d:%s>\n            Robot_ID, Robot_Name );
}

void Cmd_Disconnect( FILE *fp, int cause, void *pData )
{
    switch ( cause )
    {
        case CMD_DIS_CAUSE_TIMEOUT:
            fputs("\nTimed out\n", fp );
            break;
        case CMD_DIS_CAUSE_CLOSED:
            fputs("\nGoodBye\n", fp );
            break;
        case CMD_DIS_SOCKET_CLOSED:
            fputs("Socket closed\n", fp );
            break;
        case CMD_DIS_AUTH_FAILED:
            fputs("Authentication failed\n", fp );
            break;
    }
}

void Cmd_Init( int prio )
{
    OSTimeDly( TICKS_PER_SECOND / 2 );
    //Close the serial port incase it is already open.
SerialClose( SERIALPORT_TO_USE );

//Open the serial port
int fdserial = OpenSerial( SERIALPORT_TO_USE, BAUDRATE_TO_USE, STOP_BITS, DATA_BITS, eParityNone );

//ReplaceStdio( 0, fdserial );
//ReplaceStdio( 1, fdserial );
//ReplaceStdio( 2, fdserial );

//writestring( fdserial, "Starting\n" );

CmdAuthenticateFunc = Cmd_Auth; /* No authentication to start */
CmdCmd_func = Cmd_Execute;
CmdConnect_func = Cmd_Connect;
CmdPrompt_func = Cmd_Prompt;
CmdDisConnect_func = Cmd_Disconnect;

CmdIdleTimeout = TICKS_PER_SECOND * 5 * 60; // disconnect after 5 min
Cmdlogin_prompt = "Please Log in\n";

CmdStartCommandProcessor( prio );
CmdAddCommandFd( fdserial, TRUE, TRUE );
CmdListenOnTcpPort( TCP_PORT_TO_USE, 1, 5 );

}
A.1.4 Camera.h
/*
 * Camera.h
 *
 * Created on: Apr 12, 2012
 * ECE 405 - 406 Senior Design Project
 * Team Members: Michael DeMange
 * Rodrigo Tamashiro
 * William Westrick
 *
 */

#ifndef CAM_INIT 0x01
#define CAM_INIT 0x01
#define CAM_GET_PICTURE 0x04
#define CAM_SNAPSHOT 0x05
#define CAM_SET_PACKAGE_SIZE 0x06
#define CAM_SET_BAUD_RATE 0x07
#define CAM_RESET 0x08
#define CAM_DATA 0x0A
#define CAM_SYNC 0x0D
#define CAM_ACK 0x0E
#define CAM_NAK 0x0F
#define CAM_LIGHT 0x13

#define CAM_PACKAGE_SIZE 512
extern BYTE *JPegBuffer;
extern int JPegBufferSize;
extern BYTE *PPMBuffer1;
extern BYTE *PPMBuffer2;
extern int PPMImageWidth;
extern int PPMImageHeight;

extern char strCameraStatus[40];
extern char strCameraStatus[40];
extern int CameraSyncCount;

void CameraTask(void *pdata);

#endif /* CAMERA_H_ */
A.1.5 Camera.cpp

/*
 * Camera.cpp
 *
 * Created on: Apr 12, 2012
 * ECE 405 - 406 Senior Design Project
 * Team Members: Michael DeMange
 * Rodrigo Tamashiro
 * William Westrick
 *
 */

#include "predef.h"
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <constants.h>
#include <serial.h>
#include <iosys.h>
#include <utils.h>
#include <uCos.h>
#include "Camera.h"

char strCameraStatus[40] = "Unknown";
int CameraSyncCount = 0;

extern int CameraSyncCount;

#define MAXJPEG (32768)  // reserve 32K for jpeg buffers This should be enough space for 320x240 images from the camera but we will check for overflow.
#define MAXRAW (153600) // reserve 320x240x2 = 153600 space for raw buffers

static BYTE CompressedBuffers[2][MAXJPEG];  // buffers used for jpeg images. One for viewing and the other for loading from the camera.
//static BYTE RawBuffers[3][MAXRAW];  // buffers for the raw images. Two for viewing (LED on and LED off) and one for loading.

static int current_compressed = 0;
static int current_raw1 = 0;
static int current_raw2 = 0;

// Camera ...
int camfd = 0;
int cam_w = 0;  // image width and height
int cam_h = 0;
int cam_depth = 0;  // bit per pixel.
int cam_jw = 0;  // jpeg image width and height
int cam_jh = 0;

BYTE *JpegBuffer = NULL;
int JpegBufferSize = 0;

BYTE *PPMBuffer1 = NULL;
BYTE *PPMBuffer2 = NULL;
int PPMImageWidth;
int PPMImageHeight;

int CamGetResponse(BYTE *buf) {
    int cnt = 0;
    while(cnt<6) {
        int c = ReadWithTimeout(camfd, (char*)buf, 1, 5);
        if (c==0) break;
        cnt+=c;
        buf+=c;
    }
    return cnt;
}

BYTE* CameraSendCommand(BYTE cmd, BYTE par1=0x00, BYTE par2=0x00, BYTE par3=0x00, BYTE par4=0x00) {
    static BYTE buf[6];
    buf[0] = 0xAA; // always
    buf[1] = cmd; // the command
    buf[2] = par1;
    buf[3] = par2;
    buf[4] = par3;
    buf[5] = par4;

    writeall(camfd, (char*)buf, 6); // send command

    if (cmd!=CAM_ACK) { // if we are sending an ACK then don't get a response
        int cnt = CamGetResponse(buf);
        if (cnt!=6) buf[1] = 0xff; // no response or incomplete response
    }
    return buf; // return the response. It should be an ACK but we will let
    the caller deal with it.
}

BYTE* CameraReadPackage() {
    static BYTE buf[CAM_PACKAGE_SIZE];
    char* ptr = (char*)buf;

    int cnt = CAM_PACKAGE_SIZE;
    while(cnt!=0) {
        int c = ReadWithTimeout(camfd, ptr, 1, 5);
        if (c==0) break;
        cnt-=c;
        ptr+=c;
    }
    return buf;
}
int CameraGetJPeg(BYTE type)
{
    static BYTE buf[12];
    BYTE* result = CameraSendCommand(CAM_GET_PICTURE,type);
    int cnt = 0;
    int len = 0;

    BYTE *NewJpegBuffer = NULL;

    if(result[1] == CAM_ACK){
        cnt = CamGetResponse(buf);
        len = (int)buf[3] + (((int)buf[4])<<8) + (((int)buf[5])<<16);
        if(buf[1]==CAM_DATA && len<=MAXJPEG){
            CameraSendCommand(CAM_ACK); // send ack for data.
            current_compressed = (current_compressed + 1) % 2; // switch to the unused buffer.
            NewJpegBuffer = CompressedBuffers[current_compressed]; // get a pointer to the unused buffer.
            sprintf(strCameraStatus,"ln=%d",len); // debug message
            int n = len/(CAM_PACKAGE_SIZE-6) + 1; // number of packets to read
            BYTE *ptr = NewJpegBuffer; // ptr to the current buffer
            BYTE *packet;
            int datasize = 0;
            for(int pn=0; pn<n; pn++){
                packet = CameraReadPackage();
                datasize = (int)packet[2] + (((int)packet[3])<<8);
                memcpy(ptr,packet+4,datasize); // copy to the buffer.
                ptr += datasize;
                if(pn==n-1){
                    CameraSyncCount = (int)packet[0] + (((int)packet[1])<<8);
                }else{
                }
            }

            CameraSendCommand(CAM_ACK,0,0,packet[0]+1,packet[1]);

            if(datasize+6 == CAM_PACKAGE_SIZE){
                // special case requires camera state reset.
                CameraSendCommand(CAM_RESET,0x01);
            }

            if(NewJpegBuffer){
                OSLock(); // Just in case so other task don't see the new buffer with the old length.
                JpegBuffer = NewJpegBuffer;
                JpegBufferSize = len;
                OSUnlock(); // ok. safe again.
            }
        }else{
            strcpy(strCameraStatus,"jpeg nodata");
            return -1;
        }
    }else{
        sprintf(strCameraStatus,"err = %d",result[3]);
        //strcpy(strCameraStatus,"jpeg error");
        return -2;
    }
}
return 0; //OK
}

void CameraTask(void *pdata)
{
    static BYTE sync[6] = { 0xAA, 0x0D, 0x00, 0x00, 0x00, 0x00};
    static BYTE ack[6] = { 0xAA, 0x0E, 0x0D, 0xAB, 0x00, 0x00};
    static BYTE buf[12];
    int cnt;
    BYTE xx = 0x00;

    while(1){
        SerialClose(2);
        camfd = OpenSerial(2,115200,1,8,eParityNone); // 115200 is the fastest baud rate for the netburner and the fastest auto sync rate for the uCam
        CameraSendCommand(CAM_RESET,0x00,0x00,0x00,0xFF);
        OSTimeDly(TICKS_PER_SECOND*2); // wait 2 seconds for the camera to stabilize.

        //siprintf(strCameraStatus,"xxx = %d",camfd);

        // Initial sync...
        for(int i=0; i<60; i++){
            writeall(camfd,(char*)sync,6);
            CameraSyncCount++;
            cnt = CamGetResponse(buf);
            if(cnt==6){
                strcpy(strCameraStatus,"sync ok");
                if(buf[0] == 0xAA && buf[1]==0x0E){
                    sprintf(strCameraStatus,"ack = %d",buf[3]);
                    xx = buf[3];
                    break;
                }
            }
        }
        //while(1){
        //    OSTimeDly(TICKS_PER_SECOND*2);
        //}

        // we should get a sync from the camera...
        cnt = CamGetResponse(buf);
        if(cnt==6){
            strcpy(strCameraStatus,"sync ok");
            CameraSyncCount = (int)buf[3];
            // send ack
            ack[3] = xx;
            writeall(camfd,(char*)ack,6);
        }else{
            strcpy(strCameraStatus,"no sync");
        }
    }
}
OSTimeDly(TICKS_PER_SECOND*2); // wait 2 seconds for the camera so stabilize.

cam_w = 80; // image width and height
cam_h = 60;
cam_depth = 8; // bit per pixel.
cam_jw = 80; // jpeg image width and height
cam_jh = 64;

BYTE* result = CameraSendCommand(CAM_INIT,0x07,0x05,0x05);

if(result[1] == CAM_ACK){
    result = CameraSendCommand(CAM_SET_PACKAGE_SIZE,8,(CAM_PACKAGE_SIZE)&0xFF,(CAM_PACKAGE_SIZE)>>8);
}

int cam_status = 0;

while(cam_status==0){
    cam_status = CameraGetJpeg(0x05);
    OSTimeDly(5);
}
}
B.1 PSoC CODES

B.1.1 LEDs, Speaker, and Tilt Sensor

// TiltSensor/main.c

// This PSoC is responsible for reading the Tilt Sensor, playing notes on the
// speaker and driving the two RGB LEDs
//
// ECE 405 - 406 Senior Design Project
// Team Members: Michael DeMange
// Rodrigo Tamashiro
// William Westrick

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

enum Commands {
  cmdIdle = 0,
  cmdPlayNote = 'P'
};
enum Notes {
  noteRest = 0,
  noteC = 1,
  noteCS = 2,
  noteD = 3,
  noteDS = 4,
  noteE = 5,
  noteF = 6,
  noteFS = 7,
  noteG = 8,
  noteGS = 9,
  noteA = 10,
  noteAS = 11,
  noteB = 12
};

// Octive zero note periods ... freq =
// SysClk/VC1/VC2/VC3/(notePeriod[n]>>octive)
// Example for A4 = 24MHz/6/4/25/(notePeriod[10]>>4) = 40kHz/(1455>4) =
// 40kHz/90 = 444 Hz (i.e. slightly out of tune. should be 440 Hz :)
WORD notePeriod[] = {0, 2446, 2309, 2180, 2057, 1942, 1832, 1730, 1633, 1541,
  1455, 1373, 1296};

WORD noteNextFreq = 0;
WORD noteNextDuration = 0;
WORD noteCurrentDuration = 0;
BYTE testSeq[] = {
};
BYTE testNote = 0;
BYTE testOctive = 3;
BYTE testTotalNotes = 3*7+1;

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

enum Commands {
  cmdIdle = 0,
  cmdPlayNote = 'P'
};
enum Notes {
  noteRest = 0,
  noteC = 1,
  noteCS = 2,
  noteD = 3,
  noteDS = 4,
  noteE = 5,
  noteF = 6,
  noteFS = 7,
  noteG = 8,
  noteGS = 9,
  noteA = 10,
  noteAS = 11,
  noteB = 12
};

// Octive zero note periods ... freq =
// SysClk/VC1/VC2/VC3/(notePeriod[n]>>octive)
// Example for A4 = 24MHz/6/4/25/(notePeriod[10]>>4) = 40kHz/(1455>4) =
// 40kHz/90 = 444 Hz (i.e. slightly out of tune. should be 440 Hz :)
WORD notePeriod[] = {0, 2446, 2309, 2180, 2057, 1942, 1832, 1730, 1633, 1541,
  1455, 1373, 1296};

WORD noteNextFreq = 0;
WORD noteNextDuration = 0;
WORD noteCurrentDuration = 0;
BYTE testSeq[] = {
};
BYTE testNote = 0;
BYTE testOctive = 3;
BYTE testTotalNotes = 3*7+1;

#pragma interrupt_handler GPIO_ISR
void GPIO_ISR(void);

#pragma interrupt_handler XTilt_ISR
void XTiltComplete(void);
void YTiltComplete(void);

void TimerTick(void);

struct I2C_Regs {
  // Example I2C interface structure
  BYTE bRamRegLength;
  BYTE bRomLength;
  BYTE par[4];
BYTE bCmd;
BYTE bCmdID;
BYTE bCompletedCmdID;
BYTE bLed1_R;
BYTE bLed1_G;
BYTE bLed1_B;
BYTE bLed2_R;
BYTE bLed2_G;
BYTE bLed2_B;
WORD wXTilt;
WORD wYTilt;
BYTE bTiltCheck;
BYTE bSpeakerStat;
BYTE bxxx;
}

const BYTE Version[] = "v1.0 Tilt-Heading";

// X Tilt Vars...
// volatile BYTE XTilt_Overflow = 1; // set to 1 if there is an overflow.
// initially set so that the first reading will be ignored
// volatile WORD XTilt_Capture = 0; // value is set in ASM interrupt to
// the timer value. (we do this in ASM to reduce latency.

void main(void)
{
    Regs.bRamRegLength = sizeof(Regs);
    Regs.bRomLength = sizeof(Version);
    Regs.wXTilt = 0;
    Regs.wYTilt = 0;

    Regs.bLed1_R = Regs.bLed1_G = Regs.bLed1_B = 0; // all off
    Regs.bLed2_R = Regs.bLed2_G = Regs.bLed2_B = 0; // all off
    Regs.bCmd = cmdIdle;

    EzI2Cs_SetRamBuffer(sizeof(Regs), sizeof(Regs), (BYTE *) &Regs); // Set
    EzI2Cs_SetRomBuffer(sizeof(Version), Version); // Set up RAM buffer
    M8C_EnableGInt; // Turn on interrupts

    DigBuf_X_EnableInt();
    DigBuf_X_Start();

    DigBuf_Y_EnableInt();
    DigBuf_Y_Start();

    LED1_R_EnableInt();
    LED1_R_Start();
    LED1_G_Start();
    LED1_B_Start();
    LED2_R_Start();
    LED2_G_Start();
    LED2_B_Start();

    //Regs.bLed1_G = Regs.bLed1_B = 0x20; // all off
//Regs.bLed2_R = 0x20; // all off

//M8C_EnableIntMask(INT_MSK0, INT_MSK0_GPIO);

EzI2Cs_Start();                             // Turn on I2C

Counter16_X_Start();
//TachTimer16_X_EnableCaptureInt();
//TachTimer16_X_Start();

Counter16_Y_Start();

//PWM16_Speaker_Start();

while(1){
    LED1_R_WritePulseWidth(0xFF - Regs.bLed1_R);
    LED1_G_WritePulseWidth(0xFF - Regs.bLed1_G);
    LED1_B_WritePulseWidth(0xFF - Regs.bLed1_B);
    LED2_R_WritePulseWidth(0xFF - Regs.bLed2_R);
    LED2_G_WritePulseWidth(0xFF - Regs.bLed2_G);
    LED2_B_WritePulseWidth(0xFF - Regs.bLed2_B);

    /*if(Regs.wXTilt>5000){
        Regs.bLed1_R = 0;
        Regs.bLed2_R = (Regs.wXTilt-5000)/5;
    }else{
        Regs.bLed1_R = (5000-Regs.wXTilt)/5;
        Regs.bLed2_R = 0;
    }
    if(Regs.wYTilt>5000){
        Regs.bLed1_B = 0;
        Regs.bLed2_B = (Regs.wYTilt-5000)/5;
    }else{
        Regs.bLed1_B = (5000-Regs.wYTilt)/5;
        Regs.bLed2_B = 0;
    }*/

    //Music code...
    if(noteCurrentDurration==0){
        PWM16_Speaker_Stop();
        if(noteNextDurration){
            PWM16_Speaker_WritePeriod(noteNextFreq); // set the
            PWM16_Speaker_WritePulseWidth(noteNextFreq>>1); // pulse width is half the frequency
            noteCurrentDurration = noteNextDurration; // remember the durrartion of this note.
            noteNextDurration = 0; // ready for next note.
            Regs.bSpeakerStat = 0; // flag that note buffer is empty.
            if(noteNextFreq) PWM16_Speaker_Start(); // olny start
        }
    }

    if(Regs.bSpeakerStat == 0 && Regs.bCmd==cmdIdle){
if(testtotalnotes){
    Regs.par[0] = testSeq[testnote];
    Regs.par[1] = testoctive;
    Regs.par[2] = 1; // number of 32nd notes 32/32 = whole note.
    Regs.bCmd = cmdPlayNote;

    testnote++;
    if(testnote>=7){
        testnote = 0;
        testoctive++;
    }
    testtotalnotes--;
    if(testtotalnotes){
        Regs.bLed1_G = 176 - (testtotalnotes<<3);
        Regs.bLed2_B = 176 - (testtotalnotes<<3);
    }else{
        Regs.bLed1_G = 0;
        Regs.bLed2_B = 0;
    }
}

if(Regs.bCmd==cmdPlayNote){
    Regs.bSpeakerStat = 1; // flag that a note is waiting to be played.
    noteNextDuration = ((WORD)Regs.par[2])<<2; // duration is 4* the par value.
    if(Regs.par[0]>0 && Regs.par[0]<=12){ // 1 to 12 are valid notes. all other values are a "rest"
        noteNextFreq = notePeriod[Regs.par[0]]>>Regs.par[1];
        //note period shifted by octive value
    }else{
        noteNextFreq = 0;
    }
    Regs.bCmd = cmdIdle; // set command to idle indicating the command has been processed.
}
}

void GPIO_ISR(void)
{
    //Regs.bStat++; // = 1;
    return;
}

void XTiltComplete(void)
{
    Counter16_X_Stop(); // Stop the counter. (The counter should be currently disabled since the counter enable line comes directly from the tilt sensor output.)
    Regs.wXTilt = Counter16_X_PERIOD - Counter16_X_wReadCounter();
    Regs.bTiltCheck = ((BYTE *)(&Regs.wXTilt))[0] + ((BYTE *)(&Regs.wXTilt))[1] + ((BYTE *)(&Regs.wYTilt))[0] + ((BYTE *)(&Regs.wYTilt))[1];
Counter16_X_WritePeriod(Counter16_X_PERIOD); // reset the counter
Counter16_X_Start();  // activate and wait for next pulse.

    return;

}

void YTiltComplete(void)
{
    Counter16_Y_Stop(); // Stop the counter. (The counter should be
currently disabled since the counter enable line comes directly from the tilt
sensor output.)
    Regs.wYTilt = Counter16_Y_PERIOD - Counter16_Y_wReadCounter();
    Regs.bTiltCheck = ((BYTE *)(&Regs.wXTilt))[0] + ((BYTE *
    *)(&Regs.wXTilt))[1] + ((BYTE *)(&Regs.wYTilt))[0] + ((BYTE *
    *)(&Regs.wYTilt))[1];
    Counter16_Y_WritePeriod(Counter16_Y_PERIOD); // reset the counter
    Counter16_Y_Start();  // activate and wait for next pulse.

    return;

}

void TimerTick(void)
{
    if(noteCurrentDuration) noteCurrentDuration--;
}
### B.1.2 Motors and Voltage/Current Sensor

```c
#include <m8c.h>       // part specific constants and macros
#include "PSoCAPI.h"   // PSoC API definitions for all User Modules

#define FORWARD_LEFT 0x80
#define REVERSE_LEFT 0x20
#define FORWARD_RIGHT 0x04
#define REVERSE_RIGHT 0x01
#define MASKRIGHT 0xFA
#define MASKLEFT 0x5F
#define START 0x40
#define START1 0xC4

char start;
//char dir;
char val;
signed char left;
signed char right;

volatile BYTE LeftTickLimit = 0;
volatile BYTE RightTickLimit = 0;
char * strPtr;
char * strPtrDebug;

void power_MotorLeft(signed char l);
void power_MotorRight(signed char r);
signed char convert_Data(char i, char a, char b, char c);int convertToPwm(signed char p);
void voltage_Level(int result);
void REncoderComplete(void);
void LEncoderComplete(void);

volatile struct I2C_regs
{
    BYTE bInfo;
    BYTE bPar[4];
    BYTE bCmd;
    WORD wREncoder;
    WORD wLEncoder;
    BYTE bEncoderCheck;
```
void main(void)
{
    reg.bInfo = sizeof(reg);
    reg.bCmd = 0;
    start = START;
    PRT0DR = PRT0DR | start;

    reg.bLTick = 123;

    // Start the motors
    MotorRB_Start();
    MotorLB_Start();

    // I2C Communication
    EzI2Cs_1_SetRamBuffer(sizeof(reg), sizeof(reg), (BYTE*)&reg);
    EzI2Cs_1_Start();                            // Turn on I2C

    // Amplifier to measure the current
    Amplifier_Start(Amplifier_MEDPOWER);

    // Analog to digital converter counter
    Voltage_Current_Start(Voltage_Current_HIGHPOWER);     // Turn on Analog

    // Enable counters for the encoder
    Encoder_Right_EnableInt();                 /* enable the interrupt    */
    Encoder_Right_Start();                     /* start the counter        */
    Encoder_Left_EnableInt();                 /* enable the interrupt    */
    Encoder_Left_Start();                     /* start the counter        */

    // Enable buffers for counters
    Interrupt_Right_EnableInt();
    Interrupt_Right_Start();
    Interrupt_Left_EnableInt();
    Interrupt_Left_Start();
/********************************************
M8C_EnableGInt;                     /* enable global interrupts     */

while(1)
{
    int iResult1, iResult2;
    int debug;
    signed char 1;
    int pwmL;

    // left = -50;
    // right = -50;
    //
    // reg.bPar[0] = left;
    // reg.bPar[1] = right;
    //
    power_Motor((signed char) reg.bPar[0], (signed char) reg.bPar[1]);

    /*debug = reg.bRTick;
    if (debug &1)
    {
        left = 00;
        right = 00;
        reg.bPar[0] = left;
        reg.bPar[1] = right;
        power_Motor((signed char) reg.bPar[0], (signed char) reg.bPar[1]);
    }
    else
    {
        left = 30;
        right = 30;
        reg.bPar[0] = left;
        reg.bPar[1] = right;
        power_Motor((signed char) reg.bPar[0], (signed char) reg.bPar[1]);
    }*/

    if(reg.bCmd == 'M')
    {
        LeftTickLimit = reg.bPar[2];
        RightTickLimit = reg.bPar[3];
        power_MotorLeft((signed char) reg.bPar[0]);
        power_MotorRight((signed char) reg.bPar[1]);
        reg.bCmd = 0;
    }

    if(Voltage_Current_fIsDataAvailable()) // Wait for data to be ready
    {
        reg.wVoltage = Voltage_Current_iGetData1(); // Get Data from ADC Input1
    }
}*/
reg.wCurrent = Voltage_Current_iGetData2ClearFlag(); // Get Data from ADC Input2  

voltage_Level(iResult2); // User function to use data
}

signed char convert_Data(char i, char a, char b, char c)
{
    char sign;
    char cent;
    char dec;
    char unit;
    signed char temporary;

    sign = i;
    cent = (a & 0x0F) * 100;
    dec = (b & 0x0F) * 10;
    unit = c & 0X0F;
    temporary = dec + unit;
    if (sign == '-') {
        temporary = temporary * (-1);
    }
    return temporary;
}

void power_MotorLeft(signed char l)
{
    int pwmL;
    char dir;

    dir = PRT0DR & MASKLEFT;
    if (l < 0 ) {
        dir = dir | REVERSE_LEFT;
    } else {
        dir = dir | FORWARD_LEFT;
    }
    PRT0DR = dir;
    pwmL = convertToPwm(l);
    MotorLB_WritePulseWidth(pwmL);
}
void power_MotorRight(signed char r)
{
    int pwmR;
    char dir;

    dir = PRT0DR & MASKRIGHT;

    if(r < 0)
    {
        dir = dir | REVERSE_RIGHT;
    }
    else
    {
        dir = dir | FORWARD_RIGHT;
    }

    PRT0DR = dir;
    pwmR = convertToPwm(r);

    MotorRB_WritePulseWidth(pwmR);
}

int convertToPwm(signed char p)
{
    char temp;
    int result;

    temp = p;

    if(p < 0)
    {
        temp = temp* (-1);
    }

    result = temp*2;

    return result;
}

void voltage_Level(int result)
{
    int range;

    range = result;
}

void REncoderComplete(void)
{
    Encoder_Right_Stop(); // Stop the counter.
    reg.wREncoder = Encoder_Right_PERIOD - Encoder_Right_wReadCounter();
    reg.bEncoderCheck = ((BYTE *)&reg.wREncoder)[0] + ((BYTE *)&reg.wLEncoder)[0] + ((BYTE *)&reg.wREncoder)[1] + ((BYTE *)&reg.wLEncoder)[1];
    Encoder_Right_WritePeriod(Encoder_Left_PERIOD); // reset the counter
    Encoder_Right_Start(); // activate and wait for next pulse.
reg.bRTick++;
if(RightTickLimit != 0)
{
    RightTickLimit--;
    if(RightTickLimit == 0)
    {
        power_MotorRight(0);
        power_MotorLeft(0);
    }
}

return;

void LEncoderComplete(void)
{
    Encoder_Left_Stop(); // Stop the counter.
    reg.wLEncoder = Encoder_Left_PERIOD - Encoder_Left_wReadCounter();
    reg.bEncoderCheck = ((BYTE *)&reg.wREncoder)[0] + ((BYTE *)&reg.wLEncoder)[0] + ((BYTE *)&reg.wLEncoder)[1];
    Encoder_Left_WritePeriod(Encoder_Left_PERIOD); // reset the counter
    Encoder_Left_Start(); // activate and wait for next pulse.
    reg.bLTick++;
    if(LeftTickLimit != 0)
    {
        LeftTickLimit--;
        if(LeftTickLimit == 0)
        {
            power_MotorLeft(0);
            power_MotorRight(0);
        }
    }
    //return;
}
B.1.3 Ultrasonic Sensors and Servo Motors

// Servo/main.c

// This PSoC is responsible for reading the the 3 ultrasonic sensors and driving the 2 servo motors.

// ECE 405 - 406 Senior Design Project
// Team Members: Michael DeMange
// Rodrigo Tamashiro
// William Westrick

#include <m8c.h> // part specific constants and macros
#include "PSoCAPI.h" // PSoC API definitions for all User Modules

char val1 = 0x57;
char val2;
char sens;
char sign;
char dec;
char unit;
char * strPtr;
char * strPtrDebug;

signed char temporary;
signed char degree;

int top=0;
int bottom=0;
int servotop;
int servobottom;
int range;
int temp;
int cent;

int convert_degree(int degree);
signed char convert_Data(char i, char b, char c);
int convert_Range(char z, char y, char t);
void servo(char S, signed char d);
void sensor(char a);
int debug(int range);

volatile struct I2C_regs
{
    BYTE bInfo;
    BYTE bPar[4];
    BYTE bCmd;
void main(void) {
    signed char test;
    CHAR old_top = 0;  // used to detect a change in the value
    CHAR old_bottom = 0;

    reg.bInfo = sizeof(reg);
    reg.bCmd = 0;

    reg.bTopServo = 0;
    reg.bBottomServo = 0;

    reg.wSensorRight = 1234;

    PRT2DR = PRT2DR & val1;

    /* set period to eight clocks */
    PWM_Servo_Bottom_WritePeriod(1000);
    PWM_Servo_Top_WritePeriod(1000);

    /* start the PWM! */
    PWM_Servo_Bottom_Start();
    PWM_Servo_Top_Start();

    /**********************************************************************
    * Inverter to test the sensor *
    DigInv_1_EnableInt();  // Use if interrupts desired
    DigInv_1_Start();      // Enable Inverter
    /**********************************************************************

    // Start I2C communication
    EzI2Cs_1_SetRamBuffer(sizeof(reg), sizeof(reg), (BYTE *) &reg);
    EzI2Cs_1_Start();  // Turn on I2C

    // Start the sensor Right
    Sensor_Right_CmdReset();
    Sensor_Right_EnableInt();
    Sensor_Right_Start(Sensor_Right_PARITY_NONE);

    // Start the sensor Left
    Sensor_Left_CmdReset();
    Sensor_Left_EnableInt();
    Sensor_Left_Start(Sensor_Left_PARITY_NONE);

    // Start the sensor Middle
Sensor_Middle_CmdReset();
Sensor_Middle_EnableInt();
Sensor_Middle_Start(Sensor_Middle_PARITY_NONE);

/**************************************************************************
/* Debugger block */
Debugger_CmdReset();
Debugger_IntCntl(Debugger_ENABLE_RX_INT); // Enable RX interrupts
Debugger_Start(Debugger_PARITY_NONE);
/**************************************************************************

M8C_EnableGInt;                     /* enable global interrupts     */
Baud_Rate_Start();                 /* start the counter!           */

// Initial position from the servos top and bottom are 0
servotop = convert_degree(top);
servobottom = convert_degree(bottom);
PWM_Servo_Bottom_WritePulseWidth(servobottom);
PWM_Servo_Top_WritePulseWidth(servotop);

while(1)
{

/**************************************************************************
/* Debug block *
if(Debugger_bCmdCheck())
{
   // Wait for command
   if(strPtrDebug = Debugger_szGetParam())
   { // More than delimiter"
      Debugger_CPutString("Found valid command\r\n");

      reg.bCmd = strPtrDebug[0];

      //Call the servos to work
      if(reg.bCmd == 'S')
      {
         degree = convert_Data(strPtrDebug[2], strPtrDebug[3],
                                strPtrDebug[4]);
         reg.bPar[0] = strPtrDebug[1];
         reg.bPar[1] = degree;
         reg.bCmd = 0;
         servo(reg.bPar[0], (signed char) reg.bPar[1]);
      }
      if(reg.bCmd == 'R')
      {
         reg.bPar[0] = strPtrDebug[1];
         //reg.bCmd=0;
         sensor(reg.bPar[0]);
      }
   }
}
PRT2DR = PRT2DR & val1;
Debugger_CmdReset(); // Reset command
}  
/******************** Keep checking the value of the sensors 
*************************/

if(reg.bCmd == 'S')
{
    //servo(reg.bPar[0], (signed char) reg.bPar[1]);
    if(reg.bPar[0] == 't'){
        reg.bTopServo = (CHAR) reg.bPar[1];
    }else if(reg.bPar[0] == 'b') {
        reg.bBottomServo = (CHAR) reg.bPar[1];
    }
    reg.bCmd = 0;
}

if(reg.bCmd == 'R')
{
    sensor(reg.bPar[0]);
    reg.bCmd=0; // signal that command was read.
    //reg.wSensorRight++;
}

// Detect a change in the desired servo location.
if(reg.bTopServo != old_top){
    servo('t', reg.bTopServo);
    old_top = reg.bTopServo;
}

if(reg.bBottomServo != old_bottom){
    servo('b', reg.bBottomServo);
    old_bottom = reg.bBottomServo;
}

PRT2DR = PRT2DR & val1;

if(Sensor_Right_bCmdCheck())
{
    // Wait for command
    if(strPtr = Sensor_Right_szGetParam())
    {
        Debugger_PutString(strPtr); // Print out
        command
    }
    range = convert_Range(strPtr[1], strPtr[2], strPtr[3]);
    reg.wSensorRight = range;
    Sensor_Right_CmdReset();
    Debugger_PutSHexInt(range);
}
if(Sensor_Middle_bCmdCheck())
{
    // Wait for command
    if(strPtr = Sensor_Middle_szGetParam())
    {
        Debugger_PutString(strPtr);
    }
    range = convert_Range(strPtr[1], strPtr[2], strPtr[3]);
    reg.wSensorMiddle = range;
    Sensor_Middle_CmdReset();
    Debugger_PutSHexInt(range);
}

if(Sensor_Left_bCmdCheck())
{
    // Wait for command
    if(strPtr = Sensor_Left_szGetParam())
    {
        Debugger_PutString(strPtr);
    }
    range = convert_Range(strPtr[1], strPtr[2], strPtr[3]);
    reg.wSensorLeft = range;
    Sensor_Left_CmdReset();
    Debugger_PutSHexInt(range);
}

/**********************************************************
**************************
/**********************************************************

int convert_degree(int degree)
{
    //Convert the degrees into PWM
    int result;
    result = 125 + (0.922*degree);
    return result;
}

signed char convert_Data(char i, char b, char c)
{
    sign = i;
    dec = (b & 0x0F) * 10;
    unit = c & 0X0F;
    temporary = dec+unit;
    if (sign == '-')
    {
        temporary = temporary * (-1);
    }
```c
return temporary;
}

int convert_Range(char z, char y, char t)
{
    cent = (z & 0X0F) * 100;
    dec = (y & 0x0F) * 10;
    unit = t & 0X0F;
    temp = cent+dec+unit;
    return temp;
}

void servo(char S, signed char d)
{
    sens = S;
    if (sens == 't')
    {
        top = d;
    }
    if(sens == 'b')
    {
        bottom = d;
    }
    servotop = convert_degree(top);
    servobottom = convert_degree(bottom);
    PWM_Servo_Bottom_WritePulseWidth(servobottom);
    PWM_Servo_Top_WritePulseWidth(servotop);
}

void sensor(char a)
{
    val2 = a;
    switch (val2)
    {
        case 'r':
            PRT2DR = PRT2DR | 0x80;
            reg.wSensorRight = 0; // this will serve as a flag that the
            reading is currently being taken
            break;

        case 'm':
            PRT2DR = PRT2DR | 0x20;
            reg.wSensorMiddle = 0; // this will serve as a flag that the
            reading is currently being taken
            break;

        case 'l':
            PRT2DR = PRT2DR | 0x08;
```
reg.wSensorLeft = 0;  // this will serve as a flag that the
reading is currently being taken
break;
}
}

int debug(int range)
{
    int a;
    a=range/2;
    return a;
}