Automated Vehicle Guidance System for Parking Structures

A graduate project submitted in partial fulfillment of the requirements
For the degree of Masters of Science
in Electrical Engineering

By

Aditya Gandhi

August 2012
The graduate project of Aditya Gandhi is approved:

__________________________  ________________________
Ramin Roosta, Ph.D.  Date

__________________________  ________________________
Somnath Chattopadhyay, Ph.D.  Date

__________________________  ________________________
Ronald Mehler, Ph.D., Chair  Date

California State University, Northridge
ii
Acknowledgement

Apart from the efforts of me, the success of this project depends largely on the encouragement and guidelines of many others. I take this opportunity to express my gratitude to the people who have been instrumental in the successful completion of this project.

I would like to show my greatest appreciation to Dr. Ronald Mehler. I am extremely grateful for his tremendous support and help. Without his encouragement and guidance this project would not have materialized. I am also highly thankful to other faculty members in the committee Dr. Somnath Chattopadhyay and Dr. Ramin Roosta for their fruitful suggestions both on this project and in my general academic pursuits.
Table of Contents

Signature Page......................................................................................................................ii
Acknowledgement ............................................................................................................iii
List of Figures .....................................................................................................................v
Abstract...............................................................................................................................vii

CHAPTER 1: Understanding need for WSN based Parking Guidance Systems.............1
CHAPTER 2: Project Design Overview.................................................................3
CHAPTER 3: Sensor module and Motor..............................................................11
CHAPTER 4: Wireless Network.................................................................16
CHAPTER 5: Data Structures Used.................................................................28
CHAPTER 6: Power Optimization and battery performance...................................31
CHAPTER 7: Failsafe used in the hardware & software......................................41
CHAPTER 8: Software Architecture (Device and Server)..................................44
CHAPTER 9: Hardware Schematic.................................................................56

REFERENCES.......................................................................................................................58

APPENDIX..............................................................................................................................59
List of Figures

Figure 2.1: Components of hardware.................................................................3
Figure 2.2: System closed in an enclosure box .................................................4
Figure 2.3: AVR ATMEGA328p IO Diagram ...............................................5
Figure 2.4: XBee module .................................................................................6
Figure 2.5: Arduino FIO .................................................................................7
Figure 3.1: PING)) Sensor from parallax..........................................................12
Figure 3.2: Ultrasonic ranger from Devantech Ltd (Robot Electronics).............12
Figure 3.3: URM V3.2 Ultrasonic sensor from LetsMakeRobots.com..............12
Figure 3.4: Maxbotix Ultrasonic Sensors .....................................................13
Figure 4.1: ZigBee network architecture......................................................17
Figure 4.2: Digi Mesh network architecture.................................................18
Figure 4.3: An illustration of how Digi Mesh handles routing failures...........20
Figure 4.4: API Frame Specification for ZigBee............................................22
Figure 4.5: API Frame Specification for DigiMesh........................................22
Figure 4.6: API Frame Data for ZigBee .........................................................23
Figure 4.7: API Frame Data for DigiMesh.....................................................24
Figure 4.8: ZigBee Tx Request..........................................................25
Figure 4.9: DigiMesh Tx Request...............................................................26
Figure 5.1: Snapshot of the database from MySQL admin page.................30
Figure 6.1: 3.7V 1000mAh battery...............................................................33
Figure 6.2: Battery life estimated for CPU .........................................................34
Figure 6.3: Battery life estimated for the sensor .................................................35
Figure 6.4: Battery life calculated for ZigBee using the tool from DigiMesh.........36
Figure 6.5: 9V 1200mAh Battery ..............................................................38
Figure 6.6: Battery life calculation for stepper motor ........................................39
Figure 6.7: 7.2 V 2500 Battery ..............................................................39
Figure 6.8: 6V 2000 mAh .................................................................40
Figure 8.1: Main () flow chart part 1 ..........................................................47
Figure 8.2: Main () flow chart part 2 ..........................................................48
Figure 8.3: Main () flow chart part 3 ..........................................................49
Figure 9.1: Arduino Fio Schematic ............................................................56
Figure 9.2: Stepper Driver Schematic ..........................................................57
Figure 9.3: Ultrasonic Sensor MB1360 .......................................................58
Abstract

Automated Vehicle Guidance System for Parking Structures

By

Aditya Gandhi

Master of Science in Electrical Engineering

Parking is an ever-growing challenge in metropolitans around the world as the number of cars is continually increasing. Parking availability, in turn, is one of the most significant challenges to the city. Looking for empty parking spaces in congested parking spaces can be painstaking and time consuming. The average time spent in parking lots cruising for vacant spaces approximately varies from 3.5 -12 minutes [1]. These cruising cars also add to the traffic congestion, pollution and cost significant amount of time to the drivers. Another problem faced is distance of the parking lot is significant from destination, which leads to further time delays.

The lack of parking guidance results in increased re-circulation and delays while Drivers search for a parking space. Solution to such a problem is to develop an automated parking management system employing Wireless Sensor Network (WSN) technology, reducing installation costs and maintenance costs. The parking management system can detect the presence and/or absence of a vehicle in the respective parking spaces and automatically provide the location of the identified available spaces to prospective users in real-time. Also connecting various parking lots by the means of network can benefit drivers to find the nearest to their destination.
CHAPTER 1: Understanding need for WSN based Parking Guidance Systems

Many parking guidance systems exist today, with various configurations and various special purposes. None cover all the aspects of being cost-effective and easy maintenance. Most of them usually are very expensive for most parking to implement and maintain. And most of all the parking systems need extensive installation effort, which includes wiring and many hours of labor work. After which some systems / part of the system can fail due to faulty wiring or damaged cables. A single node failure in some systems can lead to a branch of nodes disconnected from the system. Also most systems that exist have only one car per node detection, which drives the cost of the system upwards. These problems can be easily addressed by WSN (Wireless Sensor Network), with sensors that can detect multiple spots from a single node. The node is designed for easy installation and setup. It has an ultrasonic sensor and a stepper motor, which directs the sensor. Why an ultrasonic sensor and a stepper motor is covered in later chapters.

1.1 Installation and setup

Various parking lots can have various needs based on the geographic layout of the lot, some maybe spread across multiple floors and some with only few floors and vast spaces on each floor, and many other layouts. Parking Guidance Systems (PGS), should be adaptable to various situations without increasing the installation / maintenance costs. Also the WSN have a property to self heal in case of a node failure, which does not affect the system as it would in a wired system. Multiple channels and auto channel detection takes care of various interferences that might occur due to other RF / Electrical sources.

1.1.1 Installation

The System that is designed is targeted at easy Installation and maintenance; the system requirement is to be mountable using least amount of resources. The PGS node is mountable into the ceiling above the cars with a few screws and no wiring is involved. A simple setup follows the installation.
1.1.2 Setup

The setup procedure is simple and quick, which saves time and money. Setup after installation can be done with a handheld device / tablet / laptop with a USB host port, which is connected via a serial port. The node when it first starts it is in calibration mode as no data is present of the node itself. The user on the device can select the node and start the setup. First location information is put into the device, the location information is nothing but each parking space has a number assigned to it. During the setup the locations for all the spaces which the node is detecting is entered in the system and thus onto the device, after which the device enters calibration mode. In the calibration mode the device directs the sensor towards the parking spot and takes reading for the spot empty value. Thus two values are saved the sensor reading for the spot when it is empty and the other for the angle at which the spot is located with respect to the node zero orientation. Once the setup is done, the node transmits data to the servers, which detects the new node and enters it into the database. These values are entered into the database and the local node in the same process, thus saving time of different setups on the server.

1.2 Maintenance and repairs

One of the major concerns is battery life and node failure. During these events a simple battery replacement or node replacement is feasible in a WSN network without affect the system. Battery replacement is a trivial task, which requires very less effort. Node replacement is also the same as installing a new node; just the overhead of setup is required. Thus this brings down maintenance cost by a significant amount.
CHAPTER 2: Project Design Overview

2.1 Hardware Overview

The overall system requirement was for individual nodes to detect multiple parking spot status, and transmit the data back to the server. Where the server after accumulating the information, uses it to guide vehicles entering the parking lot. There were a lot of problems in this idea, which needed solutions to be feasible. The cost of the sensors was the first problem, second being the battery life and the third being the wireless network architecture. All of these problems were overcome in simple and cost effective ways.

Fig 2.1 Components of hardware

Arduino fio, Stepper, Battery 3.7 1000mAh and 9V duracel battery are shown in Fig 1.1
Fig 2.2 System closed in an enclosure box.

Over all the system has a server and wireless sensor nodes. The server has wireless networking capability using XBee modules to communicate with individual sensor network. All devices communicate directly with the end device in the system, as if the network was transparent, the network stack used takes care of the routing issues.

Each node has three basic components.

1. Processor
2. Sensor
3. Wireless Network Module

Here I have covered the processor selection and the Wireless module selection. Choosing the right kind of sensors for robust vehicle detection in the parking lot is extremely crucial to this application. The sensor selection is covered in the next chapter.
2.1.1 Processor Selection

For each node I have chosen an 8-bit controller from Atmel, AVR ATMEGA328p.

The processor selection was based on parameters such as:

- **Power consumption**

  The power consumption during normal operations as per the data sheet during normal operations is 5.1 mA at 5V and max is 9mA at 5V, during power-down mode the typical current is .1 μA and max is 2 μA. This is very competitive or lower than other processors in the same category, such as PIC18F542 and Atmel SAM7X256.

- **Ram Size**

  The ram size is of concern as data packets and sensor data is going to be saved in the ram during normal operations. Thus a sufficient ram size was required for the data to be processed without overhead to processor to compromise for less memory space. ATMEGA328p has 2k or SRAM available.

- **Sleep modes**

  The processor should have the capability to go to various levels of sleep as to be woken up by interrupts such as UART or a timer cased interrupt. It helps in making the system power efficient and thus extended battery life.

- **IO capabilities**
The IO capabilities of the system are critical, as the nodes need to have a reliable and a fast way of reading from sensors and communicate over UART to the wireless XBee module. The ATMEGA328p has a reliable ADC and a Standard UART. ADC is used for battery voltage detection as well as Sensor readings.

- Cost
  Cost of AVR ATMEGA328p is low, about 2.5 $ for 1000 or above quantities, thus driving the system cost lower.[3]

- Availability for tools
  ATMEGA328p is a very popular controller used in various development platforms such as Arduino, Netduino, AVR development platform. It also has a large variety of free compiler available from ATMEL as well as many other communities. The libraries available for them are also vast. Which brings down the development time.

2.2.2 Wireless Module Selection

The basic criterion for the wireless module was to have mesh networking capability. ZigBee modules available easily in the market, which have such capability, other modules are very rare which have a mesh networking and are low cost as well as have enough support available for rapid development. Another alternative was Digi-Mesh from iDigi, capability for mesh networking with adequate support. For this purpose the chosen wireless module is the XBee Series 1 module, which can be a ZigBee, or iDigi module based on the firmware flashed. The advantages of each are detailed in the Networking section.
2.2.3 Development Platform

Once the processor and the ZigBee module was chosen the, the development platform was easy to select. The platform had to be small so that it could be used in the field inside an enclosure for testing. The ideal platform was Arduino Fio, which is a low cost development board with pin header to mount XBee module, which makes it suitable for this purpose.

2.2 Software Overview

The code design depended on the task, which had to be done, and in what order. The XBee modules have various sleep capabilities depending on the firmware used, ZigBee or Digi-mesh. But for the discussion henceforth we will assume the firmware is ZigBee, as ZigBee standard is more widely accepted though proprietary, but there are many manufacturers of ZigBee modules. The firmware affects the way code is written for them, so the code written will be written in a way that both the firmware can be used with minimal amount of change to the code.

One of the questions was, is a Real Time Operating System need for the System?
The answer was clearer after considering a RTOS; the best one to consider was FREERTOS, available from FREERTOS.org. It provides user with a lot of API and multitasking capabilities. RTOS has predictable response time and libraries for UART communication. Also to some extent a hardware independent code, but the complexity of the system increases many folds. The system resources being scarce in a small processor the RTOS utilizes a lot of it to provide features not really needed by our tasks. Thus needing more time to remove those capabilities from the RTOS without breaking other needed features.

Consider the following main functions of each node:

- **Calibration**
  
  This is the function Ideally called only once while installing the node, which does not need multitasking.

- **ReadRom**
  
  This Function reads from EEPROM, it is called once when the system boots up after installation or a battery change, thus also avoiding the need to be in a multitasking environment.

- **GetLocationStatus**
  
  This function can use Multi-tasking as the motor movement takes time and can be divided into sub tasks like, sensor reading and motor movement, which again depend on each other, not truly multi-task. The sensor can be read only after the motor has moved to the desired angle, so it will be a blocking task. Other tasks can be performed during the same time.

- **MakeDataPacket**
Once all the data is available from the sensors, a DataPacket is be updated to be sent over the network. Thus the function doesn’t necessarily need a multi tasking capability. It can be done in parts after each sensor reading is taken, or can be combined in the previous function to save time further.

- **SendPacket**

  This function sends the data packet prepared over the network to the Server. The DataPacket is encapsulated in the ZigBee API and sent to the module. It adds the prepared packet to the UART buffer, which is then used by the UART interrupt to send each byte from the buffer. Thus once the buffer is filled the processor is free to perform other tasks, thus a less time consuming function with no real need for multitasking.

Considering the following functions, only one function can benefit from an RTOS while others are functions, which do not have a time critical task or a need to multitask to optimize system performance. Proper partitioning of the code and following ANSI standards and using appropriate libraries can achieve other benefits of an RTOS such as hardware independent coding. They system uses lesser resources and is less complex to understand and debug.

A round robin task functionality was used for the software architecture, i.e. each function runs for a fixed amount of time and then the next function is called without the need of a task switching. For coding purposes the Arduino board was used, but the Arduino IDE and the programming language weren’t used as it has inefficiencies though simple to program. C was used for development, following AVR libc standards, which is closer to ANSI C standards thus easy hardware portability. More on Software architecture is covered in the later chapters.
2.3 Server

The server is any PC/ Embedded Device running Linux or windows with a MySQL database, Apache. Hardware requirements are serial ports or a USB ports, network connection for Ethernet connectivity. Only criterion being is for it to be fast enough to handle that amount of data over the serial ports and put it into the database, typically the amount of data for a parking lot would depend on the number of nodes. Assuming 3000 nodes in each parking lot, the maximum data packets per minute is 3000 packets per 30 seconds, assuming all the data flows through the network with minimum delay and lag. Which is not a very large number to handle for the PC, the bottleneck would be the slow wireless network.

A C++ based program is running on the PC end to receive all the data from the serial port and update the database. The tasks are done in two threads, one that receives data and puts it into a Queue, the next thread process the data and creates a database query and does a database entry. Thus the process is done in parallel avoiding any bottlenecks.

A HTML webpage displays the information from the database. It the status of each spot, time when the state of that location changed, a error is displayed in the table if there is a node which has been unable to communicate with the server for a duration of 10 min. Total number of spots and number of spots vacant is also displayed.

Further work can be done here to display statistics over time about the most amount of flow of cars during each hour of the day, which spots / parking lots are more used and which are not preferred.
CHAPTER 3: SENSOR SELECTION

Choosing the right kind of sensors for robust vehicle detection in the parking lot is extremely crucial to this application. Obstacle detection can be done using many kinds of sensors. Ultrasonic sensors, laser based distance sensors, camera and laser based distance sensor. The main target here was to achieve maximum numbers of cars detection at the lowest cost possible. Laser and Ultrasonic were two choices in the beginning. But as I researched, found a lack of Laser based distance module with an analog or digital output. Laser based sensor would have been able to detect more cars compared to an ultrasonic sensor. But the laser modules available were all above 200$ and thus not feasible in large quantities. Another factor limiting the number of cars a laser can detect was the height of the ceiling it was to be mounted on, if the ceiling height was low, the line of sight would be low and thus prohibiting the spots that be detected.

Ultrasonic sensors score over laser because of their cost, ruggedness and availability. They are not affected by the size or color of the object. They work well in harsh environments like that of a parking cellar. Since precision in the range measurement is not the main objective. Sensors used are 40 KHz - 43 KHz ultrasonic sensors; moreover they are inexpensive (10$ - 50 $) and easily available.

A 40 KHz Oscillator drives the ultrasonic transmitter crystal. The ultrasonic receiver crystal, receives a weak echo signal reflected from the surface of the obstacle (in our case, roof of a car). This echo needs to be amplified, rectified and filtered. The strength of the echo depends on the proximity of the obstacle (car). Since the ultrasonic energy can also be reflected by the other stray objects, and can lead to false detection. The echo signal strength varies from material to material. For example, the strength from a sheet of paper is very low and from a sheet of metal is high. Cars are made of mid steel and using the current setup, the echo from mid steel sheet was detected for various distances. [1]
3.1 Selecting the Right Sensor

3.1.1 Preliminary selection based on datasheets

Ultrasonic sensors I had selected were based on the distance it can measure up to, the beam width (To avoid interference from other objects) and power consumption.

In Fig 3.1 is the Ping))) sensor from Parallax. Its operating voltage is 5V, and max current 20mA. Measurement technique is time of flight. Distance measured was 3m, 9ft maximum which was less for our application. Also did not have power saving capabilities, would need extra hardware. Cost 28.99$[4]

Fig 3.1  PING))) Sensor from parallax

In Fig 3.2 Ultrasonic ranger from Devantech Ltd (Robot Electronics). Operating Voltage 5V, current typical 30mA max 50mA. Measurement technique is time of flight. Distance measured was 3m, 9ft maximum which was less for our application. Also did not have power saving capabilities, would need extra hardware. Cost 28.99$ [5]

Fig 3.2  Ultrasonic ranger from Devantech Ltd (Robot Electronics)

In Fig 3.3 URM V3.2 Ultrasonic sensor is shown. Operating Voltage 5V, current <20mA. Detecting range: 4cm-5m. Interface: PWM, RS232 or TTL (via Jumper). Distance it can measure is better than other sensors. It does not have power saving
capabilities, would need extra hardware. Cost 18$ [6]

Fig 3.3 URM V3.2 Ultrasonic sensor from LetsMakeRobots.com

Maxbotix has a line of Ultrasonic sensors, with operating voltage from 3.3V to 5V. Current required typical from 2mA to 3.4mA, and max current of 30mA for very short duration. Max distance from 765cm to 1068cm. PWM/Serial/Analog output for distance. Self-calibrating. Cost 22$ - 54$

Fig 3.4 Maxbotix Ultrasonic Sensors

3.1.2 Selecting a few from the above mentioned

To select the right ultrasonic sensor, multiple sensors were selected and tested on a test bench using a whiteboard as the target; the board size was 2 feet x 3 feet. Sensor was mounted on an Arduino Platform. Sensor was located at a fixed location and the white board was moved to measure the response of the Ultrasonic sensors.

Sensors used were:

- URM V3.2 Ultrasonic
  - Power: +5V
  - Current: <20mA
  - Working temperature: -10 ~ +70 C
  - Detecting range: 4cm-5m
  - Resolution: 1cm
  - Interface: PWM, RS232 or TTL (via Jumper)
  - Servo control: One servo control output
Operating Mode: Serial (PWM) passive control mode; Autonomous Mode; On/OFF Mode

Temperature sensor: 12 bits reading from serial port

Size: 22mm × 51 mm

- Maxbotix MB1240
  - 3 sensor outputs: Analog Voltage, Serial, Pulse Width
  - Resolution of 1 cm
  - Operates from 3.3-5.5V
  - Low 3.4mA average current requirement
  - 10Hz reading rate
  - Real-time automatic calibration (voltage, humidity, ambient noise)
  - Firmware filtering for better noise tolerance and clutter rejection
  - 200,000+ Hours Mean Time Between Failure
  - Great for large target applications
  - Maximum range of 765 cm (300 inches)

- Maxbotix MB1361
  - 3 sensor outputs: Analog Voltage, Serial, Analog Envelope
  - Resolution of 1 cm
  - Operates from 3.3-5.5V
  - Low 3.4mA average current requirement
  - Real-time automatic calibration (voltage, humidity, ambient noise)
  - Firmware filtering for better noise tolerance and clutter rejection
  - 200,000+ Hours Mean Time Between Failure
  - Maximum range of 1068 cm (420 inches)

- Maxbotix MB1040
  - 3 sensor outputs: Analog Voltage, Serial, Pulse Width
  - Resolution of 1 inch
  - Maximum Range of 254 inches (645 cm)
  - Operates from 2.5-5.5V
- Low 2.0mA average current requirement

- Maxbotix MB 7076
  - 3 select sensor outputs: Analog Voltage, Serial, Pulse Width
  - Resolution of 1 cm
  - Operates from 3.0-5.5V
  - Low 3.4mA average current requirement
  - Real-time automatic calibration (voltage, humidity, ambient noise)
  - Firmware filtering for better noise tolerance and clutter rejection
  - 200,000+ Hours Mean Time Between Failure
  - Maximum range of 765 cm (300 inches)

For the sensors various tests were conducted with the angle of field and distance it could detect up to, after all the results we concluded that MB1360 / MB 1260 could work the best for our project as it has long rage for wide objects and is reliable over a period of operations.
CHAPTER 4: WIRELESS NETWORK

Wireless Network is the backbone of this project. It needs to be robust, avoid interference and have necessary bandwidth to support the data flow. Also needs to have sleep modes and better synchronization for network so that the power saving and data flow is efficient.

Thus for these purposes two protocols ZigBee and DigiMesh have been selected and their advantages and disadvantages are listed below.

4.1 Wireless network stacks

4.1.1 ZigBee based on IEE standard 802.15.4

**Key Characteristics:** ZigBee is built on top of 802.15.4 using DSSS in the 2.4 GHz band. End points sleep, routers don't sleep, and a coordinator is needed to start the network and to allow points to join the network. ZigBee PRO includes key features for frequency agility, message fragmentation, and enhanced security associated with key management. The routing of messages follows the previously described cluster-tree methodology where routes to all points are maintained at each cluster. This allows a very short routing time, but requires lots of routes. Discovery of routes uses the AODV (Ad hoc On-Demand Distance Vector Routing) algorithm where paths are explored between clusters.

Network Architecture. The network (Figure 4.1) consists of three specific types of points. A ZigBee coordinator (ZC) is required for each network and initiates network formation; the coordinator may act as a router once the network is formed. The ZigBee router (ZR) is actually an optional network component, although a network without routers becomes a point-to-multipoint network; the router participates in multihop routing of messages. Finally, the ZigBee end device (ZED) does not allow association and does not participate in routing. As such it is often referred to as a child because it doesn't really have any responsibilities.[7]
**Strengths:** End devices are very low power because they are subservient to parental routers. Cluster-tree routing provides quick knowledge of routes and thereby efficient routing.

With ZigBee PRO, frequency agility switches away from problem channels automatically in a sort of on-demand frequency hopping. Message fragmentation support allows for long messages and security is flexible with support for separated keys. Finally, the network can scale to be very large.[7]

**Limitations:** The biggest limitation tends to be in terms of power in the routers. Routers must be powered; they can never go to sleep. In addition, cluster-tree routing means that network changes require a lot of route discovery traffic and heavy traffic volume means lots of collisions and potential message loss. Finally, a coordinator is needed to start and
manage the network, so if the coordinator goes down, no one can join and the network can't start. [7]

4.1.2 DigiMesh from Digi

**Key Characteristics:** Digi Mesh is designed to meet the need for very low power sensor networks where battery-powered routers are required. It is available in 2.4 GHz DSSS and 900 MHz FHSS. It does not rely on a full 802.15.4 implementation, as some of these functions are internal. For both message routing and discovery, it uses a variant of AODV (Ad hoc On-Demand Distance Vector Routing), so routing tables are built only for needed destinations. Because of this it is referred to as a peer-to-peer mesh instead of a cluster-tree.[7]

All nodes are viewed as equal participants; they are all routers and they can all sleep. Channel access is a sort of time-synchronized CSMA method, enabling bursty traffic, but offering few collisions. It has a full security suite.

**Network Architecture:** Figure 4.2 illustrates a typical ad hoc network topology. Unlike the cluster-tree method described in ZigBee, routes are determined on an as-needed basis, therefore routes that are never used, never get routing table entries and routes that are used frequently are continuously updated, optimizing their efficiency.
One of the other important points to note is that there is no coordinator or gateway function. Time synchronization is accomplished through a nomination and election process, enabling the network to operate autonomously.[7]

**Routing Methodology:** Fig 4.3 illustrates the process of handling routing failures. Fig 4.3A shows the initial network configuration where a route has been established from one point to another. Figure 4.3B illustrates a failure where one of the nodes has been removed for an unknown reason, removing relationships in the center of the route. Finally, Figure 4.3C shows how this route is reconstituted using a path that didn't previously exist. The relationships were there but they had never been used and they were newly discovered using AODV after the failure. [7]
**Strengths:** Every node is a router at very low power consumption. Because every message is acknowledged and routes are determined on an as needed basis, the network is not overwhelmed with unnecessary discovery traffic, which is very important if the routers are battery-powered and sleeping. Efficient route discovery and routing means that the network learns only routes that actually get used (AODV). Frequency agility is supported and security meets the requirements of both encryption and authentication.
Reliability is projected at 99.99%. Finally, the system supports larger payloads with support for message fragmentation.[7]

**Limitations:** Unfortunately, efficient power management means latency is long and nondeterministic. Even though time slots do not limit throughput, it is still limited, depending on loading and discoveries. The network can scale to a moderate size of around 500+ nodes and can be very large if traffic is light and message flow doesn't change much. [7]

4.2 Firmw**are best suited for our need**

From comparison of the two Network stacks it is clear that both have different advantages, ZigBee is more scalable and the end devices have good amount of power saving. But when the number of nodes gets very high and many routers exist in the same network, the route discovery gets complicated and slows down the complete network. Also the routers and the coordinator in ZigBee network cannot go to sleep, requiring being on mains supply. Whereas in DigiMesh all nodes can sleep and wake up in unison, also the network discovery is more efficient and simpler compared to ZigBee. Data reliability is termed to be very high at about 99.99%. But the network size is limited with the data packet size. Fortunately our data packet size is small about 30 bytes, and thus a little more than 500 nodes can exist on one channel, but this can be over come with multiple data sinks connected to the server. Thus both networks are feasible for our purpose depending on size of the parking lot and installation cost vs. maintenance cost preference.

Both the stacks have multiple ways to communicate; one is a transparent mode where the receiver address is fixed in the ZigBee or DigiMesh firmware, which is not suitable for a mesh network. For this we have an API communication method, which has a defined format of communication over UART. In API mode UART is configurable to different speeds and thus have flexibility for various devices, what this means is all nodes do not need to be on the same UART speed, this can help in making the end nodes a slower for
transmitting data and the receiving node faster so that the data is processed in a smoother flow.

There are two kinds of API modes AP = 1 & AP = 2, the first one does not need escape characters, but the AP = 2 mode needs escape characters for bytes 0x7E, 0x7D, 0x11, 0x13. This prevents the XBee node from erroneously detecting a frame start or having other communication errors as these bytes are special bytes in the protocol.

4.3 API Comparison for ZigBee & DigiMesh

Both the firmware API’s are compared for efficient code generation to keep as much code as possible common for both the firmware. Here only the relevant frame information is show.

4.3.1 API Frame Specification

![API Frame Specification for ZigBee](image-url)

When this API mode is enabled (AP = 2), the UART data frame structure is defined as follows:

**UART Data Frame Structure - with escape control characters:**

<table>
<thead>
<tr>
<th>Start Delimiter</th>
<th>Length (Bytes 2-3)</th>
<th>Frame Data (Bytes 4-n)</th>
<th>Checksum (Byte n + 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7E</td>
<td>MSB, LSB</td>
<td>API-specific Structure</td>
<td>1 Byte</td>
</tr>
</tbody>
</table>

Characters Escaped If Needed

MSB = Most Significant Byte, LSB = Least Significant Byte

Fig 4.4 API Frame Specification for ZigBee

![API Frame Specification for ZigBee](image-url)
Fig 4.5 API Frame Specification for DigiMesh

Comparing the ZigBee API Frame from Fig 4.4 and DigiMesh API Frame from Fig 4.5, we see that both have the same frame structure, thus enabling us to have a common encapsulation for both, API-Specific structure may differ.

Frame Data

Frame data of the UART data frame forms an API-specific structure as follows:

<table>
<thead>
<tr>
<th>UART Data Frame &amp; API-specific Structure:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Delimiter (Byte 1)</td>
</tr>
<tr>
<td>0x7E</td>
</tr>
<tr>
<td>cmdID</td>
</tr>
</tbody>
</table>

The cmdID frame (API-identifier) indicates which API messages will be contained in the cmdData frame (Identifier-specific data). Note that multi-byte values are sent big endian. The XBee modules support the following API frames:

<table>
<thead>
<tr>
<th>API Frame Names and Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Frame Names</td>
</tr>
<tr>
<td>AT Command</td>
</tr>
<tr>
<td>AT Command - Queue Parameter Value</td>
</tr>
<tr>
<td>ZigBee Transmit Request</td>
</tr>
<tr>
<td>Explicit Addressing ZigBee Command Frame</td>
</tr>
<tr>
<td>Remote Command Request</td>
</tr>
<tr>
<td>Create Source Route</td>
</tr>
<tr>
<td>AT Command Response</td>
</tr>
<tr>
<td>Mode Status</td>
</tr>
<tr>
<td>ZigBee Transmit Status</td>
</tr>
<tr>
<td>ZigBee Receive Packet (AO=0)</td>
</tr>
<tr>
<td>ZigBee Explicit Rx Indicator (AO=1)</td>
</tr>
<tr>
<td>ZigBee IO Data Sample Rx Indicator</td>
</tr>
<tr>
<td>XBee Sensor Read Indicator (AO=0)</td>
</tr>
<tr>
<td>Node Identification Indicator (AO=0)</td>
</tr>
<tr>
<td>Remote Command Response</td>
</tr>
<tr>
<td>Over-the-Air Firmware Update Status</td>
</tr>
<tr>
<td>Route Record Indicator</td>
</tr>
<tr>
<td>Many-to-One Route Request Indicator</td>
</tr>
</tbody>
</table>

Fig 4.6 API Frame Data for ZigBee
Comparing the API ID for DigiMesh Fig 4.7 and ZigBee Fig 4.6, we see that all the API ID in DigiMesh is the same as the one’s used in ZigBee, which implies that code written for ZigBee can work with DigiMesh. Almost all API’s that are needed are covered in DigiMesh.
### 4.3.2 Detailed Frame for Transmit Request

<table>
<thead>
<tr>
<th>Frame Fields</th>
<th>Offset</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Delimiter</td>
<td>0</td>
<td>0x7E</td>
<td></td>
</tr>
<tr>
<td>Length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frame ID</td>
<td>3</td>
<td>0x10</td>
<td></td>
</tr>
<tr>
<td>Frame Type</td>
<td>4</td>
<td>0x01</td>
<td>Identifies the UART data frame for the host to correlate with a subsequent ACK (acknowledgement). If set to 0, no response is sent.</td>
</tr>
<tr>
<td>64-bit Destination</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Network Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Broadcast Radius</td>
<td>15</td>
<td>0x00</td>
<td>Sets maximum number of hops a broadcast transmission can occur. If set to 0, the broadcast radius will be set to the maximum hops value.</td>
</tr>
<tr>
<td>Options</td>
<td>16</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>RF Data</td>
<td>17</td>
<td>0x54</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>25</td>
<td>0x13</td>
<td>0xFF - the 8 bit sum of bytes from offset 3 to this byte.</td>
</tr>
</tbody>
</table>

#### Details:
- **MSB 5**: 0x00
- **MSB 6**: 0x13
- **MSB 7**: 0xA2
- **MSB 8**: 0x00
- **MSB 9**: 0x40
- **MSB 10**: 0x0A
- **LSB 11**: 0x01
- **LSB 12**: 0x27
- **MSB 13**: 0xFF
- **LSB 14**: 0xFE

- **64-bit Destination Address**: Set to the 64-bit address of the destination device. The following addresses are also supported:
  - 0x0000000000000000 - Reserved 64-bit address for the coordinator
  - 0x00000000000000FF - Broadcast address

- **16-bit Destination Network Address**: Set to the 16-bit address of the destination device, if known. Set to 0xFFFF if the address is unknown, or if sending a broadcast.

- **Options**:
  - Bitfield of supported transmission options. Supported values include the following:
    - 0x01 - Disable retries and route repair
    - 0x20 - Enable APS encryption (if EE=1)
    - 0x40 - Use the extended transmission timeout
  - Enabling APS encryption presumes the source and destination have been authenticated. I also decreases the maximum number of RF payload bytes by 4 (below the value reported by NP).
  - The extended transmission timeout is needed when addressing sleeping end devices. It also increases the retry interval between retries to compensate for end device polling. See Chapter 4, Transmission Timeouts, Extended Timeout for a description.
  - Unused bits must be set to 0.
Comparing ZigBee Tx frame in Fig 4.8 and DigiMesh Tx frame in Fig 4.9, we see that only two fields differ namely in ZigBee 16-Bit destination address and Options, in the DigiMesh Tx Frame these fields are Reserved and Transmit options respectively. While
using the 64-but addressing the 16-bit address field needs to be set to 0xFF 0xFE, which is the same as DigiMesh and also using 0x00 in options field in ZigBee is for transmitting without an acknowledgement, this is done to reduce network traffic.

4.4 So which stack to use?

Comparing AT commands and other API frames we see that one API can work for both with minor adjustments. Thus the stack best suited for the parking lot needs can be used without much overhead of code editing. Also both have their advantages and disadvantages.

Also on detailed inspection we find that the data is already encapsulated and a Checksum is needed so the communication is reliable compared to the transparent mode. Also in API mode multiple requests are made before the packet is dropped. Thus ensuring much more reliability.
CHAPTER 5: Data Structures Used

Various data structures are used to store and transfer data. Once the data structures are defined it is easier to program them and write code accordingly.

5.1 Storing Data in the EEPROM

When the device is calibrated, the values are stored in the EEPROM, or when the battery is critically low the system sets a flag in the EEPROM and halts in sleep mode. The EEPROM needs to save calibration data, location angles with respect to the node orientation and also the base address of the server to communicate with.

typedef struct {
    long BaseAddrL;
    long BaseAddrH;
    char NumberOfLocations;
    int LocationAngle[5];
    int Location[5];
    int EmptyValue[5];
    long CRC;
} RomData;

The struct above is stored in the EEPROM, it saves the BaseAddress of the server, which it communicates to in two long data types ‘BaseAddrL’ & ‘BaseAddrH’. The number of spots that this particular node detects, in a char ‘NumberOfLocations’. Location angles of each spot with respect to the node orientation in an int array ‘LocationAngle’. Identities of the location it detects in an int array ‘Location’. The ADC values at which the location is Empty against which it will compare the new incoming values from the ADC in a int array ‘EmptyValue’. All the arrays are stored with respect to the location number corresponding array index of ‘Location’, so for Location[3], the LocationAngle is stored on LocationAngle[3] and ADC value on EmptyValue[3]. And finally the checksum, which is a sum of all the bytes, added in a long data type ‘CRC’.
5.2 Sending a Packet over Network

The network packet needs to have all the required data needed to update a location and the address the packet is being sent from. It needs to be compact enough for the network to be optimally used and easy enough to access for fast data manipulations inside the CPU node. It also needs to have the necessary battery status for that particular node. Each packet contains multiple location data, thus reducing numbers of packet that are sent over the network.

typedef struct {
    long BaseAddrL;
    long BaseAddrH;
    char NumberOfLocations;
    int Location[5];
    char LocationStatus[5];
    int BatteryStatus;
}DataPacket;

The data packet contains the address it need to send the data to in two long data type variables, ‘BaseAddrL’ & BaseAddrH’. The number of spots that particular node is detecting, in a char ‘NumberOfLocations’. The location identifiers in an int array ‘Location’. The status of each location, empty or occupied in a char array ‘LocationStatus’. Battery status for that node as an int ‘BatteryStatus’.

5.3 Database on the server

On the server side a MySQL database is used to store the data received from the sensor nodes. Reasons as to why a MySQL database was chosen:

- Fast and light weight compared to Oracle of MS SQL
- Open source and free to use
- Multi-platform, so no portability issues
- Robust data management with minimal hardware requirement
- Ease of use and support available from communities

MySQL database queries make it easy to find number of empty spots, nodes that are out of sync, location is empty, etc.

A single table is made for each parking lot, MySQL can handle up to 100k entries per table, which is very large compared to our requirement. Each parking spot has its individual entry in the table. The fields in the table are show in fig 5.1 below.

<table>
<thead>
<tr>
<th>Location</th>
<th>Address</th>
<th>Location Status</th>
<th>BatteryStatus</th>
<th>TimeIn</th>
<th>TimeOut</th>
<th>LastUpdate</th>
<th>Reserved</th>
<th>SkippedUpdate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>12345678</td>
<td>0</td>
<td>32</td>
<td>2012-04-20 00:56:38</td>
<td>2012-04-20 00:00:00</td>
<td>2012-04-20 00:00:00</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig 5.1: Snapshot of the database from MySQL admin page

*Location (int)*: This is the Primary key in the table. It is the unique value in the table, which is used to identify each location.

*Address (double)*: This is the node address for that location. It can be the same across more than one Location value

*Location Status (smallint)*: This is states if the location if empty or occupied.

*BatteryStatus (int)*: Battery Status for that node.

*TimeIn (TimeStamp)*: Time the Location Status changed from empty to occupied.

*TimeOut (TimeStamp)*: Time the location Status changed from occupied to empty.

*LastUpdate (TimeStamp)*: Time the last data packet from that node was received.

*Reserved (tinyint)*: This location specifies if the location is reserved for a category, for example disabled people, or residents or workers only.
SkippedUpdate (tinyint): If the LastUpdate has been after a threshold set, it will be set to 1 indicating a problem with the node.
CHAPTER 6: Power Optimization and battery performance

Each node is assumed to be battery operated, thus have a sleep cycle, which is in sync with the XBee wireless module. The main power drain is due to the Stepper Motor operation, but it has its own independent battery. The CPU is powered by a Polymer Lithium Ion Battery - 1000mAh. So the stepper power is optimized separately compared to the rest of the system.

The XBee wireless module has a pin ON_SLEEP, which determines the state of the module. This is used as an interrupt to sync the data transfer to the server. The synchronization is done with the CPU such that, all the sensor reading and packet has been formed before the XBee module wakes up and just as it does the

6.1 power Optimizations

6.1.1 Power Optimization in System Design

Taking advantage of ZigBee topology each node can sleep for maximum amount of time, as there is no Data being sent to each node. Each node transmits 32 bytes of Data per minute, which is the only data being transmitted. Routers and Coordinators never sleep, so sleep synchronization is not a problem.

For the system design let the node update time be 1 minute. From the stepper motor specification and the Ultrasonic sensor specification, the time it takes to get 5 readings is ~3.5 sec (100ms per ADC reading from sensor, and rest to move the stepper) To transmit 32bytes it takes 4.3 ms, which is derived from the battery calculator provided by ZigBee, shown in Fig 6.1. We can chronologically determine as to which module of the CPU need to be awake and the one’s that need to be in low power mode. Extra time is spent switching on and off processor modules, but it leads to power saving. For maximum power saving the CPU is put into power-down mode, and the Watchdog timer interrupt is used to wake up from sleep.
For every cycle that takes place every minute, assuming initialization is done and all processor modules are disabled other than the timer, for an time based interrupt.

<table>
<thead>
<tr>
<th>Time</th>
<th>Task Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_0) ((&lt; 1\mu s))</td>
<td>Wake up using watchdog timer Interrupt (Sleep for 5 seconds)</td>
</tr>
<tr>
<td>(T_1) ((&lt; 1\mu s))</td>
<td>Disable watchdog timer</td>
</tr>
<tr>
<td>(T_2) ((&lt; 300\text{ms}))</td>
<td>Initialize the stepper and move to next position</td>
</tr>
<tr>
<td>(T_3) ((&lt; 1\mu s))</td>
<td>Power down stepper</td>
</tr>
<tr>
<td>(T_4) ((&lt; 100\text{ms}))</td>
<td>Start ADC and get reading</td>
</tr>
<tr>
<td>(T_5) ((&lt; 1\mu s))</td>
<td>Power down ADC</td>
</tr>
<tr>
<td>(T_6) ((&lt; 1.6\text{s}))</td>
<td>Repeat steps (T_2 - T_5) all the locations have been covered (Assuming 5 locations per node for time calculation)</td>
</tr>
<tr>
<td>(T_7) ((&lt; 1\text{s}))</td>
<td>Reset Stepper Location and Power Down Stepper</td>
</tr>
<tr>
<td>(T_8) ((&lt; 1\mu s))</td>
<td>Check if XBee is awake, if yes jump to (T_{10})</td>
</tr>
<tr>
<td>(T_9) ((&lt; 1\mu s))</td>
<td>Sleep and wait for it to wake up</td>
</tr>
<tr>
<td>(T_{10}) ((&lt; 5\text{ms}))</td>
<td>Transmit data and reset timer value to be in sync with ZigBee</td>
</tr>
<tr>
<td>(T_{11}) ((&lt; 1\mu s))</td>
<td>Initialize watchdog timer and Deep sleep</td>
</tr>
</tbody>
</table>

1. UART is always enabled for the purpose if some data is available in sleep mode or module is out of sync.
2. Every time ADC is powered down the Ultrasonic sensor is also powered down.
3. Whenever the time for movement of sleep is more than ms, the processor goes to sleep and wakes up again to optimize power saving.
6.1.2 Power Optimization in Network

The ZigBee network does not have much scope for power optimization, as syncing nodes is not possible. But with DigiMesh stack, the sleep synchronization of all nodes is possible. Depending on the number of hops and distance from the server the nodes are awake, the node furthest from the server is awake for the least time and the node closest to the server is awake for the maximum amount of time to route all data. This compared to ZigBee network is still optimized in terms of power because the routers can sleep, but having them to be operated on a battery is not practical as the size of the network increases. For a small number of nodes battery operation is still possible over DigiMesh network. Battery calculations are shown in the next section

6.2 Battery Calculations for battery 1

The Polymer Lithium Ion Battery - 1000mAh shown in fir 6.1 is used for the CPU, Ultrasonic sensor and XBee.

Fig 6.1: 3.7V 1000mAh battery

Battery Calculation for the XBee is done below in fig 6.4, using tool available from DigiMesh website, but it only tells up about the battery life for the XBee. I have used http://oregonembedded.com/batterycalc.htm to calculate the battery life for the system battery life using estimated current consumption of AVR controller and Ultrasonic Sensor based on their data sheets.
The fig 6.2 shows the maximum current consumed by the AVR, when awake for 4 sec every minute and assuming always at peak current of 4.0mA. The average current for the time CPU is awake is lesser than 4.0mA as the processor sleeps for parts when the stepper is moving, when the sensor reading is taking place and when the Xbee starts transmitting, still using maximum current to calculate battery life.

**Battery Life Calculator**

<table>
<thead>
<tr>
<th>Capacity rating of battery (mAh)</th>
<th>1000</th>
<th>mAhr = milli-Amp-hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current consumption of device during sleep (mA)</td>
<td>0.001</td>
<td>mA = milli-Amps</td>
</tr>
<tr>
<td>Current consumption of device during wake (mA)</td>
<td>4</td>
<td>mA = milli-Amps</td>
</tr>
<tr>
<td>Number of wakeups per hour</td>
<td>60</td>
<td>If always on, enter 3600 here.</td>
</tr>
<tr>
<td>Duration of wake time (ms)</td>
<td>4000</td>
<td>ms = milli-Seconds. If always on, enter 1000 here.</td>
</tr>
</tbody>
</table>

The output of this calculator is an estimate, not a guarantee. Capacity is automatically derated by 15% to account for some self discharge. Other factors such as temperature can extend or shorten the battery life.

**Capacity of typical batteries that we're likely to use:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Manufacturer</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>1.2V NiCd</td>
<td>1500mAH</td>
</tr>
<tr>
<td>AA</td>
<td>2.4V NiCd</td>
<td>2500mAH</td>
</tr>
<tr>
<td>C</td>
<td>1.2V NiCd</td>
<td>8000mAH</td>
</tr>
<tr>
<td>D</td>
<td>1.2V NiCd</td>
<td>10000mAH</td>
</tr>
<tr>
<td>9V</td>
<td>9V NiCd</td>
<td>25000mAH</td>
</tr>
</tbody>
</table>

The fig 6.3 represents the Average current of the ultrasonic sensor, assuming the ultrasonic sensor takes reading 5 times every minute, so it stays on for 500ms, as each reading takes about 100ms. Now for the most part of the 100 ms the current is 5mA through the sensor, but when the sensor sends a ultrasonic pulse the current peaks to
40mA for 1ms, thus the average current comes to be 5.35 mA for 100ms. Using this to calculate the battery life.

**Battery Life Calculator**

<table>
<thead>
<tr>
<th>Capacity rating of battery (mAh)</th>
<th>1000 mAh = milli-Amp-hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current consumption of device during sleep (mA)</td>
<td>6.001 mA = milli-Amps</td>
</tr>
<tr>
<td>Current consumption of device during wake (mA)</td>
<td>5.35 mA = milli-Amps</td>
</tr>
<tr>
<td>Number of wakeups per hour</td>
<td>60 If always on, enter 3600 here.</td>
</tr>
<tr>
<td>Duration of wake time (ms)</td>
<td>500 ms = milli-Seconds. If always on, enter 1000 here.</td>
</tr>
</tbody>
</table>

The output of this calculator is an estimate, not a guarantee. Capacity is automatically derated by 15% to account for some self-discharge. Other factors such as temperature can extend or shorten the battery life.

**Capacity of typical batteries that we’re likely to use:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>1200 (alkaline) 800–1000 (Ni–Mh)</td>
</tr>
<tr>
<td>AA</td>
<td>2700 (alkaline) 3000 (Lithium–FeS2) 1700–2900 (NiMH)</td>
</tr>
<tr>
<td>C</td>
<td>8000 (alkaline) 4500–6000 (NiMH)</td>
</tr>
<tr>
<td>D</td>
<td>12000 (alkaline) 2200–12000 (NiMH) 19000 (Lithium-MnO2) 3.6V</td>
</tr>
<tr>
<td>9V Transistor</td>
<td>985 (alkaline) 220 (lithium) 175–300 (NiMH)</td>
</tr>
<tr>
<td>6V Lantern</td>
<td>26000 (alkaline)</td>
</tr>
<tr>
<td>CR2032</td>
<td>240 (Lithium-MnO2) 3.6V</td>
</tr>
<tr>
<td>1/10D</td>
<td>1000 (Lithium-MnO2) 3.6V</td>
</tr>
</tbody>
</table>

Fig 6.3: Battery life estimated for the sensor alone

The battery life for ZigBee end nodes is very similar to that of the DigiMesh end nodes, but as the DigiMesh nodes get close to the router the battery life decreases. Shown in the fig 6.3 below column one show average current for ZigBee and Digimesh end node for a battery of 1000mAh. Now as the nodes get closer, the number of bytes transmitted and received by the node increase thus decreasing battery life substantially. Which make the battery not feasible for nodes closer to the server. But if many nodes are closer to the
server the traffic on each node can be balanced by DigiMesh automatically and have a more optimized battery life.
Battery Life Calculator v2.2 © MaxStream, Inc.

**Overview**
- Use this calculator to:
  - a) recommend a battery capacity based on a target battery life.
  - b) estimate battery life for a given battery capacity.
  - c) compare overall power consumption scenarios.

**Details**
This calculator compares six power usage scenarios in vertical columns A-F. Configure each scenario by inputting the duration and current consumption for Sleep, Idle/Receive and Transmit states for an entire system.

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sleep</td>
<td>s</td>
<td>60</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Idle/Receive</td>
<td>ms</td>
<td>5</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transmit</td>
<td>ms</td>
<td>4.292</td>
<td>35.268</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radio type</td>
<td></td>
<td>XBee</td>
<td>XBee</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># of bytes transmitted</td>
<td></td>
<td>52</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Total System Current**
- Sleep mA: 0.05
- Idle/Receive mA: 50
- Transmit mA: 45

**Power usage comparison**
- Sleep %: 87.13
- Idle/Receive %: 7.26
- Transmit %: 5.61

Legend: Red > 100%, Green <= 100%

**Average current**
- mA: 0.05737675

**Design Goals**
- System efficiency: 90%
- Target battery life: 2 yrs
- Required battery capacity: mAh (3105.60)
- Given battery capacity: mAh (1000)
- Estimated battery life: yr (1.79)

**Scenario descriptions**
- Scenario A: ZigBee End node
- Scenario B: ZigBee node 3 steps to the router
- Scenario C:
- Scenario D:
- Scenario E:
- Scenario F:

**Reference**
- Pin Sleep (mA): 0.010 / 0.050
- Serial port Sleep (mA): n/a
- Cyclic Sleep (mA): 0.05
- Cyclic Sleep wake time (ms): 30
- Idle/Receive (mA): 50
- Transmit (mA): 45

---

Fig 6.4: Battery life calculated for ZigBee using the tool from DigiMesh
Now calculating overall battery life from the above figures, using average current from each.

Average current for the CPU per hour

\[ i_{cpu} = \frac{((4s \times 60) \times 4mA + ((3600s - (60 \times 4s)) \times 0.001))}{3600} \]

\[ i_{cpu} = 0.2676 \text{ mA} \]

Average current for the Sensor per hour

\[ i_{sen} = \frac{(((.5s \times 60) \times 5.35mA) + ((3600s - 60 \times 0.5s)) \times 0.001)}{3600} \]

\[ i_{sen} = 0.0455 \text{ mA} \]

From fig 6.3

\[ i_{xbee} = 0.0573 \text{ mA} \]

Calculating battery life based on these currents

Total current \[ i_{tot} = i_{cpu} + i_{sen} + i_{xbee} \]

\[ i_{tot} = 0.2676 + 0.0455 + 0.0573 = 0.3704 \]

Thus for a battery of 1000mAh, the battery life = \[ 1000 / 0.3704 = \sim 2699 \text{ hours} \]

Which is 122.4 days overall which includes battery degradation for 90% system efficiency included during calculation of each individual current.
6.2 Battery Calculations for battery 2

Battery currently used to run the Stepper motor is a 9v 1200 mA battery, purchased from http://www.batteryspace.com/primary-lithium-battery-9v-1200mah-10.8wh-50ma-rate.aspx. Shown in fig 6.5

![Fig 6.5: 9V 1200mAh Battery](image)

For the stepper motor the motor driver can go to sleep with consuming almost no power at all, thus saving power in sleep mode, the power used during runs is about 35mA from a test using multimeter. Calculations are shown below in fig 6.6. It shows that the battery can last for 94 days.
6.3 Suggested Battery Alternatives

The battery life the batteries used are less, suggested alternatives to increase battery life thus decreasing maintenance are:

1) DuraTrax NiMH 7.2V 2500mAh fig 6.7

- Capacity: 2500mAh
- Number of Cells in Series: 6
- Rated Voltage: 7.2V
- Min. Discharge Voltage: 5.4V
This battery is suggested as the replacement for the 9V 1200 mAh battery as more than doubles the battery life and has lower voltage resulting in lower currents when through the stepper. It is rechargeable so that during maintenance, a larger service battery can be attached temporarily to the node and can charge both batteries without having the need to replace them. Cost is 14$ retail, which is also very low for a rechargeable battery.[10]

2) Tenergy 6V 2000mAh in fig 6.8

- 6.0V 2000mAh NiMH Battery
- 5x AA NiMH 2000mah cells
- Rapidly charge up and long cycle life

This battery is suggested as replacement for 1000 mAh 3.7V battery, this increases the battery life more than 2x, as voltage down converted to 3.3V. It is again a rechargeable battery so the replacement / recharging cost is lower again. Cost 9.99$ [11]
CHAPTER 7: Failsafe used in the hardware & software

Failsafe mechanisms are needed in a system, as multiple points of failures are possible in any given system. We recognize the most important one’s and try and prevent them from causing errors or damage and shutdown the system or send a warning to the operator.

7.1 Failsafe Mechanisms in Hardware

In hardware multiple mechanisms to prevent failure are added to the design or are present in the controller.

7.1.1 Brown Out detection

The battery low voltage should be detected by the software mechanism, but if it fails there is an inbuilt mechanism in the controller called as Brownout detection, which keeps the CPU reset pin low for the period when the battery voltage falls below a certain voltage. This is helpful when the battery voltage is low, or battery life has degraded, for only CPU operations the voltage is above the threshold. But when the sensor or XBee transmits data, the current consumed increases which can cause the battery voltage to drop and detection from the CPU might take time, so the brown-out detection mechanism can prevent CPU data errors.

7.1.2 Separate Batteries for Motor and CPU

In normal operation a single battery with a step down converter can support both the CPU and the Motor. It has certain problems, in case of a mechanical failure or a obstacle in the path of sensor movement the motor can stall consuming significantly large amount of current, thus leaving the controller and XBee with very less or no power to work. Thus causing the complete system to fail. Another factor is that when the stepper motor moves the voltage fluctuation can be large across the CPU leading to errors, and false ADC readings. A separate battery prevents the system from failing.
7.1.3 Limiting Switch of Motor Reset

There is a limiting switch attached to the Motor Assembly, which is connected to a Digital IO pin. As the battery voltage gradually decreases the Motor performance and movement pattern changes, so as to have a zero reference, the limiting switch helps the motor detect the Zero and the 360 positions. Thus enabling to have a more reliable operation.

7.2 Failsafe Mechanisms in Software

In software multiple error can be caused, a corrupted EEROM or a low battery or a serial communication error. To detect them and prevent further damage failsafe mechanisms are built into the code.

7.2.1 ROM Checksum Error

When a controller EEPROM data is corrupted due to some reason, such as low voltage a checksum error can be detected. Thus when the error is detected an error flag is set and the calibration is done again. Thus whenever the Rom data is corrupted or missing a right message with the error flag is sent and after calibration the ROM checksum error should be resolved. But if the error persists it also can indicate an error with the EEPROM, thus requiring a controller replacement.

7.2.2 Data Packet ACK error

When a data packet is unable to be transmitted the XBee module sends a error packet to the CPU and the CPU resends the packet over the network. Although XBee already has an inbuilt mechanism for resending data multiple times, in case of a large network or
interference the through put can be limited thus retransmitting data can make it a little more robust.

7.2.3 Low Battery Detection

The system has two batteries, which are independent of each other. Each one is monitored and the battery status is sent back to the server. The server also has a low battery detection mechanism, so before the threshold is reached the battery can be recharged or replaced. But just incase the battery falls below the threshold, the data variables are stored into the ROM and the CPU goes to Extended sleep without waking up till manually reset.

7.2.4 Node Network related errors

On the server database, the timestamp for the time when a data packet was received is stored inside the table for each node. After every 10 minutes, the table is checked for last updated values that exceed 10 minutes, thus at the rate of update i.e. 1 minute, a node if it misses 10 updates continuously a error is set for that node and the operator can resolve the problem easily.
CHAPTER 8: Software Architecture (Device and Server)

The Parking Guidance System has two parts to the software, one for the nodes and one for the server. For the nodes two firmware are compiled using the same source code, one for the nodes which go to sleep, the end nodes in the ZigBee mesh network and one for the Nodes that are always awake the coordinators of the network.

8.1 Node software Architecture

The node software follows a round robin scheduling, where each function is called, each function is a non-blocking and a time limited function thus it is assured that the code will never be looping inside a function call forever. The functions are written using C & C++, with the help of available standar avr libc library, arduino-xbee library for arduino API handling and avr uart library which is modified for our use.

The flow chart for the software is given below in fig 8.1, fig 8.2 & fig 8.3. These figures help us understand the flow of the main program. I am going to describe each function in detail that has been listed in the flowchart.

All functions are detailed below:

**Int main (void) :**

Init_main();

```c
if(ReadRom(&CurrentSpot) == 1){
    Flag = Flag | CRCERROR;
    Calibrate = 1;
}
```

```c
packet1.NumberOfLocations =CurrentSpot.NumberOfLocations;
```
for (count = 0; count < 5; count++){
    packet1.Location[count] = CurrentSpot.Location[count];
}

for(i = 0; i < 20; i++){
    uart_flush();
    _delay_ms(10);
}

while(1){
    if(Calibrate == 0){

        //Get data from sensor, and save to cars
        resetStepper();
        DetectCars(&packet1, &CurrentSpot);
        packet1.BatteryStatus = CheckBattery();

        //Send packet over Zigbee
        SendPacket(&packet1);

        //Get all data from Various Spots based on EEPROM

        if(CheckBattery()){
            cli();
            sleep();
        }
        SyncXbee();
        WDT_on();
        if(SleepDone == 0){
            sleep();
        }
    } else {
        resetStepper();
        Calibrate();
        WriteRom();
    }
}
First in the main function, main _Init() function is called.

```c
void Init_main (void){
    SwitchPeripheralsOff();
    uart_init(UART_BAUD_SELECT(57600,16000000));
    uart_flush();
    sei();
    InitMotor();
    InitAtoDConverter();
    WDT_off();
}
```

In the main_Init all the hardware components are initialized and made available for use within the code. First all peripherals are switched off for saving power by disabling peripherals not used like SPI and timers.

Uart_init, initializes the USART for communication with the XBee module at 57600 baud rate at 16000000 Mhz CPU clock speed.

Uart_flush, empty’s the uart receive buffer, for pointers to the buffer to be initialized.

Sei() Sets the global interrupt enable.

InitMotor, Initializes the IO pins for the Stepper motor controller, i.e. three pins, one for stepping, one for direction control and one for power down mode.

InitAtoDConverter, initializes the ADC module and pins, to measure voltages from Ultrasonic sensor, and both the batteries.

WDT_Off, initializes the watchdog timer and switches it off until a sync with xbee timing module is done.
Fig 8.1: Main () flow chart part 1
Fig 8.2: Main () flow chart part 2
The next function called in the main readRom, which is responsible to read the EEPROM from address 0x0000 and fill up the data structure DataPacket. It checks for a checksum value called crc and verifies it and returns a value based on the verification passed or failed.

```c
int ReadRom(RomData *data) {
    int i;
    long CRC;
    CRC = 0;

    typedef union RomDataAccess {
        RomData NewData;
        char buffer[StructSize];
    } RomDataAccess;

    RomDataAccess temp;

    eeprom_read_block((void*)&temp.buffer, (const void*)0, sizeof(RomData));
```
data->BaseAddrL = temp.NewData.BaseAddrL;
CRC += data->BaseAddrL;

data->BaseAddrH = temp.NewData.BaseAddrH;
CRC += data->BaseAddrH;

data->NumberOfLocations = temp.NewData.NumberOfLocations;
CRC += data->NumberOfLocations;

for (i = 0; i < 5; i++){
    data->LocationAngle[i] = temp.NewData.LocationAngle[i];
    CRC += data->LocationAngle[i];
}

for (i = 0; i < 5; i++){
    data->Location[i] = temp.NewData.Location[i];
    CRC += data->Location[i];
}

for (i = 0; i < 5; i++){
    data->EmptyValue[i] = temp.NewData.EmptyValue[i];
    CRC += data->EmptyValue[i];
}

data->CRC = temp.NewData.CRC;

if (CRC == temp.NewData.CRC)
    return 0;
else {
    return 1;
}
}

Based on the return value we set a error flag and calibrate flag, which indicates that the systems has to be recalibrated.

After which the values read are put into a new structure called packet1 of type DataPacket, which is sent over the network to the server.
After which a while(1) loop is entered where the program control stays until a system reset is done.

The first function in the loop is a resetStepper function, which resets the stepper position with the help of a reset switch.

```c
void resetStepper(void){
    Move(-1*CurrentPosition);
    while(!(PORTD & RESET_PIN)){
        Move(-10);
    }
    CurrentPosition = 0;
}
```

In this function the stepper is moved back to position 0, and a global variable CurrentPosition is set.

Next in the loop a function is called to detect cars, which reads position of spots from the data packet and moves the stepper to those locations and returns the sensor voltage read via ADC.

```c
void DetectCars (DataPacket *p1, RomData *r1){
    int i;

    for(i = 0; i < p1->NumberOfLocations; i++){
        MoveToPosition(r1->LocationAngle[i]);
        p1->LocationStatus[i] = GetStatus(r1, i);
    }
}
```
A for loop is run where the number of time it loops is determined by the number of spots that node is monitoring. First the location angle is passed to a function MoveToPosition, which moves the stepper motor and hence the sensor to that location. After which a function to read ADC value from the sensor is called. This value is stored into the data packet. There needs to be some amount of time for the motor to stabilize after the quick movement, but the delay of 83ms inside the function GetStatus is enough for this to happen. This is repeated until all locations monitored are covered.

After which a function is called to check for battery condition, checkBattery();
This function returns the ADC value read from the battery and saves it into the data packet to be transmitted.

After which a again the CheckBattery function is called and checked against batter threshold values, if the battery is lower than the threshold a datapacket is sent with the appropriate flag and the processor is put to sleep until battery is replaced.

A sendPacket command follows the battery check which sends the data packet under normal conditions.

```c
void SendPacket (DataPacket *d1){
    XBeeAddress64 addr64 = XBeeAddress64((uint32_t)d1->BaseAddrH, (uint32_t)d1->BaseAddrL);

    // Create a TX Request
    ZBTxRequest zbTx = ZBTxRequest(addr64, (uint8_t*)d1, (uint8_t)sizeof(DataPacket));

    // Send your request
    xbee.send(zbTx);
}
```
The send packet function, initializes the address to which the data is to be sent, which is the hard coded server address. Address of the datapacket is passed and the send command is called from the XBee library.

SyncZigBee function puts the processor to sleep and checks for a global variable, which is set by the ZigBee sleep_on pin when it changes state. This pin is used to synchronize the timing with ZigBee to go to sleep and wake up 4s before the ZigBee module wakes up. ZigBee module is under cyclic sleep, so it wakes up for a short duration to transmit packets that are pending. The watchdog timer is enable after sync and the processor is put to sleep for 56 seconds. This is done by having an ISR which counts number of WDT interrupts and after 7 interrupts each spaced at 8 sec, it wakes up the processor and starts the loop again.

8.2 Server Software Architecture

The server has a C++ program using OpenFramework which is a cross platform framework for C++ applications. For the server prerequisites of software are:

Xcode 4 to compile the code.

MAMP software stack available freely, it includes Apache, MySQL and PHP.

We only use MySQL and Apache.

The main code for the server relies in the openframework class called TestApp

The code, which is responsible for reading data from serial port and inserting into the database is

```cpp
void testApp::update(){
    char temp;
```
```cpp
xbee.readPacket(100);

if (xbee.getResponse().isAvailable()) {
    // got something
    cout << "got something" << endl;
    if (xbee.getResponse().getApiId() == ZB_RX_RESPONSE) {
        // got a zb rx packet
        cout << "zb pack" << endl;
        // now fill our zb rx class
        xbee.getResponse().getZBRxResponse(rx);

        if (rx.getOption() == ZB_PACKET_ACKNOWLEDGED) {
            // the sender got an ACK
            ofLogVerbose("ACK done properly");
        } else {
            // we got it (obviously) but sender didn't get an ACK
            ofLogVerbose("ACK not received properly");
            insertDB(&DataAccess.NewData);
        }
    } else {
        // set dataLed PWM to value of the first byte in the data
        cout << "length of packet" << rx.getPacketLength() << endl;

        DataAccess newPacket;
        for (int i = 0; i < 52; i++) {
            newPacket.buffer[i] = rx.getData(i);
        }
        testData = (DataPacket *)rx.getData();
    }
} else if (xbee.getResponse().getApiId() == MODEM_STATUS_RESPONSE) {
    xbee.getResponse().getModemStatusResponse(msr);
    // the local XBee sends this response on certain events, like
    // association/dissociation

    if (msr.getStatus() == ASSOCIATED) {
        // yay this is great. flash led
    } else if (msr.getStatus() == DISASSOCIATED) {
```
// this is awful.. flash led to show our discontent

} else {
    // another status

} else {
    // not something we were expecting

} else if (xbee.getResponse().isError()) {
    //nss.print("Error reading packet. Error code: ");
    //nss.println(xbee.getResponse().getErrorCode());
}

}  

We continuously read from serial port, if data is available we push it into the database using function called insertDB.
CHAPTER 9: Hardware Schematic
MB1260 & MB1360 Circuit

The sensor functions using active components consisting of an LM324 and PIC16F690, together with a variety of other components. The schematic is shown to provide the user with detailed connection information.

Fig 9.3: Ultrasonic Sensor MB1360
REFERENCES

1. R Vishnubhotla, et al, “ZigBee Based Multi-Level Parking Vacancy Monitoring System”


APPENDIX

1. Tools & Software used for Development
   a. AVR Libc
   b. AVR Studio 5.1
   c. ArduUp hex uploader for Arduino
   d. Xcode for Server software
   e. MAMP package for database and webpage
   f. Dreamweaver 5.5 for making a html page with dynamic data list
   g. OpenFrameworks 0.7
   h. ofxMySql Addon

To Compile the source code for each node download and install avr studio 5.1
Unzip the NodeSourceCode.zip and open project file. Which can be compiled directly without any changes.

To compile and install Server software follow the following steps:

• Download and install Xcode 4.1
• Download and install MAMP package for OSX 10.7
• Let MySQL username be ‘root’ and password be ‘root’
• Download and unzip Openframeworks 0.7
• Download ofxMySql addon from openframeworks addon
• Unzip ofxMySql addon in addons folder under Openframeworks folder unzipped
• Add #include “ofxMySQL.h” to addons.h file
• Unzip the server source code and add under openframeworks/apps/myapps/parking
• OpenXcode project from parking folder and compile open frameworks first to compile addon and then compile parking app
2. Code for AVR Atmega328p based nodes:

Main.cpp

#include "defines.h"
// Avr Include files (Hardware specific)
#include <math.h>
#include <avr/io.h>
#include <avr/sleep.h>
#include <avr/interrupt.h>
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <util/delay.h>
#include <avr/wdt.h>

// Local files
#include "MotorController.h"
#include "SensorData.h"
#include "Sensors.h"
#include "PowerManagement.h"
#include "uart.h"
#include "Comm.h"
#include "wiring.h"
#include "adc.h"

void Init_main (void);
void Init_peripherals (void);
void OneSecDelay (void);
void OneMinuteDelay (void);

volatile int SleepDone = 0;
int ZigBee;
RomData CurrentSpot;
DataPacket packet1;
typedef union Access {
    DataPacket NewData;
    unsigned char buffer[sizeof(DataPacket)];
} Access;

Access atemp;

int Flag;
int count;
int Calibrate;

int main(void)
{
    int i;
    Init_main();
    packet1.Flags = 0x00;
    Calibrate = 0;
    SleepDone = 0;
    WriteRom();
    if(ReadRom(&CurrentSpot) == 1){
        Flag = Flag | CRCERROR;
        Calibrate = 0;
    }
    packet1.NumberOfLocations = CurrentSpot.NumberOfLocations;
    for (count = 0; count < 5; count++){
        packet1.Location[count] = CurrentSpot.Location[count];
    }
    for(i = 0; i < 20; i++){
        uart_flush();
        _delay_ms(10);
    }
    while(1){
        if(Calibrate == 0)
            //Get data from sensor, and save to cars
//Get all data from Various Spots based on EEPROM

if(CheckBattery(&packet1, &CurrentSpot)){
    //Send packet over Zigbee
    SendPacket(&packet1);
    //Sleep forever
    //cli();
    //sleep();
}

if(CheckAuxBattery(&packet1, &CurrentSpot)){
    //Send packet over Zigbee
    SendPacket(&packet1);
    //Sleep forever
    WDT_off();
    cli();
    ForeverSleep();
}

DetectCars(&packet1, &CurrentSpot);

    //Send packet over Zigbee
SendPacket(&packet1);

    WDT_on();
    while(SleepDone == 0)
        EnterSleep();

    SleepDone = 0;
    Init_peripherals();
} else {
    resetStepper();
    //Calibrate();
    WriteRom();
}
}

return 0;
}

//to avoid compile error
extern "C" void __cxa_pure_virtual() { while (1); }
uart_init(UART_BAUD_SELECT(57600,16000000));
uart_flush();
init_xbee();
sei();
InitMotor();
InitAtoDConverter();
WDT_off();
}

void Init_peripherals(void){
    uart_init(UART_BAUD_SELECT(57600,16000000));
    InitAtoDConverter();
    InitMotor();
}

void OneSecDelay(void){
    int i;
    for(i = 0; i < 100; i++){
        delay_ms(10);
    }
}

void OneMinuteDelay (void){
    int i;
    for(i = 0; i < 60; i++){
        OneSecDelay();
    }
}
adc.cpp

#include <avr/io.h>
#include "adc.h"
#include "uart.h"

void InitAtoDConverter(void)
{
    ADMUX = 0x6;

    //Set free running mode
    ADCSRB = 0;

    //Disable digital input for all pins
    DIDR0 = 0x3F;

    //Enable ADC, start conversion set auto trigger and pre-scalar according to F_CPU

    //Pre-scar 32 for 400Khz at 16Mhz
    ADCSRA = _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0);
}

int GetAdc(char PIN){

int temp =0;
l LONG adc_vtg;
if (PIN == AuxBatt){
    ADMUX = 0x06;
    ADCSRA = _BV(ADEN) | _BV(ADSC) | _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0);
    while((ADCSRA & _BV(ADSC)) == _BV(ADSC));
    //ADCSRA = _BV(ADEN) | _BV(ADSC) | _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0);
} else if(PIN == SensorPin){
    ADMUX = 0x07;
    ADCSRA = _BV(ADEN) | _BV(ADSC) | _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0);
    while((ADCSRA & _BV(ADSC)) == _BV(ADSC));
    //ADCSRA = _BV(ADEN) | _BV(ADSC) | _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0);
}

//while((ADCSRA & _BV(ADSC)) == _BV(ADSC));

temp = ADC;
ADCSRA = _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0);
return temp;
}

float GetBattery(void) {

    float vcc;
    unsigned int adc_data;
    ADMUX = 0x0E;
    ADCSRA = _BV(ADEN) | _BV(ADSC) | _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0);

    while((ADCSRA & _BV(ADSC)) == _BV(ADSC));
    ADCSRA = _BV(ADEN) | _BV(ADSC) | _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0);

    while((ADCSRA & _BV(ADSC)) == _BV(ADSC));

    adc_data = ADC; //read 8 bit value
    ADCSRA = _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0);
    vcc = 1.1 * 1023 / adc_data;

    return vcc;
adc.h

/*
 * adc.h
 *
 * Created: 4/13/2012 12:48:50 AM
 * Author: Aditya
 */

#ifndef ADC_H_
#define ADC_H_

// Prototypes

#define AuxBatt 1
#define SensorPin 2

void InitAtoDConverter(void);
int GetAdc(char PIN);
float GetBattery(void);

#endif /* ADC_H_ */
```cpp
#include <avr/eeprom.h>
#include "adc.h"
#include "uart.h"
#include "SensorData.h"
#include "adc.h"

extern RomData CurrentSpot;

void Calibrate(void) {
    long CRC;
    CRC = 0;

    typedef union RomDataAccess {
        RomData r1;
        char buffer[sizeof(RomData)];
    } RomDataAccess;

    RomDataAccess temp;

    temp.r1.BaseAddrL = 0x40619E8A;
    temp.r1.BaseAddrH = 0x0013A200;
    temp.r1.NumberOfLocations = 0x04;
    temp.r1.LocationAngle[0] = 60;
    temp.r1.LocationAngle[1] = 120;
    temp.r1.LocationAngle[2] = 200;
    temp.r1.LocationAngle[3] = 270;
    temp.r1.LocationAngle[4] = 0x00;
    temp.r1.Location[0] = 1000;
    temp.r1.Location[1] = 1001;
    temp.r1.Location[2] = 1002;
    temp.r1.Location[3] = 1003;
    temp.r1.Location[4] = 1004;
    temp.r1.EmptyValue[0] = 500;
    temp.r1.EmptyValue[1] = 500;
}```
temp.r1.CRC += temp.r1.BaseAddrH + temp.r1.BaseAddrL;
temp.r1.CRC += temp.r1.NumberOfLocations;


//eeprom_write_block((void*), (const void*)1, sizeof(RomData));
eeprom_write_block((const void*)&temp.buffer, (void*)0, sizeof(RomData));
}
calibration.h

/*
 * Calibration.h
 * Created: 4/22/2012 3:51:36 PM
 * Author: Aditya
 */

#ifndef CALIBRATION_H_
#define CALIBRATION_H_

void Calibrate(void);

#endif /* CALIBRATION_H_ */
#include <avr/io.h>
#include <math.h>
#include <util/delay.h>
#include "defines.h"
#include "Comm.h"
#include "Xbee.h"
#include "SensorData.h"
#include "uart.h"

volatile extern char UartBusy;
// Create an XBee object at the top of your sketch
XBee xbee = XBee();

void init_xbee(void){
    DDRB |= PIN_xbee;
    PORTB &= ~PIN_xbee;
}

void SendPacket (DataPacket *d1){
    PORTB &= ~PIN_xbee;
    _delay_ms(5);
    XBeeAddress64 addr64 = XBeeAddress64((uint32_t)d1->BaseAddrH, (uint32_t)d1->BaseAddrL);

    // Create a TX Request
    ZBTxRequest zbTx = ZBTxRequest(addr64, (uint8_t *)d1, (uint8_t)sizeof(DataPacket));

    // Send your request
    xbee.send(zbTx);
    while(UartBusy == 1);
    _delay_ms(5);
    PORTB |= PIN_xbee;
}

void SendDistress(DataPacket *d1){
    XBeeAddress64 addr64 = XBeeAddress64((uint32_t)d1->BaseAddrH, (uint32_t)d1->BaseAddrL);

    // Create a TX Request
    ZBTxRequest zbTx = ZBTxRequest(addr64, (uint8_t *)d1, (uint8_t)sizeof(DataPacket));

    // Send your request
```c
xbee.send(zbTx);
}

Comm.h

/*
 * Comm.h
 *
 * Created: 4/16/2012 7:19:27 PM
 * Author: Aditya
 */

#ifndef COMM_H_
#define COMM_H_

#include "SensorData.h"
#define PIN_xbee _BV(PB0)

void init_xbee(void);
void SendPacket (DataPacket *d1);
void SendDistress(DataPacket *d1);

#endif /* COMM_H_ */
```
defines.h

//
// defines.h
// ParkingLot
//
// Created by Aditya Gandhi on 3/22/12.
// Copyright (c) 2012 __MyCompany Name__. All rights reserved.
//

#ifndef ParkingLot_defines_h
#define ParkingLot_defines_h

/* CPU frequency */
#define F_CPU 16000000UL

/* UART baud rate */
#define UART_BAUD 57600

/* Whether to read the busy flag, or fall back to worst-time delays. */
#define USE_BUSY_BIT 1

//Motor
#define OFF 0
#define ON 1

//Power Mode
#define IDLE 0
#define ADCOnly 1
#define POWERON 2

//Flag Values
#define NETWORKERROR 0x01
#define BATTERYLOW 0x02
#define WATCHDOGERROR 0x04
#define CRCERROR 0x08
#define SLEEPING 0x01
#define AWAKE 0x00

void SyncTime (void);
void InterruptRoutine (void);
void SaveContents(void);
void SetErrorFlag(void);
#endif

MotorController.cpp

//
// MotorController.c
// ParkingLot
//
// Created by Aditya Gandhi on 3/25/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.
//

#include "defines.h"
#include <math.h>
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "MotorController.h"

unsigned int CurrentPosition = 0;
volatile char reset = 0;

void InitMotor(void){
    DDRB &= ~RESET_PIN;
    PORTB |= RESET_PIN;
    DDRD |= DIR_PIN | STEP_PIN | SLEEP_PIN;
    PORTD &= ~SLEEP_PIN;
    PORTD |= DIR_PIN;
    PORTD &= ~STEP_PIN;
    PCICR = _BV(PCIE0);
    PCMSK0 = _BV(PCINT1);
}

ISR(PCINT0_vect)
{
    reset = 0;
}

void MoveToPosition(int position){
    if(CurrentPosition > 360)
        resetStepper();

    Move(position - CurrentPosition);
CurrentPosition = position;

return;
}

void MotorEnable(void){
    PORTD |= SLEEP_PIN;
}

void MotorDisable(void){
    PORTD &= ~SLEEP_PIN;
}

void Move (int deg){
    int i;
    if(deg < 0){
        PORTD &= ~DIR_PIN;
        deg = deg * -1;
    } else{
        PORTD |= DIR_PIN;
    }
    if(deg > 0){
        float steps = deg / 0.935;
        float usDelay = (1/0.03) * 80;
        _delay_us(100);
        for(i=0; i < (unsigned long)steps; i++){  
            PORTD |= STEP_PIN;
            _delay_us(usDelay);
            PORTD &= ~STEP_PIN;
            _delay_us(usDelay);
        }
    }
    return;
}

void resetStepper(void){
    reset = 1;
    Move(-1*CurrentPosition);
    while(reset != 0x00){
        Move(-25);
    }
    CurrentPosition = 0;
MotorController.h

//
// MotorController.h
// ParkingLot
//
// Created by Aditya Gandhi on 3/25/12.
// Copyright (c) 2012 __MyCompany_Name__. All rights reserved.
//

ifndef MOTORCONTROLLER_H
#define MOTORCONTROLLER_H

define DIR_PIN _BV(PD6)
define STEP_PIN _BV(PD5)
define SLEEP_PIN _BV(PD2)
define RESET_PIN _BV(PB1)

void InitMotor(void);
void SetMotorPower (char PowerMode);
void resetStepper(void);
void MotorEnable(void);
void MotorDisable(void);
void MoveToPosition(int position);
void Move (int deg);
endif
PowerManagement.cpp

//
//  PowerManagement.c
//  ParkingLot
//
//  Created by Aditya Gandhi on 3/28/12.
//  Copyright (c) 2012 __MyCompanyName__. All rights reserved.
//

#include <stdio.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include "PowerManagement.h"

#define wdt_reset() __asm__ __volatile__ ("wdr")

int globalSleepCount = 0;
volatile extern int SleepDone;

ISR(WDT_vect){
    globalSleepCount++;
    if(globalSleepCount >= 3){
        SleepDone = 1;
        MCUSR = 0x00;
        WDTCSR |= (1<<WDCE) | (1<<WDE);
        WDTCSR = 0x00;
    } else{
        MCUSR = 0;
        WDTCSR |= (1<<WDCE) | (1<<WDE);
        WDTCSR = 0xE1;
        SleepDone = 0;
    }
}

void PowerDownModules(void){
    power_spi_disable();
    power_timer2_disable();
    power_twi_disable();
}

void EnterSleep (void){
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);
    power_adc_disable();
}
cli();
    sleep_enable();
    sleep_bod_disable();
    sei();
    sleep_cpu();
    sleep_disable();
    power_adc_enable();
    sei();
}

void ForeverSleep (void){
    set_sleep_mode(SLEEP_MODE_PWR_DOWN);
    power_adc_disable();
    cli();
    sleep_enable();
    sleep_bod_disable();
    sleep_cpu();
}

void WDT_off(void)
{
    cli();
    wdt_reset();
    /* Clear WDRF in MCUSR */
    MCUSR = 0x00;
    /* Write logical one to WDCE and WDE */
    /* Keep old prescaler setting to prevent unintentional time-out */
    WDTCSR |= (1<<WDCE) | (1<<WDE);
    /* Turn off WDT */
    WDTCSR = 0x00;
    globalSleepCount = 5;
    sei();
}

void WDT_on (void){
    cli();
    wdt_reset();
    /* Clear WDRF in MCUSR */
    MCUSR = 0;
    /* Write logical one to WDCE and WDE */
    /* Keep old prescaler setting to prevent unintentional time-out */
    WDTCSR |= (1<<WDCE) | (1<<WDE);
    /* Turn off WDT */
    WDTCSR = 0xE1;
    globalSleepCount = 0;
    sei();
}
PowerManagement.h

//
//  PowerManagement.h
//  ParkingLot
//
//  Created by Aditya Gandhi on 3/28/12.
//  Copyright (c) 2012 __MyCompanyName__. All rights reserved.
//

#ifndef POWERMANAGEMENT_H
#define POWERMANAGEMENT_H

#include <avr/power.h>

void PowerSaveMode (char Mode);
void ForeverSleep (void);
void PowerDownModules(void);
void ResetBrownOut (void);
void EnterSleep (void);
void WDT_off(void);
void WDT_on(void);

#endif
SensorData.cpp

//
// SensorData.c
// ParkingLot
//
// Created by Aditya Gandhi on 3/25/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.
//

#include <avr/eeprom.h>
#include "SensorData.h"

#define StructSize sizeof(RomData)

int ReadRom(RomData *data) {
    int i;
    long CRC;
    CRC = 0;

    typedef union RomDataAccess {
        RomData NewData;
        char buffer[StructSize];
    } RomDataAccess;

    RomDataAccess temp;

    eeprom_read_block((void*)&temp.buffer, (const void*)0, sizeof(RomData));

    data->BaseAddrL = temp.NewData.BaseAddrL;
    CRC += data->BaseAddrL;

    data->BaseAddrH = temp.NewData.BaseAddrH;
    CRC += data->BaseAddrH;

    data->NumberOfLocations = temp.NewData.NumberOfLocations;
    CRC += data->NumberOfLocations;

    for (i = 0; i < 5; i++){
        data->LocationAngle[i] = temp.NewData.LocationAngle[i];
        CRC += data->LocationAngle[i];
    }

    for (i = 0; i < 5; i++){
        data->Location[i] = temp.NewData.Location[i];
        CRC += data->Location[i];
    }
}
for (i = 0; i < 5; i++) {
    data->EmptyValue[i] = temp.NewData.EmptyValue[i];
    CRC += data->EmptyValue[i];
}

CRC += data->AuxBatVTG + data->BatteryVTG;

data->CRC = temp.NewData.CRC;

if (CRC == temp.NewData.CRC)
    return 0;
else {
    return 1;
}
}

// Write to rom

int WriteRom(void) {

    long CRC;
    CRC = 0;

    typedef union RomDataAccess {
        RomData NewData;
        char buffer[StructSize];
    } RomDataAccess;

    RomDataAccess temp;

    // Filling in temp data

    temp.NewData.BaseAddrL = 0x40619E8A;
    temp.NewData.BaseAddrH = 0x0013A200;
    temp.NewData.NumberOfLocations = 0x04;
    temp.NewData.LocationAngle[0] = 60;
    temp.NewData.LocationAngle[1] = 120;
    temp.NewData.LocationAngle[4] = 0x00;
    temp.NewData.Location[0] = 1005;
    temp.NewData.Location[1] = 1006;

    return CRC;
}
temp.NewData.EmptyValue[0] = 100;
temp.NewData.EmptyValue[1] = 100;
temp.NewData.AuxBatVTG = 900;
temp.NewData.BatteryVTG = 2500;

// Crc calculation

temp.NewData.CRC += temp.NewData.NumberOfLocations;


//eeprom_write_block((void*), (const void*)1, sizeof(RomData));
eeprom_write_block((const void**)&temp.buffer, (void*)0, sizeof(RomData));
return 0;
}
SensorData.h

//
//  SensorData.h
//  ParkingLot
//
//  Created by Aditya Gandhi on 3/25/12.
//  Copyright (c) 2012 __MyCompanyName__. All rights reserved.
//

#ifndef SENSORDATA_H
#define SENSORDATA_H

typedef struct {
   int location;
   int status;
} ParkingSpot;

typedef struct {
   long BaseAddrL;
   long BaseAddrH;
   char NumberOfLocations;
   int LocationAngle[5];
   int Location[5];
   int EmptyValue[5];
   int BatteryVTG;
   int AuxBatVTG;
   long CRC;
} RomData;

typedef struct {
   long BaseAddrL;
   long BaseAddrH;
   char NumberOfLocations;
   int Location[5];
   char LocationStatus[5];
   int BatteryStatus;
   int AuxBattStatus;
   char Flags;
} DataPacket;

int ReadRom(RomData *data);
int WriteRom(void);
void SetLocations (void);
void SetValues (void);
int JoinNetwork(int Address);

char GetCarLocation(char SpotNumber);
void Enable_sensor (void);
void Disable_sensor (void);
#endif
#include "defines.h"
#include <math.h>
#include <stdio.h>
#include <avr/io.h>
#include <util/delay.h>
#include "SensorData.h"
#include "Sensors.h"
#include "MotorController.h"
#include "adc.h"

#define SENSOR_PIN _BV(PD7)

void init_sensor(void){
    DDRD |= SENSOR_PIN;
    PORTD |= SENSOR_PIN;
}

void DetectCars (DataPacket *p1, RomData *r1){
    int i;
    MotorEnable();
    for(i = 0; i < p1->NumberOfLocations; i++){
        MoveToPosition(r1->LocationAngle[i]);
        p1->LocationStatus[i] = GetStatus(r1, i);
    }
    _delay_ms(10);
    resetStepper();
    MotorDisable();
}

char GetStatus(RomData *r1, int i){
    int temp;

    //Get Adc
    Enable_sensor();
    _delay_ms(83);
}
temp = GetAdc(SensorPin);
Disable_sensor();

if((r1->EmptyValue[i] - 50) > (temp))
    return 1;
else
    return 0;
}

int CheckBattery (DataPacket *p1, RomData *r1){
p1->BatteryStatus = (int) (GetBattery() * 1000.0);
if(p1->BatteryStatus < r1->BatteryVTG){
    return 1;
} else {
    return 0;
}
}

int CheckAuxBattery (DataPacket *p1, RomData *r1){
p1->AuxBattStatus = GetAdc(AuxBatt);
if(p1->AuxBattStatus < r1->BatteryVTG){
    return 1;
} else {
    return 0;
}
}

void Enable_sensor(void){
    PORTD &= ~SENSOR_PIN;
}

void Disable_sensor(void){
    PORTD |= SENSOR_PIN;
}
Sensors.h

//
// Sensors.h
// ParkingLot
//
// Created by Aditya Gandhi on 3/28/12.
// Copyright (c) 2012 __MyCompanyName__. All rights reserved.
//

#ifndef SENSORS_H_
#define SENSORS_H_
#include "SensorData.h"

int CheckBattery (DataPacket *p1, RomData *r1);
int CheckAuxBattery (DataPacket *p1, RomData *r1);
void DetectCars (DataPacket *p1, RomData *r1);
char GetStatus(RomData *r1, int i);
#endif
uart.cpp

*****************************************************************************
Title: Interrupt UART library with receive/transmit circular buffers
Author: Peter Fleury <pfleury@gmx.ch>  http://jump.to/fleury
File: Std: uart.c,v 1.6.2.1 2007/07/01 11:14:38 peter Exp $
Software: AVR-GCC 4.1, AVR Libc 1.4.6 or higher
Hardware: any AVR with built-in UART,
License: GNU General Public License

DESCRIPTION:
An interrupt is generated when the UART has finished transmitting or
receiving a byte. The interrupt handling routines use circular buffers
for buffering received and transmitted data.

The UART_RX_BUFFER_SIZE and UART_TXBUFFER_SIZE variables define
the buffer size in bytes. Note that these variables must be a
power of 2.

USAGE:
Refere to the header file uart.h for a description of the routines.
See also example test_uart.c.

NOTES:
Based on Atmel Application Note AVR306

LICENSE:
Copyright (c) 2006 Peter Fleury

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

*****************************************************************************

uart_available, uart_flush, uart1_available, and uart1_flush functions
were adapted from the Arduino HardwareSerial.h library by Tim Sharpe on
11 Jan 2009. The license info for HardwareSerial.h is as follows:

HardwareSerial.cpp - Hardware serial library for Wiring
Copyright (c) 2006 Nicholas Zambetti. All right reserved.
This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Modified 23 November 2006 by David A. Mellis
************************************************************************/

#include <defines.h>
#include <avr/io.h>

Date   Description
=====================================================================
====
05/11/2009 Changed all existing UARTx_RECEIVE_INTERRUPT and UARTx_TRANSMIT_INTERRUPT macros to use the "_vect" format introduced in AVR-Libc v1.4.0. Had to split the 3290 and 6490 out of their existing macro due to an inconsistency in the UART0_RECEIVE_INTERRUPT vector name (seems like a typo: USART_RX_vect for the 3290/6490 vice USART0_RX_vect for the others in the macro). Verified all existing macro register names against the device header files in AVR-Libc v1.6.6 to catch any inconsistencies.
05/12/2009 Added support for 48P, 88P, 168P, and 328P by adding them to the existing 48/88/168 macro. Added Arduino-style available() and flush() functions for both supported UARTs. Really wanted to keep them out of the library, so that it would be as close as possible to Peter Fleury's original library, but has scoping issues accessing internal variables from another program. Go C!
05/13/2009 Changed Interrupt Service Routine label from the old "SIGNAL" to the "ISR" format introduced in AVR-Libc v1.4.0.

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

#include "defines.h"
#include <avr/io.h>
volatile char UartBusy;

/**
 * constants and macros
 */

/* size of RX/TX buffers */
#define UART_RX_BUFFER_MASK ( UART_RX_BUFFER_SIZE - 1)
#define UART_TX_BUFFER_MASK ( UART_TX_BUFFER_SIZE - 1)

#if ( UART_RX_BUFFER_SIZE & UART_RX_BUFFER_MASK )
#error RX buffer size is not a power of 2
#endif
#if ( UART_TX_BUFFER_SIZE & UART_TX_BUFFER_MASK )
#error TX buffer size is not a power of 2
#endif

#if defined(__AVR_AT90S2313__) ||
 || defined(__AVR_AT90S4414__) || defined(__AVR_AT90S4434__) ||
 || defined(__AVR_AT90S8515__) || defined(__AVR_AT90S8535__) ||
 || defined(__AVR_ATmega103__)
/* old AVR classic or ATmega103 with one UART */
#define AT90_UART
#define UART0_RECEIVE_INTERRUPT   UART_RX_vect
#define UART0_TRANSMIT_INTERRUPT  UART_UDRE_vect
#define UART0_STATUS   UCSRA
#define UART0_CONTROL  UCSRB
#define UART0_DATA     UDR
#define UART0_UDRIE    UDRIE
#elif defined(__AVR_AT90S2333__) || defined(__AVR_AT90S4433__) ||
 || defined(__AVR_ATmega128__) || defined(__AVR_ATmega16__) ||
 || defined(__AVR_ATmega32__) || defined(__AVR_ATmega323__) ||
 /* ATmega with one USART */
#define ATMEGA_USART
#define UART0_RECEIVE_INTERRUPT   USART_RXC_vect

#if defined(__AVR_ATmega8__) || defined(__AVR_ATmega16__) ||
 || defined(__AVR_ATmega32__) ||
 || defined(__AVR_ATmega323__)
/* ATmega with one USART */
#define ATMEGA_USART
#define UART0_RECEIVE_INTERRUPT   USART_RXC_vect
#define UART0_TRANSMIT_INTERRUPT  USART_UDRE_vect
#define UART0_STATUS UCSRA
#define UART0_CONTROL UCSRB
#define UART0_DATA UDR
#define UART0_UDRIE UDRIE

#elif defined(__AVR_ATmega8515__) || defined(__AVR_ATmega8535__)
/* ATmega with one USART */
#define ATMEGA_USART
#define UART0_RECEIVE_INTERRUPT   USART_RX_vect
#define UART0_TRANSMIT_INTERRUPT  USART_UDRE_vect
#define UART0_STATUS UCSRA
#define UART0_CONTROL UCSRB
#define UART0_DATA UDR
#define UART0_UDRIE UDRIE

#elif defined(__AVR_ATmega163__)
/* ATmega163 with one UART */
#define ATMEGA_UART
#define UART0_RECEIVE_INTERRUPT   UART_RX_vect
#define UART0_TRANSMIT_INTERRUPT  UART_UDRE_vect
#define UART0_STATUS UCSRA
#define UART0_CONTROL UCSRB
#define UART0_DATA UDR
#define UART0_UDRIE UDRIE

#elif defined(__AVR_ATmega162__) /* ATmega with two USART */
#define ATMEGA_USART0
#define ATMEGA_USART1
#define UART0_RECEIVE_INTERRUPT   USART0_RXC_vect
#define UART1_RECEIVE_INTERRUPT   USART1_RXC_vect
#define UART0_TRANSMIT_INTERRUPT  USART0_UDRE_vect
#define UART1_TRANSMIT_INTERRUPT  USART1_UDRE_vect
#define UART0_STATUS UCSR0A
#define UART0_CONTROL UCSR0B
#define UART0_DATA UDR0
#define UART0_UDRIE UDRIE0
#define UART1_STATUS UCSR1A
#define UART1_CONTROL UCSR1B
#define UART1_DATA UDR1
#define UART1_UDRIE UDRIE1

#elif defined(__AVR_ATmega64__) || defined(__AVR_ATmega128__) /* ATmega with two UART */
#define ATMEGA_USART0
#define ATMEGA_USART1
#define UART0_RECEIVE_INTERRUPT   USART0_RX_vect
#define UART1_RECEIVE_INTERRUPT   USART1_RX_vect
#define UART0_TRANSMIT_INTERRUPT  USART0_UDRE_vect
#define UART1_TRANSMIT_INTERRUPT  USART1_UDRE_vect
#define UART0_STATUS UCSR0A
#define UART1_STATUS UCSR1A
#define UART0_CONTROL  UCSR0B
#define UART0_DATA     UDR0
#define UART0_UDRIE    UDRIE0
#define UART1_STATUS   UCSR1A
#define UART1_CONTROL  UCSR1B
#define UART1_DATA     UDR1
#define UART1_UDRIE    UDRIE1
#if defined(__AVR_ATmega161__)
  /* ATmega with UART */
  #error "AVR ATmega161 currently not supported by this library!"
#endif defined(__AVR_ATmega161__)
#if defined(__AVR_ATmega169__)
  /* ATmega with one UART */
  #define ATMEGA_USART
  #define UART0_RECEIVE_INTERRUPT   USART0_RX_vect
  #define UART0_TRANSMIT_INTERRUPT  USART0_UDRE_vect
  #define UART0_STATUS   UCSRA
  #define UART0_CONTROL  UCSRB
  #define UART0_DATA     UDR
  #define UART0_UDRIE    UDRIE
#elif defined(__AVR_ATmega48__) || defined(__AVR_ATmega88__) || defined(__AVR_ATmega168__) || defined(__AVR_ATmega48P__) || defined(__AVR_ATmega88P__) || defined(__AVR_ATmega168P__) || defined(__AVR_ATmega328P__)
  /* TLS-Added 48P/88P/168P/328P */
  /* ATmega with one UART */
  #define ATMEGA_USART0
  #define UART0_RECEIVE_INTERRUPT   USART0_RX_vect
  #define UART0_TRANSMIT_INTERRUPT  USART0_UDRE_vect
  #define UART0_STATUS   UCSRA
  #define UART0_CONTROL  UCSRB
  #define UART0_DATA     UDR
  #define UART0_UDRIE    UDRIE
#elif defined(__AVR_ATtiny2313__)
  #define ATMEGA_USART
  #define UART0_RECEIVE_INTERRUPT   USART0_RX_vect
  #define UART0_TRANSMIT_INTERRUPT  USART0_UDRE_vect
  #define UART0_STATUS   UCSRA
  #define UART0_CONTROL  UCSRB
  #define UART0_DATA     UDR
  #define UART0_UDRIE    UDRIE
#elif defined(__AVR_ATmega329__) || defined(__AVR_ATmega649__) || defined(__AVR_ATmega325__) || defined(__AVR_ATmega3250__) || defined(__AVR_ATmega645__) || defined(__AVR_ATmega6450__)
  /* ATmega with one UART */
#endif defined(__AVR_ATmega329__) || defined(__AVR_ATmega649__) || defined(__AVR_ATmega325__) || defined(__AVR_ATmega3250__) || defined(__AVR_ATmega645__) || defined(__AVR_ATmega6450__)
#define UART0_TRANSMIT_INTERRUPT USART0_UDRE_vect
#define UART0_STATUS UCSR0A
#define UART0_CONTROL UCSR0B
#define UART0_DATA UDR0
#define UART0_UDRIE UDRIE0
#if defined(__AVR_ATmega3290__) ||
    defined(__AVR_ATmega6490__)
    /* TLS-
    Separated these two from the previous group because of inconsistency in the
    USART_RX */
#elif defined(__AVR_ATmega2560__) || defined(__AVR_ATmega1280__) ||
    defined(__AVR_ATmega640__)
    /* ATmega with one USART */
#if defined(__AVR_ATmega2560__) || defined(__AVR_ATmega1280__) ||
    defined(__AVR_ATmega640__)
    /* ATmega with two USART */
#elif defined(__AVR_ATmega644__)  /* ATmega with one USART */
#else defined(__AVR_ATmega164P__) || defined(__AVR_ATmega324P__) ||
    defined(__AVR_ATmega644P__)  /* ATmega with two USART */
#elif defined(__AVR_ATmega644__)
    /* ATmega with one USART */
#else defined(__AVR_ATmega164P__) || defined(__AVR_ATmega324P__) ||
    defined(__AVR_ATmega644P__)
    /* ATmega with two USART */
#endif
#endif
#endif
```
#define UART0_RECEIVE_INTERRUPT   USART0_RX_vect
#define UART1_RECEIVE_INTERRUPT   USART0_UDRE_vect
#define UART0_TRANSMIT_INTERRUPT  USART1_RX_vect
#define UART1_TRANSMIT_INTERRUPT  USART1_UDRE_vect
#define UART0_STATUS   UCSR0A
#define UART0_CONTROL  UCSR0B
#define UART0_DATA     UDR0
#define UART0_UDRIE    UDRIE0
#define UART1_STATUS   UCSR1A
#define UART1_CONTROL  UCSR1B
#define UART1_DATA     UDR1
#define UART1_UDRIE    UDRIE1
#else
#error "no UART definition for MCU available"
#endif

/*
 *  module global variables
 */
static volatile unsigned char UART_TxBuf[UART_TX_BUFFER_SIZE];
static volatile unsigned char UART_RxBuf[UART_RX_BUFFER_SIZE];
static volatile unsigned char UART_TxHead;
static volatile unsigned char UART_TxTail;
static volatile unsigned char UART_RxHead;
static volatile unsigned char UART_RxTail;
static volatile unsigned char UART_LastRxError;
#if defined( ATMEGA_USART1 )
static volatile unsigned char UART1_TxBuf[UART_TX_BUFFER_SIZE];
static volatile unsigned char UART1_RxBuf[UART_RX_BUFFER_SIZE];
static volatile unsigned char UART1_TxHead;
static volatile unsigned char UART1_TxTail;
static volatile unsigned char UART1_RxHead;
static volatile unsigned char UART1_RxTail;
static volatile unsigned char UART1_LastRxError;
#endif

ISR(UART0_RECEIVE_INTERRUPT)
/*************************************************************************/
Function: UART Receive Complete interrupt
Purpose: called when the UART has received a character
**************************************************************************/
{
  unsigned char tmphead;
  unsigned char data;
unsigned char usr;
unsigned char lastRxError;

/* read UART status register and UART data register */
usr = UART0_STATUS;
data = UART0_DATA;

/* */
#if defined( AT90_UART )
    lastRxError = (usr & (_BV(FE)|_BV(DOR)) );
#elif defined( ATMEGA_USART )
    lastRxError = (usr & (_BV(FE)|_BV(DOR)) );
#elif defined( ATMEGA_USART0 )
    lastRxError = (usr & (_BV(FE0)|_BV(DOR0)) );
#elif defined ( ATMEGA_UART )
    lastRxError = (usr & (_BV(FE)|_BV(DOR)) );
#endif

/* calculate buffer index */
tmphead = ( UART_RxHead + 1 ) & UART_RX_BUFFER_MASK;

if ( tmphead == UART_RxTail ) {
    /* error: receive buffer overflow */
    lastRxError = UART_BUFFER_OVERFLOW >> 8;
} else{
    /* store new index */
    UART_RxHead = tmphead;
    /* store received data in buffer */
    UART_RxBuf[tmphead] = data;
}
UART_LastRxError = lastRxError;

ISR(UART0_TRANSMIT_INTERRUPT)
/*************************************************************************
Function: UART Data Register Empty interrupt
Purpose:  called when the UART is ready to transmit the next byte
**************************************
************************************/
{
    unsigned char tmptail;

    if ( UART_TxHead != UART_TxTail ) { 
        /* calculate and store new buffer index */
        tmptail = (UART_TxTail + 1) & UART_TX_BUFFER_MASK;
        UART_TxTail = tmptail;
    }
/* get one byte from buffer and write it to UART */
UART0_DATA = UART_TxBuf[tmptail];  /* start transmission */
UartBusy = 1;
}
else{
    /* tx buffer empty, disable UDRE interrupt */
    UART0_CONTROL &= ~_BV(UART0_UDRIE);
    UartBusy = 0;
}
}

/**********************************************
Function: uart_init()
Purpose: initialize UART and set baudrate
Input: baudrate using macro UART_BAUD_SELECT()
Returns: none
***********************************************/
void uart_init(unsigned int baudrate)
{
    UART_TxHead = 0;
    UART_TxTail = 0;
    UART_RxHead = 0;
    UART_RxTail = 0;

    #if defined( _AT90_UART )
    /* set baud rate */
    UBRR = (unsigned char)baudrate;

    /* enable UART receiver and transmitter and receive complete interrupt */
    UART0_CONTROL = _BV(RXCIE)|_BV(RXEN)|_BV(TXEN);
    
    #elif defined (ATMEGA_USART)
    /* Set baud rate */
    if ( baudrate & 0x8000 )
    {
        UART0_STATUS = (1<<U2X);  //Enable 2x speed
        baudrate &= ~0x8000;
    }
    UBRRH = (unsigned char)(baudrate>>8);
    UBRRL = (unsigned char) baudrate;

    /* Enable USART receiver and transmitter and receive complete interrupt */
    UART0_CONTROL = _BV(RXCIE)|(1<<RXEN)|(1<<TXEN);

    /* Set frame format: asynchronous, 8data, no parity, 1stop bit */
    #ifdef URSEL
    UCSRC = (1<<URSEL)|(3<<UCSZ0);
    #else

    100
UCSRC = (3<<UCSZ0);
#endif

#elif defined (ATMEGA_USART0)
    /* Set baud rate */
    if (baudrate & 0x8000)
        {
            UART0_STATUS = (1<<U2X0); //Enable 2x speed
            baudrate &= ~0x8000;
        }
    UBRR0H = (unsigned char)(baudrate>>8);
    UBRR0L = (unsigned char) baudrate;

    /* Enable USART receiver and transmitter and receive complete interrupt */
    UART0_CONTROL = _BV(RXCIE0)|(1<<RXEN0)|(1<<TXEN0);

    /* Set frame format: asynchronous, 8data, no parity, 1stop bit */
    #ifdef URSEL0
        UCSROC = (1<<URSEL0)|(3<<UCSZ00);
    #else
        UCSROC = (3<<UCSZ00);
    #endif
#endif

#elif defined( ATMEGA_UART)
    /* set baud rate */
    if (baudrate & 0x8000)
        {
            UART0_STATUS = (1<<U2X); //Enable 2x speed
            baudrate &= ~0x8000;
        }
    UBRRHI = (unsigned char)(baudrate>>8);
    UBRRR = (unsigned char) baudrate;

    /* Enable UART receiver and transmitter and receive complete interrupt */
    UART0_CONTROL = _BV(RXCIE)|(1<<RXEN)|(1<<TXEN);
#endif

} /* uart_init */

 /*************************************************************************
 Function: uart_getc()
 Purpose: return byte from ringbuffer
 Returns: lower byte: received byte from ringbuffer
         higher byte: last receive error
**************************************************************************/
unsigned int uart_getc(void)
{ 
    unsigned char tmptail;
    unsigned char data;

    if ( UART_RxHead == UART_RxTail ) {
        return UART_NO_DATA; /* no data available */
    }

    /* calculate /store buffer index */
    tmptail = (UART_RxTail + 1) & UART_RX_BUFFER_MASK;
    UART_RxTail = tmptail;

    /* get data from receive buffer */
    data = UART_RxBuf[tmptail];

    return (UART_LastRxError << 8) + data;
}

}/* uart_getc */

/*************************************************************************/
Function: uart_putc()
Purpose: write byte to ringbuffer for transmitting via UART
Input:    byte to be transmitted
Returns:  none
/***************************************************************************/
void uart_putc(unsigned char data)
{
    unsigned char tmphead;

    tmphead = (UART_TxHead + 1) & UART_TX_BUFFER_MASK;

    while ( tmphead == UART_TxTail ){
        /* wait for free space in buffer */
    }

    UART_TxBuf[tmphead] = data;
    UART_TxHead = tmphead;
    UartBusy = 1;
    /* enable UDRE interrupt */
    UART0_CONTROL |= _BV(UART0_UDRIE);
}/* uart_putc */

/*************************************************************************/
Function: uart_puts()
Purpose:  transmit string to UART
Input:    string to be transmitted
Returns:  none
***************************************************************************/
void uart_puts(const char *s )
{
    while (*s)
        uart_putchar(*s++);
}
/* uart_puts */

/****************************************************************************
Function: uart_puts_p()
Purpose:  transmit string from program memory to UART
Input:    program memory string to be transmitted
Returns:  none
***************************************************************************/
void uart_puts_p(const char *progmem_s )
{
    register char c;
    while ( (c = pgm_read_byte(progmem_s++)) )
        uart_putchar(c);
}
/* uart_puts_p */

/****************************************************************************
Function: uart_available()
Purpose:  Determine the number of bytes waiting in the receive buffer
Input:    None
Returns:  Integer number of bytes in the receive buffer
***************************************************************************/
int uart_available(void)
{
    return (UART_RX_BUFFER_MASK + UART_RxHead - UART_RxTail) %
            UART_RX_BUFFER_MASK;
}
/* uart_available */

/****************************************************************************
Function: uart_flush()
Purpose:  Flush bytes waiting the receive buffer. Actually ignores them.
Input:    None
Returns:  None
****************************************************************************/
void uart_flush(void)
{
    UART_RxHead = UART_RxTail;
} /* uart_flush */

/*
  * these functions are only for ATmegas with two USART
  */
#if defined( ATMEGA_USART1 )

SIGNAL(UART1_RECEIVE_INTERRUPT)
/*************************************************************************
Function: UART1 Receive Complete interrupt
Purpose: called when the UART1 has received a character
**************************************************************************/
{
    unsigned char tmphead;
    unsigned char data;
    unsigned char usr;
    unsigned char lastRxError;

    /* read UART status register and UART data register */
    usr  = UART1_STATUS;
    data = UART1_DATA;

    /* */
    lastRxError = (usr & (_BV(FE1)|_BV(DOR1)));

    /* calculate buffer index */
    tmphead = ( UART1_RxHead + 1 ) & UART_RX_BUFFER_MASK;

    if ( tmphead == UART1_RxTail ) {
        /* error: receive buffer overflow */
        lastRxError = UART_BUFFER_OVERFLOW >> 8;
    } else{
        /* store new index */
        UART1_RxHead = tmphead;
        /* store received data in buffer */
        UART1_RxBuf[tmphead] = data;
    }
    UART1_LastRxError = lastRxError;
}

104
SIGNAL(UART1_TRANSMIT_INTERRUPT)
/***************************************************************************/
Function: UART1 Data Register Empty interrupt
Purpose:  called when the UART1 is ready to transmit the next byte
***************************************************************************/
{
    unsigned char tmptail;

    if ( UART1_TxHead != UART1_TxTail) {
        /* calculate and store new buffer index */
        tmptail = (UART1_TxTail + 1) & UART_TX_BUFFER_MASK;
        UART1_TxTail = tmptail;
        /* get one byte from buffer and write it to UART */
        UART1_DATA = UART1_TxBuf[tmptail]; /* start transmission */
    } else {
        /* tx buffer empty, disable UDRE interrupt */
        UART1_CONTROL &= ~_BV(UART1_UDRIE);
    }
}

/***************************************************************************/
Function: uart1_init()
Purpose:  initialize UART1 and set baudrate
Input:    baudrate using macro UART_BAUD_SELECT()
Returns:  none
***************************************************************************/
void uart1_init(unsigned int baudrate)
{
    UART1_TxHead = 0;
    UART1_TxTail = 0;
    UART1_RxHead = 0;
    UART1_RxTail = 0;

    /* Set baud rate */
    if ( baudrate & 0x8000 )
    {
        UART1_STATUS = (1<<U2X1); //Enable 2x speed
        baudrate &= ~0x8000;
    }
    UBRR1H = (unsigned char)(baudrate>>8);
    UBRR1L = (unsigned char) baudrate;

    /* Enable USART receiver and transmitter and receive complete interrupt */
UART1_CONTROL = _BV(RXCIE1)|(1<<RXEN1)|(1<<TXEN1);

/* Set frame format: asynchronous, 8data, no parity, 1stop bit */
#endif
UCSR1C = (1<<URSEL1)|(3<<UCSZ10);
#else
UCSR1C = (3<<UCSZ10);
#endif
}/* uart_init */

/**********************************************************
****** Function: uart1_getc() ******
Purpose: return byte from ringbuffer
Returns: lower byte: received byte from ringbuffer
        higher byte: last receive error
**********************************************************/
unsigned int uart1_getc(void)
{
    unsigned char tmptail;
    unsigned char data;

    if ( UART1_RxHead == UART1_RxTail ) {
        return UART_NO_DATA; /* no data available */
    }

    /* calculate /store buffer index */
    tmptail = (UART1_RxTail + 1) & UART_RX_BUFFER_MASK;
    UART1_RxTail = tmptail;

    /* get data from receive buffer */
    data = UART1_RxBuf[tmptail];

    return (UART1_LastRxError << 8) + data;
}/* uart1_getc */

/**********************************************************
****** Function: uart1_putc() ******
Purpose: write byte to ringbuffer for transmitting via UART
Input: byte to be transmitted
Returns: none
**********************************************************/
void uart1_putc(unsigned char data)
{
    unsigned char tmphead;
tmphead = (UART1_TxHead + 1) & UART_TX_BUFFER_MASK;

while ( tmphead == UART1_TxTail ){
    /* wait for free space in buffer */
}

UART1_TxBuf[tmphead] = data;
UART1_TxHead = tmphead;

/* enable UDRE interrupt */
UART1_CONTROL |= _BV(UART1_UDRIE);

} /* uart1_putc */

/****************************************************************************
Function: uart1_puts()
Purpose: transmit string to UART1
Input: string to be transmitted
Returns: none
***************************************************************************/
void uart1_puts(const char *s )
{
    while (*s)
        uart1_putc(*s++);

} /* uart1_puts */

/****************************************************************************
Function: uart1_puts_p()
Purpose: transmit string from program memory to UART1
Input: program memory string to be transmitted
Returns: none
***************************************************************************/
void uart1_puts_p(const char *progmem_s )
{
    register char c;

    while ( (c = pgm_read_byte(progmem_s++)) )
        uart1_putc(c);

} /* uart1_puts_p */
Function: uart1_available()
Purpose: Determine the number of bytes waiting in the receive buffer
Input: None
Returns: Integer number of bytes in the receive buffer
**************************************************************************/

int uart1_available(void)
{
    return (UART_RX_BUFFER_MASK + UART1_RxHead - UART1_RxTail) %
    UART_RX_BUFFER_MASK;
}/* uart1_available */

**************************************************************************/
Function: uart1_flush()
Purpose: Flush bytes waiting the receive buffer. Actually ignores them.
Input: None
Returns: None
**************************************************************************/

void uart1_flush(void)
{
    UART1_RxHead = UART1_RxTail;
}/* uart1_flush */

#endif
uart.h

#ifndef UART_H
#define UART_H
/************************************************************************
Title:    Interrupt UART library with receive/transmit circular buffers
Author:   Peter Fleury <pfleury@gmx.ch>  http://jump.to/fleury
File:     $Id: uart.h,v 1.8.2.1 2007/07/01 11:14:38 peter Exp $
Software: AVR-GCC 4.1, AVR Libc 1.4
Hardware: any AVR with built-in UART, tested on AT90S8515 & ATmega8 at 4 Mhz
License:  GNU General Public License
Usage:    see Doxygen manual

LICENSE:
   Copyright (C) 2006 Peter Fleury

   This program is free software; you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation; either version 2 of the License, or
   any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

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

************************************************************************
uart_available, uart_flush, uart1_available, and uart1_flush functions
were adapted from the Arduino HardwareSerial.h library by Tim Sharpe on
11 Jan 2009.  The license info for HardwareSerial.h is as follows:

HardwareSerial.h - Hardware serial library for Wiring
Copyright (c) 2006 Nicholas Zambetti.  All right reserved.

   This library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   This library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with this library; if not, write to the Free Software
Changelog for modifications made by Tim Sharpe, starting with the current library version on his Web site as of 05/01/2009.

Date        Description
=====================================================================
05/12/2009  Added Arduino-style available() and flush() functions for both supported UARTs. Really wanted to keep them out of the library, so that it would be as close as possible to Peter Fleury's original library, but has scoping issues accessing internal variables from another program. Go C!

**
* @defgroup pfleury_uart UART Library
* @code #include <uart.h> @endcode
* 
* @brief Interrupt UART library using the built-in UART with transmit and receive circular buffers.
* 
* This library can be used to transmit and receive data through the built in UART.
* 
* An interrupt is generated when the UART has finished transmitting or receiving a byte. The interrupt handling routines use circular buffers for buffering received and transmitted data.
* 
* The UART_RX_BUFFER_SIZE and UART_TX_BUFFER_SIZE constants define the size of the circular buffers in bytes. Note that these constants must be a power of 2.
* 
* You may need to adapt this constants to your target and your application by adding
* CDEFS += -DUART_RX_BUFFER_SIZE=nn -DUART_RX_BUFFER_SIZE=nn to your Makefile.
* 
* @note Based on Atmel Application Note AVR306
* @author Peter Fleury pfleury@gmx.ch http://jump.to/fleury
*
/** @} */

#if (__GNUC__ * 100 + __GNUC_MINOR__) < 304
#error "This library requires AVR-GCC 3.4 or later, update to newer AVR-GCC compiler !"
#endif
```c
#define __AVR_ATmega328p__
/*
** constants and macros
*/

/** @brief UART Baudrate Expression
 * @param xtalcpu system clock in Mhz, e.g. 4000000L for 4Mhz
 * @param baudrate baudrate in bps, e.g. 1200, 2400, 9600
*/
#define UART_BAUD_SELECT(baudRate,xtalCpu) ((xtalCpu)/((baudRate)*16l)-1)

/** @brief UART Baudrate Expression for ATmega double speed mode
 * @param xtalcpu system clock in Mhz, e.g. 4000000L for 4Mhz
 * @param baudrate baudrate in bps, e.g. 1200, 2400, 9600
*/
#define UART_BAUD_SELECT_DOUBLE_SPEED(baudRate,xtalCpu) (((xtalCpu)/((baudRate)*8l)-1)|0x8000)

/** Size of the circular receive buffer, must be power of 2 */
#ifndef UART_RX_BUFFER_SIZE
#define UART_RX_BUFFER_SIZE 64
#endif

/** Size of the circular transmit buffer, must be power of 2 */
#ifndef UART_TX_BUFFER_SIZE
#define UART_TX_BUFFER_SIZE 64
#endif

/* test if the size of the circular buffers fits into SRAM */
#if ( (UART_RX_BUFFER_SIZE+UART_TX_BUFFER_SIZE) >= (0x8FF-0x60) )
#error "size of UART_RX_BUFFER_SIZE + UART_TX_BUFFER_SIZE larger than size of SRAM"
#endif

/*
** high byte error return code of uart_getc()
*/
#define UART_FRAME_ERROR 0x0800 /* Framing Error by UART */
#define UART_OVERRUN_ERROR 0x0400 /* Overrun condition by UART */
#define UART_BUFFER_OVERFLOW 0x0200 /* receive ringbuffer overflow */
#define UART_NO_DATA 0x0100 /* no receive data available */

/*
** function prototypes
*/
```

111
@brief Initialize UART and set baudrate
@param baudrate Specify baudrate using macro UART_BAUD_SELECT()
@return none
*/
void uart_init(unsigned int baudrate);

/**
 * @brief Get received byte from ringbuffer
 * Returns in the lower byte the received character and in the
 * higher byte the last receive error.
 * UART_NO_DATA is returned when no data is available.
 *
 * @param void
 * @return lower byte: received byte from ringbuffer
 * @return higher byte: last receive status
 * - \b 0 successfully received data from UART
 * - \b UART_NO_DATA
 * <br> no receive data available
 * - \b UART_BUFFER_OVERFLOW
 * <br> Receive ringbuffer overflow.
 * We are not reading the receive buffer fast enough,
 * one or more received character have been dropped
 * - \b UART_OVERRUN_ERROR
 * <br> Overrun condition by UART.
 * A character already present in the UART UDR register was
 * not read by the interrupt handler before the next character arrived,
 * one or more received characters have been dropped.
 * - \b UART_FRAME_ERROR
 * <br> Framing Error by UART
 */
unsigned int uart_getc(void);

/**
 * @brief Put byte to ringbuffer for transmitting via UART
 * @param data byte to be transmitted
 * @return none
 */
void uart_putchar(unsigned char data);

/**
 * @brief Put string to ringbuffer for transmitting via UART
 *
 * The string is buffered by the uart library in a circular buffer
 * and one character at a time is transmitted to the UART using interrupts.

* Blocks if it can not write the whole string into the circular buffer.
* @param s string to be transmitted
* @return none
*/
void uart_puts(const char *s);

/**
* @brief Put string from program memory to ringbuffer for transmitting via UART.
* The string is buffered by the uart library in a circular buffer
* and one character at a time is transmitted to the UART using interrupts.
* Blocks if it can not write the whole string into the circular buffer.
* @param s program memory string to be transmitted
* @return none
* @see uart_puts_P
*/
void uart_puts_p(const char *s);

#define uart_puts_P(__s)       uart_puts_p(PSTR(__s))

/**
* @brief Macro to automatically put a string constant into program memory
*/

/**
* @brief Return number of bytes waiting in the receive buffer
* @param none
* @return bytes waiting in the receive buffer
*/
int uart_available(void);

/**
* @brief Flush bytes waiting in receive buffer
* @param none
* @return none
*/
void uart_flush(void);

/** @brief Initialize USART1 (only available on selected ATmegas) @see uart_init */
extern void uart1_init(unsigned int baudrate);
/** @brief Get received byte of USART1 from ringbuffer. (only available on selected ATmega)
* @see uart_getc */
extern unsigned int uart1_getc(void);
/** @brief Put byte to ringbuffer for transmitting via USART1 (only available on selected
ATmega) @see uart_putc */
extern void uart1_putc(unsigned char data);
/** @brief Put string to ringbuffer for transmitting via USART1 (only available on selected
ATmega) @see uart_puts */
extern void uart1_puts(const char *s);
/** @brief Put string from program memory to ringbuffer for transmitting via USART1 (only
available on selected ATmega) @see uart_puts_p */
extern void uart1_puts_p(const char *s);
/** @brief Macro to automatically put a string constant into program memory */
#define uart1_puts_P(__s)       uart1_puts_p(PSTR(__s))
/** @brief Return number of bytes waiting in the receive buffer */
extern int uart1_available(void);
/** @brief Flush bytes waiting in receive buffer */
extern void uart1_flush(void);

/** @} */

#endif  // UART_H
wiring.c

/*
 w wiring.c - Partial implementation of the Wiring API for the ATmega8.
 Part of Arduino - http://www.arduino.cc/

 Copyright (c) 2005-2006 David A. Mellis

 This library is free software; you can redistribute it and/or
 modify it under the terms of the GNU Lesser General Public
 License as published by the Free Software Foundation; either
 version 2.1 of the License, or (at your option) any later version.

 This library is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 Lesser General Public License for more details.

 You should have received a copy of the GNU Lesser General
 Public License along with this library; if not, write to the
 Free Software Foundation, Inc., 59 Temple Place, Suite 330,
 Boston, MA 02111-1307 USA

 $Id$
 */

#include "wiring_private.h"

// the prescaler is set so that timer0 ticks every 64 clock cycles, and the
// the overflow handler is called every 256 ticks.
#define MICROSECONDS_PER_TIMER0_OVERFLOW (clockCyclesToMicroseconds(64 * 256))

// the whole number of milliseconds per timer0 overflow
#define MILLIS_INC (MICROSECONDS_PER_TIMER0_OVERFLOW / 1000)

// the fractional number of milliseconds per timer0 overflow. we shift right
// by three to fit these numbers into a byte. (for the clock speeds we care
// about - 8 and 16 MHz - this doesn't lose precision.)
#define FRACT_INC ((MICROSECONDS_PER_TIMER0_OVERFLOW % 1000) >> 3)
#define FRACT_MAX (1000 >> 3)

volatile unsigned long timer0_overflow_count = 0;
volatile unsigned long timer0_millis = 0;
static unsigned char timer0_fract = 0;

SIGNAL(TIMER0_OVF_vect)
{

115
// copy these to local variables so they can be stored in registers
// (volatile variables must be read from memory on every access)
unsigned long m = timer0_millis;
unsigned char f = timer0_fract;

m += MILLIS_INC;
f += FRACT_INC;
if (f >= FRACT_MAX) {
  f -= FRACT_MAX;
  m += 1;
}
timer0_fract = f;
timer0_millis = m;
timer0_overflow_count++;
}

unsigned long millis()
{
  unsigned long m;
  uint8_t oldSREG = SREG;

  // disable interrupts while we read timer0_millis or we might get an
  // inconsistent value (e.g. in the middle of a write to timer0_millis)
  cli();
  m = timer0_millis;
  SREG = oldSREG;

  return m;
}

unsigned long micros() {
  unsigned long m;
  uint8_t oldSREG = SREG, t;

  cli();
  m = timer0_overflow_count;
  t = TCNT0;

#ifdef TIFR0
  if ((TIFR0 & _BV(TOV0)) && (t < 255))
    m++;
#else
  if ((TIFR & _BV(TOV0)) && (t < 255))
    m++;
#endif

  SREG = oldSREG;
void delay(unsigned long ms)
{
    uint16_t start = (uint16_t)micros();

    while (ms > 0) {
        if (((uint16_t)micros() - start) >= 1000) {
            ms--;  
            start += 1000;
        }
    }
}

/* Delay for the given number of microseconds. Assumes a 8 or 16 MHz clock. */
void delayMicroseconds(unsigned int us)
{
    // calling avrlib's delay_us() function with low values (e.g. 1 or
    // 2 microseconds) gives delays longer than desired.
    //delay_us(us);

    #if F_CPU >= 16000000L
    // for the 16 MHz clock on most Arduino boards

    // for a one-microsecond delay, simply return. the overhead
    // of the function call yields a delay of approximately 1 1/8 us.
    if (--us == 0)
        return;

    // the following loop takes a quarter of a microsecond (4 cycles)
    // per iteration, so execute it four times for each microsecond of
    // delay requested.
    us <<= 2;

    // account for the time taken in the preceeding commands.
    us -= 2;
    #else
    // for the 8 MHz internal clock on the ATmega168

    // for a one- or two-microsecond delay, simply return. the overhead of
    // the function calls takes more than two microseconds. can't just
    // subtract two, since us is unsigned; we'd overflow.
    if (--us == 0)
        return;
    if (--us == 0)
        return;
    #else
    // for the 8 MHz internal clock on the ATmega168

    // for a one- or two-microsecond delay, simply return. the overhead of
    // the function calls takes more than two microseconds. can't just
    // subtract two, since us is unsigned; we'd overflow.
    if (--us == 0)
        return;
    if (--us == 0)
        return;
    #else
    // for the 8 MHz internal clock on the ATmega168

    // for a one- or two-microsecond delay, simply return. the overhead of
    // the function calls takes more than two microseconds. can't just
    // subtract two, since us is unsigned; we'd overflow.
    if (--us == 0)
        return;
    if (--us == 0)
        return;
    #else
    // for the 8 MHz internal clock on the ATmega168

    // for a one- or two-microsecond delay, simply return. the overhead of
    // the function calls takes more than two microseconds. can't just
    // subtract two, since us is unsigned; we'd overflow.
    if (--us == 0)
        return;
    if (--us == 0)
        return;
    #else
    // for the 8 MHz internal clock on the ATmega168

    // for a one- or two-microsecond delay, simply return. the overhead of
    // the function calls takes more than two microseconds. can't just
    // subtract two, since us is unsigned; we'd overflow.
    if (--us == 0)
        return;
    if (--us == 0)
        return;
    #else
    // for the 8 MHz internal clock on the ATmega168

    // for a one- or two-microsecond delay, simply return. the overhead of
    // the function calls takes more than two microseconds. can't just
    // subtract two, since us is unsigned; we'd overflow.
    if (--us == 0)
        return;
    if (--us == 0)
        return;
// the following loop takes half of a microsecond (4 cycles)  
// per iteration, so execute it twice for each microsecond of  
// delay requested.
us <<= 1;

// partially compensate for the time taken by the preceeding commands.  
// we can't subtract any more than this or we'd overflow w/ small delays.
us--;
#endif

    // busy wait
    __asm__ __volatile__ (  
"1: sbiw %0,1" "\n" // 2 cycles  
"brne 1b" : "=w" (us) : "0" (us) // 2 cycles
    );
}

void init()  
{
    // this needs to be called before setup() or some functions won't  
    // work there
    sei();

    // on the ATmega168, timer 0 is also used for fast hardware pwm  
    // (using phase-correct PWM would mean that timer 0 overflowed half as often  
    // resulting in different millis() behavior on the ATmega8 and ATmega168)
#if !defined(__AVR_ATmega8__)  
sbi(TCCR0A, WGM01);
sbi(TCCR0A, WGM00);
#endif
    // set timer 0 prescale factor to 64  
#if defined(__AVR_ATmega8__)  
sbi(TCCR0, CS01);
sbi(TCCR0, CS00);
#else  
sbi(TCCR0B, CS01);
sbi(TCCR0B, CS00);
#endif
    // enable timer 0 overflow interrupt  
#if defined(__AVR_ATmega8__)  
sbi(TIMSK, TOIE0);
#else  
sbi(TIMSK0, TOIE0);
#endif

    // timers 1 and 2 are used for phase-correct hardware pwm  
    // this is better for motors as it ensures an even waveform
// note, however, that fast pwm mode can achieve a frequency of up
// 8 MHz (with a 16 MHz clock) at 50% duty cycle

// set timer 1 prescale factor to 64
sbi(TCCR1B, CS11);
sbi(TCCR1B, CS10);
// put timer 1 in 8-bit phase correct pwm mode
sbi(TCCR1A, WGM10);

// set timer 2 prescale factor to 64
#if defined(__AVR_ATmega8__)
sbi(TCCR2, CS22);
#else
sbi(TCCR2B, CS22);
#endif
// configure timer 2 for phase correct pwm (8-bit)
#if defined(__AVR_ATmega8__)
sbi(TCCR2, WGM20);
#else
sbi(TCCR2A, WGM20);
#endif

#if defined(__AVR_ATmega1280__)
// set timer 3, 4, 5 prescale factor to 64
sbi(TCCR3B, CS31); sbi(TCCR3B, CS30);
sbi(TCCR4B, CS41); sbi(TCCR4B, CS40);
sbi(TCCR5B, CS51); sbi(TCCR5B, CS50);
// put timer 3, 4, 5 in 8-bit phase correct pwm mode
sbi(TCCR3A, WGM30);
sbi(TCCR4A, WGM40);
sbi(TCCR5A, WGM50);
#endif

// set a2d prescale factor to 128
// 16 MHz / 128 = 125 KHz, inside the desired 50-200 KHz range.
// XXX: this will not work properly for other clock speeds, and
// this code should use F_CPU to determine the prescale factor.
sbi(ADCSRA, ADPS2);
sbi(ADCSRA, ADPS1);
sbi(ADCSRA, ADPS0);

// enable a2d conversions
sbi(ADCSRA, ADEN);

// the bootloader connects pins 0 and 1 to the USART; disconnect them
// here so they can be used as normal digital i/o; they will be
// reconnected in Serial.begin()
#if defined(__AVR_ATmega8__)
UCSRB = 0;
#else
  UCSR0B = 0;
#endif
}
wiring.h

#include "binary.h"

#ifndef Wiring_h
#define Wiring_h

#define HIGH 0x1
#define LOW 0x0
#define INPUT 0x0
#define OUTPUT 0x1
#define true 0x1
#define false 0x0
#define PI 3.1415926535897932384626433832795
#define HALF_PI 1.5707963267948966192313216916398
#define TWO_PI 6.283185307179586476925286766559

#include <avr/io.h>

#ifdef __cplusplus
extern "C"
{
#endif

$Id$
*/
#define DEG_TO_RAD 0.017453292519943295769236907684886
#define RAD_TO_DEG 57.295779513082320876798154814105

#define SERIAL 0x0
#define DISPLAY 0x1

#define LSBFIRST 0
#define MSBFIRST 1

#define CHANGE 1
#define FALLING 2
#define RISING 3

#define INTERNAL 3
#define DEFAULT 1
#define EXTERNAL 0

// undefine stdlib's abs if encountered
#ifdef abs
#undef abs
#endif

#define min(a,b) ((a)<(b)?(a):(b))
#define max(a,b) ((a)>(b)?(a):(b))
#define abs(x) ((x)>0?(x):-(x))
#define constrain(amt,low,high) ((amt)<(low)?(low):((amt)>(high)?(high):(amt)))
#define round(x)     ((x)>=0?(long)((x)+0.5):(long)((x)-0.5))
#define radians(deg) ((deg)*DEG_TO_RAD)
#define degrees(rad) ((rad)*RAD_TO_DEG)
#define sq(x) ((x)*(x))
#define interrupts() sei()
#define noInterrupts() cli()

#define clockCyclesPerMicrosecond() ( F_CPU / 1000000L )
#define clockCyclesToMicroseconds(a) ( (a) / clockCyclesPerMicrosecond() )
#define microsecondsToClockCycles(a) ( (a) * clockCyclesPerMicrosecond() )
#define lowByte(w) ((uint8_t) ((w) & 0xff))
#define highByte(w) ((uint8_t) ((w) >> 8))

#define bitRead(value, bit) (((value) >> (bit)) & 0x01)
#define bitSet(value, bit) ((value) |= (1UL << (bit)))
#define bitClear(value, bit) ((value) &= ~(1UL << (bit)))
#define bitWrite(value, bit, bitvalue) (bitvalue ? bitSet(value, bit) : bitClear(value, bit))

typedef unsigned int word;
#define bit(b) (1UL << (b))

typedef uint8_t boolean;
typedef uint8_t byte;

void init(void);

void pinMode(uint8_t, uint8_t);
void digitalWrite(uint8_t, uint8_t);
int digitalRead(uint8_t);
int analogRead(uint8_t);
void analogReference(uint8_t mode);
void analogWrite(uint8_t, int);

unsigned long millis(void);
unsigned long micros(void);
void delay(unsigned long);
void delayMicroseconds(unsigned int us);
unsigned long pulseIn(uint8_t pin, uint8_t state, unsigned long timeout);

void shiftOut(uint8_t dataPin, uint8_t clockPin, uint8_t bitOrder, byte val);

void attachInterrupt(uint8_t, void (*)(void), int mode);
void detachInterrupt(uint8_t);

void setup(void);
void loop(void);

#ifdef __cplusplus
} // extern "C"
#endif

#endif
Xbee.cpp

/**
 * Copyright (c) 2009 Andrew Rapp. All rights reserved.
 *
 * This file is part of XBee-Arduino.
 *
 * XBee-Arduino is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * XBee-Arduino is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with XBee-Arduino. If not, see <http://www.gnu.org/licenses/>.
 */

#include "XBee.h"

//if defined(ARDUINO) && ARDUINO >= 100
   //#include "Arduino.h"
//else
   //#include "WProgram.h"
//#endif

#include <stdlib.h>
#include "defines.h"
#include "uart.h"
#include "wiring.h"

XBeeResponse::XBeeResponse() {
}

uint8_t XBeeResponse::getApiId() {
   return _apiId;
}

void XBeeResponse::setApiId(uint8_t apiId) {
   _apiId = apiId;
}

uint8_t XBeeResponse::getMsbLength() {
   return _msbLength;
XBeeResponse::setMsbLength(uint8_t msbLength) {
    _msbLength = msbLength;
}

uint8_t XBeeResponse::getLsbLength() {
    return _lsbLength;
}

XBeeResponse::setLsbLength(uint8_t lsbLength) {
    _lsbLength = lsbLength;
}

uint8_t XBeeResponse::getChecksum() {
    return _checksum;
}

XBeeResponse::setChecksum(uint8_t checksum) {
    _checksum = checksum;
}

uint8_t XBeeResponse::getFrameDataLength() {
    return _frameLength;
}

XBeeResponse::setFrameLength(uint8_t frameLength) {
    _frameLength = frameLength;
}

bool XBeeResponse::isAvailable() {
    return _complete;
}

XBeeResponse::setAvailable(bool complete) {
    _complete = complete;
}

bool XBeeResponse::isError() {
    return _errorCode > 0;
}

uint8_t XBeeResponse::getErrorCode() {
    return _errorCode;
}

XBeeResponse::setErrorCode(uint8_t errorCode) {
    _errorCode = errorCode;
}
void XBeeResponse::setCommon(XBeeResponse &target) {
    target.setApiId(getApiId());
    target.setAvailable(isAvailable());
    target.setChecksum(getChecksum());
    target.setErrorCode(getErrorCode());
    target.setFrameLength(getFrameDataLength());
    target.setMsbLength(getMsbLength());
    target.setLsbLength(getLsbLength());
}

#ifdef SERIES_2
ZBTxStatusResponse::ZBTxStatusResponse() : FrameIdResponse() {
}

uint16_t ZBTxStatusResponse::getRemoteAddress() {
    return (getFrameData()[1] << 8) + getFrameData()[2];
}

uint8_t ZBTxStatusResponse::getTxRetryCount() {
    return getFrameData()[3];
}

uint8_t ZBTxStatusResponse::getDeliveryStatus() {
    return getFrameData()[4];
}

uint8_t ZBTxStatusResponse::getDiscoveryStatus() {
    return getFrameData()[5];
}

bool ZBTxStatusResponse::isSuccess() {
    return getDeliveryStatus() == SUCCESS;
}

void XBeeResponse::getZBTxStatusResponse(XBeeResponse &zbXBeeResponse) {
    // way off?
    ZBTxStatusResponse* zb = static_cast<ZBTxStatusResponse*>(zbXBeeResponse);
    // pass pointer array to subclass
    zb->setFrameData(getFrameData());
    setCommon(zbXBeeResponse);
}
ZBRxResponse::ZBRxResponse(): RxDataResponse() {
  _remoteAddress64 = XBeeAddress64();
}

uint16_t ZBRxResponse::getRemoteAddress16() {
  return (getFrameData()[8] << 8) + getFrameData()[9];
}

uint8_t ZBRxResponse::getOption() {
  return getFrameData()[10];
}

// markers to read data from packet array. this is the index, so the 12th item in the array
uint8_t ZBRxResponse::getDataOffset() {
  return 11;
}

uint8_t ZBRxResponse::getDataLength() {
  return getPacketLength() - getDataOffset() - 1;
}

XBeeAddress64& ZBRxResponse::getRemoteAddress64() {
  return _remoteAddress64;
}

void XBeeResponse::getZBRxResponse(XBeeResponse &rxResponse) {
  ZBRxResponse* zb = static_cast<ZBRxResponse*>(&rxResponse);

  //TODO verify response api id matches this api for this response

  // pass pointer array to subclass
  zb->setFrameData(getFrameData());
  setCommon(rxResponse);

  zb->getRemoteAddress64().setMsb((uint32_t(unsigned long long(getFrameData())[0]) << 24) +
                                      (uint32_t(unsigned long long(getFrameData())[1]) << 16) +
                                      (uint16_t(unsigned long long(getFrameData())[2]) << 8) +
                                      getFrameData()[3]);
  zb->getRemoteAddress64().setLsb((uint32_t(unsigned long long(getFrameData())[4]) << 24) +
                                      (uint32_t(unsigned long long(getFrameData())[5]) << 16) +
                                      (uint16_t(unsigned long long(getFrameData())[6]) << 8) +
                                      (getFrameData()[7]));
}

ZBRxIoSampleResponse::ZBRxIoSampleResponse() : ZBRxResponse() {
}
// 64 + 16 addresses, sample size, option = 12 (index 11), so this starts at 12
uint8_t ZBRxIoSampleResponse::getDigitalMaskMsb() {
    return getFrameData()[12] & 0x1c;
}

uint8_t ZBRxIoSampleResponse::getDigitalMaskLsb() {
    return getFrameData()[13];
}

uint8_t ZBRxIoSampleResponse::getAnalogMask() {
    return getFrameData()[14] & 0x8f;
}

bool ZBRxIoSampleResponse::containsAnalog() {   
    return getAnalogMask() > 0;
}

bool ZBRxIoSampleResponse::containsDigital() {
    return getDigitalMaskMsb() > 0 || getDigitalMaskLsb() > 0;
}

bool ZBRxIoSampleResponse::isAnalogEnabled(uint8_t pin) {
    return ((getAnalogMask() >> pin) & 1) == 1;
}

bool ZBRxIoSampleResponse::isDigitalEnabled(uint8_t pin) {
    if (pin <= 7) {
        // added extra parens to calm avr compiler
        return ((getDigitalMaskLsb() >> pin) & 1) == 1;
    } else {
        return ((getDigitalMaskMsb() >> (pin - 8)) & 1) == 1;
    }
}

uint16_t ZBRxIoSampleResponse::getAnalog(uint8_t pin) {
    // analog starts 13 bytes after sample size, if no dio enabled
    uint8_t start = 15;
    if (containsDigital()) {
        // make room for digital i/o
        start += 2;
    }
    std::cout << "spacing is " << static_cast<unsigned int>(spacing) << std::endl;
    // start depends on how many pins before this pin are enabled
    for (int i = 0; i < pin; i++) {
        if (isAnalogEnabled(i)) {
start+=2;
}

// std::cout << "start for analog pin ["<< static_cast<unsigned int>(pin) << "]/sample " << static_cast<unsigned int>(sample) << " is ["<< static_cast<unsigned int>(start) << std::endl;

// std::cout << "returning index " << static_cast<unsigned int>(getSampleOffset() + start) << " and index " << static_cast<unsigned int>(getSampleOffset() + start + 1) << ", val is ["<< static_cast<unsigned int>(getFrameData()[getSampleOffset() + start]) << ["<< static_cast<unsigned int>(getFrameData()[getSampleOffset() + start + 1]) << std::endl;

return (uint16_t)((getFrameData()[start] << 8) + getFrameData()[start + 1]);

bool ZBRxIoSampleResponse::isDigitalOn(uint8_t pin) {
    if (pin <= 7) {
        // D0-7
        // DIO LSB is index 5
        return ((getFrameData()[16] >> pin) & 1) == 1;
    } else {
        // D10-12
        // DIO MSB is index 4
        return ((getFrameData()[15] >> (pin - 8)) & 1) == 1;
    }
}

void XBeeResponse::getZBRxIoSampleResponse(XBeeResponse &response) {
    ZBRxIoSampleResponse* zb = static_cast<ZBRxIoSampleResponse*>(&response);

    // pass pointer array to subclass
    zb->setFrameData(getFrameData());
    setCommon(response);

    zb->getRemoteAddress64().setMsb((uint32_t(getFrameData()[0]) << 24) + (uint32_t(getFrameData()[1]) << 16) + (uint16_t(getFrameData()[2]) << 8) + getFrameData()[3]);
    zb->getRemoteAddress64().setLsb((uint32_t(getFrameData()[4]) << 24) + (uint32_t(getFrameData()[5]) << 16) + (uint16_t(getFrameData()[6]) << 8) + (getFrameData()[7]));
}

#endif

#ifdef SERIES_1

RxResponse::RxResponse() : RxDataResponse() {

129
uint16_t Rx16Response::getRemoteAddress16() {
    return (getFrameData()[0] << 8) + getFrameData()[1];
}

XBeeAddress64& Rx64Response::getRemoteAddress64() {
    return _remoteAddress;
}

Rx64Response::Rx64Response() : RxResponse() {
    _remoteAddress = XBeeAddress64;
}

Rx16Response::Rx16Response() : RxResponse() {
}

RxIoSampleBaseResponse::RxIoSampleBaseResponse() : RxResponse() {
}

uint8_t RxIoSampleBaseResponse::getSampleOffset() {
    // sample starts 2 bytes after rssi
    return getRssiOffset() + 2;
}

uint8_t RxIoSampleBaseResponse::getSampleSize() {
    return getFrameData()[getSampleOffset()];
}

bool RxIoSampleBaseResponse::containsAnalog() {
    return (getFrameData()[getSampleOffset() + 1] & 0x7e) > 0;
}

bool RxIoSampleBaseResponse::containsDigital() {
    return (getFrameData()[getSampleOffset() + 1] & 0x1) > 0 ||
    getFrameData()[getSampleOffset() + 2] > 0;
}

//uint16_t RxIoSampleBaseResponse::getAnalog0(uint8_t sample) {
//    return getAnalog(0, sample);
//}

bool RxIoSampleBaseResponse::isAnalogEnabled(uint8_t pin) {
    return (((getFrameData()[getSampleOffset() + 1] >> (pin + 1)) & 1) == 1);
```cpp
bool RxIoSampleBaseResponse::isDigitalEnabled(uint8_t pin) {
  if (pin < 8) {
    return ((getFrameData()[getSampleOffset() + 4] >> pin) & 1) == 1;
  } else {
    return (getFrameData()[getSampleOffset() + 3] & 1) == 1;
  }
}

// // verified (from XBee-API)
// private int getSampleWidth() {
  int width = 0;
  // // width of sample depends on how many I/O pins are enabled. add one for each analog
  // that's enabled
  for (int i = 0; i <= 5; i++) {
    if (isAnalogEnabled(i)) {
      // each analog is two bytes
      width += 2;
    }
  }
  // if (this.containsDigital()) {
  //   // digital enabled takes two bytes, no matter how many pins enabled
  //   width += 2;
  // }
  return width;
}

// private int getStartIndex() {
  // int startIndex;
  // if (this.getSourceAddress() instanceof XBeeAddress16) {
  //   // 16 bit
  //   startIndex = 7;
  // } else {
  //   // 64 bit
  //   startIndex = 13;
  // }
  // return startIndex;
//
  // public int getDigitalMsb(int sample) {
  //   // msb digital always starts 3 bytes after sample size
  //   return;
```
// return this.getProcessedPacketBytes()[this.getStartIndex() + 3 + this.getSampleWidth() * sample];
// }
//
public int getDigitalLsb(int sample) {
    return this.getProcessedPacketBytes()[this.getStartIndex() + 3 + this.getSampleWidth() * sample + 1];
}
//
public Boolean isDigitalOn(int pin, int sample) {
    if (sample < 0 || sample >= this.getSampleSize()) {
        throw new IllegalArgumentException("invalid sample size: " + sample);
    }
    if (!this.containsDigital()) {
        throw new RuntimeException("Digital is not enabled");
    }
    if (pin >= 0 && pin < 8) {
        return ((this.getDigitalLsb(sample) >> pin) & 1) == 1;
    } else if (pin == 8) {
        // uses msb dio line
        return (this.getDigitalMsb(sample) & 1) == 1;
    } else {
        throw new IllegalArgumentException("Invalid pin: " + pin);
    }
}
//
public Integer getAnalog(int pin, int sample) {
    if (sample < 0 || sample >= this.getSampleSize()) {
        throw new IllegalArgumentException("invalid sample size: " + sample);
    }
    // analog starts 3 bytes after start of sample, if no dio enabled
    int startIndex = this.getStartIndex() + 3;
    //
    if (this.containsDigital()) {
        // make room for digital i/o sample (2 bytes per sample)
        startIndex+= 2;
    }
    //
    startIndex+= this.getSampleWidth() * sample;
    //
    // start depends on how many pins before this pin are enabled
    // this will throw IllegalArgumentException if invalid pin
    for (int i = 0; i < pin; i++) {

if (isAnalogEnabled(i)) {
    startIndex+=2;
}

return (this.getProcessedPacketBytes()[startIndex] << 8) +
    this.getProcessedPacketBytes()[startIndex + 1];

// THIS IS WRONG
uint16_t RxIoSampleBaseResponse::getAnalog(uint8_t pin, uint8_t sample) {

    // analog starts 3 bytes after sample size, if no dio enabled
    uint8_t start = 3;

    if (containsDigital()) {
        // make room for digital i/o sample (2 bytes per sample)
        start+=2*(sample + 1);
    }

    uint8_t spacing = 0;

    // spacing between samples depends on how many are enabled. add one for each analog
    that's enabled
    for (int i = 0; i <= 5; i++) {
        if (isAnalogEnabled(i)) {
            // each analog is two bytes
            spacing+=2;
        }
    }

    std::cout << "spacing is " << static_cast<unsigned int>(spacing) << std::endl;

    // start depends on how many pins before this pin are enabled
    for (int i = 0; i < pin; i++) {
        if (isAnalogEnabled(i)) {
            start+=2;
        }
    }

    start+= sample * spacing;

    std::cout << "start for analog pin [" << static_cast<unsigned int>(pin) << "]/sample " <<
        static_cast<unsigned int>(sample) << " is " << static_cast<unsigned int>(start) << std::endl;

    std::cout << "returning index " << static_cast<unsigned int>(getSampleOffset() + start) <<
        " and index " << static_cast<unsigned int>(getSampleOffset() + start + 1) << ", val is " <<
        static_cast<unsigned int>(getFrameData()[getSampleOffset() + start] << 8) << " and " << +
        static_cast<unsigned int>(getFrameData()[getSampleOffset() + start + 1] << 8) << " is " 
        << static_cast<unsigned int>(getFrameData()[getSampleOffset() + start + 2]) << " and " << static_cast…
static_cast<unsigned int>(getFrameData()[getSampleOffset() + start + 1]) << std::endl;

    return (uint16_t)((getFrameData()[getSampleOffset() + start] << 8) +
    getFrameData()[getSampleOffset() + start + 1]);
}

bool RxIoSampleBaseResponse::isDigitalOn(uint8_t pin, uint8_t sample) {
    if (pin < 8) {
        return ((getFrameData()[getSampleOffset() + 4] >> pin) & 1) == 1;
    } else {
        return (getFrameData()[getSampleOffset() + 3] & 1) == 1;
    }
}

//bool RxIoSampleBaseResponse::isDigital0On(uint8_t sample) {
//    return isDigitalOn(0, sample);
//}

Rx16IoSampleResponse::Rx16IoSampleResponse() : RxIoSampleBaseResponse() {
}

uint16_t Rx16IoSampleResponse::getRemoteAddress16() {
    return (uint16_t)((getFrameData()[0] << 8) + getFrameData()[1]);
}

uint8_t Rx16IoSampleResponse::getRssiOffset() {
    return 2;
}

void XBeeResponse::getRx16IoSampleResponse(XBeeResponse &response) {
    Rx16IoSampleResponse* rx = static_cast<Rx16IoSampleResponse*>(&response);
    rx->setFrameData(getFrameData());
    setCommon(response);
}

Rx64IoSampleResponse::Rx64IoSampleResponse() : RxIoSampleBaseResponse() {
    _remoteAddress = XBeeAddress64();
}

XBeeAddress64& Rx64IoSampleResponse::getRemoteAddress64() {
    return _remoteAddress;
}

uint8_t Rx64IoSampleResponse::getRssiOffset() {
return 8;
}

void XBeeResponse::getRx64IoSampleResponse(XBeeResponse &response) {
  Rx64IoSampleResponse* rx = static_cast<Rx64IoSampleResponse*>(&response);
  rx->setFrameData(getFrameData());
  setCommon(response);
  rx->getRemoteAddress64().setMsb((uint32_t(getFrameData()[0]) << 24) +
      (uint32_t(getFrameData()[1]) << 16) + (uint16_t(getFrameData()[2]) << 8) +
      getFrameData()[3]);
  rx->getRemoteAddress64().setLsb((uint32_t(getFrameData()[4]) << 24) +
      (uint32_t(getFrameData()[5]) << 16) + (uint16_t(getFrameData()[6]) << 8) +
      getFrameData()[7]);
}

TxStatusResponse::TxStatusResponse() : FrameIdResponse() {
}

uint8_t TxStatusResponse::getStatus() {
  return getFrameData()[1];
}

bool TxStatusResponse::isSuccess() {
  return getStatus() == SUCCESS;
}

void XBeeResponse::getTxStatusResponse(XBeeResponse &txResponse) {
  TxStatusResponse* txStatus = static_cast<TxStatusResponse*>(
      txResponse);
  // pass pointer array to subclass
  txStatus->setFrameData(getFrameData());
  setCommon(txResponse);
}

uint8_t RxResponse::getRssi() {
  return getFrameData()[getRssiOffset()];
}

uint8_t RxResponse::getOption() {
  return getFrameData()[getRssiOffset() + 1];
}

bool RxResponse::isAddressBroadcast() {
  return (getOption() & 2) == 2;
bool RxResponse::isPanBroadcast() {
    return (getOption() & 4) == 4;
}

uint8_t RxResponse::getDataLength() {
    return getPacketLength() - getDataOffset() - 1;
}

uint8_t RxResponse::getDataOffset() {
    return getRssiOffset() + 2;
}

uint8_t Rx16Response::getRssiOffset() {
    return RX_16_RSSI_OFFSET;
}

void XBeeResponse::getRx16Response(XBeeResponse &rx16Response) {
    Rx16Response* rx16 = static_cast<Rx16Response*>(&rx16Response);
    // pass pointer array to subclass
    rx16->setFrameData(getFrameData());
    setCommon(rx16Response);
    // rx16->getRemoteAddress16().setAddress((getFrameData()[0] << 8) + getFrameData()[1]);
}

uint8_t Rx64Response::getRssiOffset() {
    return RX_64_RSSI_OFFSET;
}

void XBeeResponse::getRx64Response(XBeeResponse &rx64Response) {
    Rx64Response* rx64 = static_cast<Rx64Response*>(&rx64Response);
    // pass pointer array to subclass
    rx64->setFrameData(getFrameData());
    setCommon(rx64Response);
    rx64->getRemoteAddress64().setMsb((uint32_t(getFrameData()[0]) << 24) +
          (uint32_t(getFrameData()[1]) << 16) + (uint16_t(getFrameData()[2]) << 8) +
          getFrameData()[3]);
    // rx64->getRemoteAddress64().setLsb((uint32_t(getFrameData()[4]) << 24) +
          (uint32_t(getFrameData()[5]) << 16) + (uint16_t(getFrameData()[6]) << 8) +
          getFrameData()[7]);
}
```cpp
#include <cstdint>

namespace XBee
{
    class XBeeResponse;
    class RemoteAtCommandResponse;

    class RemoteAtCommandResponse
    {
    public:

        RemoteAtCommandResponse();

        uint8_t* getCommand();
        uint8_t getStatus();
        bool isOk();
        uint8_t getValueLength();
        uint8_t* getValue();
        uint16_t getRemoteAddress16();
        XBeeAddress64& getRemoteAddress64();

    private:

        RemoteAtCommandResponse();

    public:

        uint8_t* getCommand();
        uint8_t getStatus();
        bool isOk();
        uint8_t getValueLength();
        uint8_t* getValue();
        uint16_t getRemoteAddress16();
        XBeeAddress64& getRemoteAddress64();

    public:

        // TODO no real need to cast. change arg to match expected class
        void getRemoteAtCommandResponse(XBeeResponse &response) {

            RemoteAtCommandResponse* at = static_cast<RemoteAtCommandResponse*>(response);
        }
    };

private:

    RemoteAtCommandResponse;

# endif

RemoteAtCommandResponse::RemoteAtCommandResponse() : AtCommandResponse() {

}

uint8_t* RemoteAtCommandResponse::getCommand() {
    return getFrameData() + 11;
}

uint8_t RemoteAtCommandResponse::getStatus() {
    return getFrameData()[13];
}

bool RemoteAtCommandResponse::isOk() {
    // weird c++ behavior. w/o this method, it calls AtCommandResponse::isOk(), which calls
    the AtCommandResponse::getStatus, not this.getStatus!!!
    return getStatus() == AT_OK;
}

uint8_t RemoteAtCommandResponse::getValueLength() {
    return getFrameDataLength() - 14;
}

uint8_t* RemoteAtCommandResponse::getValue() {
    if (getValueLength() > 0) {
        // value is only included for query commands. set commands does not return a value
        return getFrameData() + 14;
    }

    return NULL;
}

uint16_t RemoteAtCommandResponse::getRemoteAddress16() {
    return uint16_t((getFrameData()[9] << 8) + getFrameData()[10]);
}

XBeeAddress64& RemoteAtCommandResponse::getRemoteAddress64() {
    return _remoteAddress64;
}

void XBeeResponse::getRemoteAtCommandResponse(XBeeResponse &response) {

    // TODO no real need to cast. change arg to match expected class
    RemoteAtCommandResponse* at =
    static_cast<RemoteAtCommandResponse*>(response);
}

137
```
// pass pointer array to subclass
at->setFrameData(getFrameData());
setCommon(response);

    at->getRemoteAddress64().setMsb((uint32_t(getFrameData()[1]) << 24) +
        (uint32_t(getFrameData()[2]) << 16) + (uint16_t(getFrameData()[3]) << 8) +
        getFrameData()[4]);
    at->getRemoteAddress64().setLsb((uint32_t(getFrameData()[5]) << 24) +
        (uint32_t(getFrameData()[6]) << 16) + (uint16_t(getFrameData()[7]) << 8) +
        (getFrameData()[8]));

} RxDataResponse::RxDataResponse() : XBeeResponse() {

}

uint8_t RxDataResponse::getData(int index) {
    return getFrameData()[getDataOffset() + index];
}

uint8_t* RxDataResponse::getData() {
    return getFrameData() + getDataOffset();
}

FrameIdResponse::FrameIdResponse() {

}

uint8_t FrameIdResponse::getFrameId() {
    return getFrameData()[0];
}

ModemStatusResponse::ModemStatusResponse() {

}

uint8_t ModemStatusResponse::getStatus() {
    return getFrameData()[0];
}

void XBeeResponse::getModemStatusResponse(XBeeResponse &modemStatusResponse) {
    ModemStatusResponse* modem =
        static_cast<ModemStatusResponse*>(&modemStatusResponse);

    // pass pointer array to subclass
modem->setFrameData(getFrameData());
setCommon(modemStatusResponse);
}

AtCommandResponse::AtCommandResponse() {
}

uint8_t* AtCommandResponse::getCommand() {
    return getFrameData() + 1;
}

uint8_t AtCommandResponse::getStatus() {
    return getFrameData()[3];
}

uint8_t AtCommandResponse::getValueLength() {
    return getFrameDataLength() - 4;
}

uint8_t* AtCommandResponse::getValue() {
    if (getValueLength() > 0) {
        // value is only included for query commands. set commands does not return a value
        return getFrameData() + 4;
    }
    return NULL;
}

bool AtCommandResponse::isOk() {
    return getStatus() == AT_OK;
}

void XBeeResponse::getAtCommandResponse(XBeeResponse &atCommandResponse) {
    AtCommandResponse* at =
    static_cast<AtCommandResponse*>(&atCommandResponse);
    // pass pointer array to subclass
    at->setFrameData(getFrameData());
    setCommon(atCommandResponse);
}

uint16_t XBeeResponse::getPacketLength() {
    return ((_msbLength << 8) & 0xff) + (_lsbLength & 0xff);
}
uint8_t* XBeeResponse::getFrameData() {
    return _frameDataPtr;
}

void XBeeResponse::setFrameData(uint8_t* frameDataPtr) {
    _frameDataPtr = frameDataPtr;
}

void XBeeResponse::init() {
    _complete = false;
    _errorCode = NO_ERROR;
    _checksum = 0;
}

void XBeeResponse::reset() {
    init();
    _apiId = 0;
    _msbLength = 0;
    _lsbLength = 0;
    _checksum = 0;
    _frameLength = 0;

    _errorCode = NO_ERROR;

    for (int i = 0; i < MAX_FRAME_DATA_SIZE; i++) {
        getFrameData()[i] = 0;
    }
}

void XBee::resetResponse() {
    _pos = 0;
    _escape = false;
    _response.reset();
}

XBee::XBee(): _response(XBeeResponse()) {
    _pos = 0;
    _escape = false;
    _checksumTotal = 0;
    _nextFrameId = 0;

    _response.init();
    _response.setFrameData(_responseFrameData);
    // default
}

uint8_t XBee::getNextFrameId() {

_nextFrameId++;

if (_nextFrameId == 0) {
    // can't send 0 because that disables status response
    _nextFrameId = 1;
}

return _nextFrameId;
}

void XBee::begin(long baud) {
    uart_init((unsigned int)baud);
}

//void XBee::setSerial(HardwareSerial &serial) {
///_serial = &serial;
//#}

bool XBee::available() {
    return uart_available();
}

uint8_t XBee::read() {
    return uart_getc();
}

void XBee::flush() {
    uart_flush();
}

void XBee::write(uint8_t val) {
    uart_putc((unsigned char)val);
}

XBeeResponse& XBee::getResponse() {
    return _response;
}

// TODO how to convert response to proper subclass?
void XBee::getResponse(XBeeResponse &response) {
    response.setMsbLength(_response.getMsbLength());
    response.setLsbLength(_response.getLsbLength());
    response.setApiId(_response.getApiId());
    response.setFrameLength(_response.getFrameDataLength());
    response.setFrameData(_response.getFrameData());
}
void XBee::readPacketUntilAvailable() {
    while (!getResponse().isAvailable() || getResponse().isError()) {
        // read some more
        readPacket();
    }
}

bool XBee::readPacket(int timeout) {
    if (timeout < 0) {
        return false;
    }
    unsigned long start = millis();
    while (int((millis() - start)) < timeout) {
        readPacket();
        if (getResponse().isAvailable()) {
            return true;
        } else if (getResponse().isError()) {
            return false;
        }
    }
    // timed out
    return false;
}

void XBee::readPacket() {
    // reset previous response
    if (_response.isAvailable() || _response.isError()) {
        // discard previous packet and start over
        resetResponse();
    }
    while (available()) {
        b = read();
        if (_pos > 0 && b == START_BYTE && ATAP == 2) {
            // new packet start before previous packeted completed -- discard previous packet and start over
            _response.setErrorCode(UNEXPECTED_START_BYTE);
            return;
        }
    }
}
if (_pos > 0 && b == ESCAPE) {
    if (available()) {
        b = read();
        b = 0x20 ^ b;
    } else {
        // escape byte. next byte will be
        _escape = true;
        continue;
    }
}

if (_escape == true) {
    b = 0x20 ^ b;
    _escape = false;
}

// checksum includes all bytes starting with api id
if (_pos >= API_ID_INDEX) {
    _checksumTotal+= b;
}

switch(_pos) {
    case 0:
        if (b == START_BYTE) {
            _pos++;
        }
        break;
    case 1:
        // length msb
        _response.setMsbLength(b);
        _pos++;

        break;
    case 2:
        // length lsb
        _response.setLsbLength(b);
        _pos++;

        break;
    case 3:
        _response.setApiId(b);
        _pos++;

        break;
    default:
        // starts at fifth byte
if (_pos > MAX_FRAME_DATA_SIZE) {
    // exceed max size. should never occur

    _response.setErrorCode(PACKET_EXCEEDS_BYTE_ARRAY_LENGTH);
    return;
}

// check if we're at the end of the packet
// packet length does not include start, length, or checksum bytes, so add 3
if (_pos == (_response.getPacketLength() + 3)) {
    // verify checksum

    //std::cout << "read checksum " << static_cast<unsigned int>(b) << " at pos " << static_cast<unsigned int>(_pos) << std::endl;

    if ((_checksumTotal & 0xff) == 0xff) {
        _response.setChecksum(b);
        _response.setAvailable(true);

        _response.setErrorCode(NO_ERROR);
    } else {
        // checksum failed
        _response.setErrorCode(CHECKSUM_FAILURE);
    }
}

// minus 4 because we start after start,msb,lsb,api and up to but not including checksum
// e.g. if frame was one byte, _pos=4 would be the byte, pos=5 is the checksum, where end stop reading

    _response.setFrameLength(_pos - 4);

    // reset state vars
    _pos = 0;

    _checksumTotal = 0;

    return;
} else {
    // add to packet array, starting with the fourth byte of the apiFrame

        _response.getFrameData()[_pos - 4] = b;
        _pos++;
    }
}
// it's peanut butter jelly time!!

XBeeRequest::XBeeRequest(uint8_t apiId, uint8_t frameId) {
    _apiId = apiId;
    _frameId = frameId;
}

void XBeeRequest::setFrameId(uint8_t frameId) {
    _frameId = frameId;
}

uint8_t XBeeRequest::getFrameId() {
    return _frameId;
}

uint8_t XBeeRequest::getApiId() {
    return _apiId;
}

void XBeeRequest::setApiId(uint8_t apiId) {
    _apiId = apiId;
}

//void XBeeRequest::reset() {
//    _frameId = DEFAULT_FRAME_ID;
//}

//uint8_t XBeeRequest::getPayloadOffset() {
//    return _payloadOffset;
//}

//uint8_t XBeeRequest::setPayloadOffset(uint8_t payloadOffset) {
//    _payloadOffset = payloadOffset;
//}

PayloadRequest::PayloadRequest(uint8_t apiId, uint8_t frameId, uint8_t *payload, uint8_t payloadLength) : XBeeRequest(apiId, frameId) {
    _payloadPtr = payload;
    _payloadLength = payloadLength;
}

uint8_t* PayloadRequest::getPayload() {
    return _payloadPtr;
}

void PayloadRequest::setPayload(uint8_t* payload) {

}
_payloadPtr = payload;
}

uint8_t PayloadRequest::getPayloadLength() {
    return _payloadLength;
}

void PayloadRequest::setPayloadLength(uint8_t payloadLength) {
    _payloadLength = payloadLength;
}

XBeeAddress::XBeeAddress() {
}

XBeeAddress64::XBeeAddress64() : XBeeAddress() {
}

XBeeAddress64::XBeeAddress64(uint32_t msb, uint32_t lsb) : XBeeAddress() {
    _msb = msb;
    _lsb = lsb;
}

uint32_t XBeeAddress64::getMsb() {
    return _msb;
}

void XBeeAddress64::setMsb(uint32_t msb) {
    _msb = msb;
}

uint32_t XBeeAddress64::getLsb() {
    return _lsb;
}

void XBeeAddress64::setLsb(uint32_t lsb) {
    _lsb = lsb;
}

#if defined(SERIES_2)

ZBTxRequest::ZBTxRequest() : PayloadRequest(ZB_TX_REQUEST, DEFAULT_FRAME_ID, NULL, 0) {
}

#endif


ZBTxRequest::ZBTxRequest(XBeeAddress64 &addr64, uint16_t addr16, uint8_t broadcastRadius, uint8_t option, uint8_t *data, uint8_t dataLength, uint8_t frameId):
PayloadRequest(ZB_TX_REQUEST, frameId, data, dataLength) {
    _addr64 = addr64;
    _addr16 = addr16;
    _broadcastRadius = broadcastRadius;
    _option = option;
}

ZBTxRequest::ZBTxRequest(XBeeAddress64 &addr64, uint8_t *data, uint8_t dataLength):
PayloadRequest(ZB_TX_REQUEST, DEFAULT_FRAME_ID, data, dataLength) {
    _addr64 = addr64;
    _addr16 = ZB_BROADCAST_ADDRESS;
    _broadcastRadius = ZB_BROADCAST_RADIUS_MAX_HOPS;
    _option = ZB_TX_UNICAST;
}

uint8_t ZBTxRequest::getFrameData(uint8_t pos) {
    if (pos == 0) {
        return (_addr64.getMsb() >> 24) & 0xff;
    } else if (pos == 1) {
        return (_addr64.getMsb() >> 16) & 0xff;
    } else if (pos == 2) {
        return (_addr64.getMsb() >> 8) & 0xff;
    } else if (pos == 3) {
        return _addr64.getMsb() & 0xff;
    } else if (pos == 4) {
        return (_addr64.getLsb() >> 24) & 0xff;
    } else if (pos == 5) {
        return (_addr64.getLsb() >> 16) & 0xff;
    } else if (pos == 6) {
        return (_addr64.getLsb() >> 8) & 0xff;
    } else if (pos == 7) {
        return _addr64.getLsb() & 0xff;
    } else if (pos == 8) {
        return _addr16 >> 8) & 0xff;
    } else if (pos == 9) {
        return _addr16 & 0xff;
    } else if (pos == 10) {
        return _broadcastRadius;
    } else if (pos == 11) {
        return _option;
    } else {
        return getPayload()[pos - ZB_TX_API_LENGTH];
    }
}
uint8_t ZBTxRequest::getFrameDataLength() {
    return ZB_TX_API_LENGTH + getPayloadLength();
}

XBeeAddress64& ZBTxRequest::getAddress64() {
    return _addr64;
}

uint16_t ZBTxRequest::getAddress16() {
    return _addr16;
}

uint8_t ZBTxRequest::getBroadcast Radius() {
    return _broadcastRadius;
}

uint8_t ZBTxRequest::getOption() {
    return _option;
}

void ZBTxRequest::setAddress64(XBeeAddress64& addr64) {
    _addr64 = addr64;
}

void ZBTxRequest::setAddress16(uint16_t addr16) {
    _addr16 = addr16;
}

void ZBTxRequest::setBroadcastRadius(uint8_t broadcastRadius) {
    _broadcastRadius = broadcastRadius;
}

void ZBTxRequest::setOption(uint8_t option) {
    _option = option;
}

#endif

#ifdef SERIES_1

Tx16Request::Tx16Request() : PayloadRequest(TX_16_REQUEST, DEFAULT_FRAME_ID, NULL, 0) {
}

Tx16Request::Tx16Request(uint16_t addr16, uint8_t option, uint8_t *data, uint8_t dataLength, uint8_t frameId) : PayloadRequest(TX_16_REQUEST, frameId, data, dataLength) {
    _addr16 = addr16;
}
_option = option;
}

Tx16Request::Tx16Request(uint16_t addr16, uint8_t *data, uint8_t dataLength) :
PayloadRequest(TX_16_REQUEST, DEFAULT_FRAME_ID, data, dataLength) {
_addr16 = addr16;
_option = ACK_OPTION;
}

uint8_t Tx16Request::getFrameData(uint8_t pos) {
    if (pos == 0) {
        return (_addr16 >> 8) & 0xff;
    } else if (pos == 1) {
        return _addr16 & 0xff;
    } else if (pos == 2) {
        return _option;
    } else {
        return getPayload()[pos - TX_16_API_LENGTH];
    }
}

uint8_t Tx16Request::getFrameDataLength() {
    return TX_16_API_LENGTH + getPayloadLength();
}

uint16_t Tx16Request::getAddress16() {
    return _addr16;
}

void Tx16Request::setAddress16(uint16_t addr16) {
    _addr16 = addr16;
}

uint8_t Tx16Request::getOption() {
    return _option;
}

void Tx16Request::setOption(uint8_t option) {
    _option = option;
}

Tx64Request::Tx64Request() : PayloadRequest(TX_64_REQUEST, DEFAULT_FRAME_ID, NULL, 0) {
}

Tx64Request::Tx64Request(XBeeAddress64 &addr64, uint8_t option, uint8_t *data, uint8_t
dataLength, uint8_t frameId) : PayloadRequest(TX_64_REQUEST, frameId, data, dataLength) {
    _addr64 = addr64;
    _option = option;
}

Tx64Request::Tx64Request(XBeeAddress64 &addr64, uint8_t *data, uint8_t dataLength) :
PayloadRequest(TX_64_REQUEST, DEFAULT_FRAME_ID, data, dataLength) {
    _addr64 = addr64;
    _option = ACK_OPTION;
}

uint8_t Tx64Request::getFrameData(uint8_t pos) {
    if (pos == 0) {
        return (_addr64.getMsb() >> 24) & 0xff;
    } else if (pos == 1) {
        return (_addr64.getMsb() >> 16) & 0xff;
    } else if (pos == 2) {
        return (_addr64.getMsb() >> 8) & 0xff;
    } else if (pos == 3) {
        return _addr64.getMsb() & 0xff;
    } else if (pos == 4) {
        return (_addr64.getLsb() >> 24) & 0xff;
    } else if (pos == 5) {
        return (_addr64.getLsb() >> 16) & 0xff;
    } else if (pos == 6) {
        return (_addr64.getLsb() >> 8) & 0xff;
    } else if (pos == 7) {
        return _addr64.getLsb() & 0xff;
    } else if (pos == 8) {
        return _option;
    } else {
        return getPayload()[pos - TX_64_API_LENGTH];
    }
}

uint8_t Tx64Request::getFrameDataLength() {
    return TX_64_API_LENGTH + getPayloadLength();
}

XBeeAddress64& Tx64Request::getAddress64() {
    return _addr64;
}

void Tx64Request::setAddress64(XBeeAddress64 &addr64) {
    _addr64 = addr64;
}
#ifndef

AtCommandRequest::AtCommandRequest() : XBeeRequest(AT_COMMAND_REQUEST, DEFAULT_FRAME_ID) {
    _command = NULL;
    clearCommandValue();
}

AtCommandRequest::AtCommandRequest(uint8_t *command, uint8_t *commandValue, uint8_t commandValueLength) : XBeeRequest(AT_COMMAND_REQUEST, DEFAULT_FRAME_ID) {
    _command = command;
    _commandValue = commandValue;
    _commandValueLength = commandValueLength;
}

AtCommandRequest::AtCommandRequest(uint8_t *command) :
    XBeeRequest(AT_COMMAND_REQUEST, DEFAULT_FRAME_ID) {
    _command = command;
    clearCommandValue();
}

uint8_t* AtCommandRequest::getCommand() {
    return _command;
}

uint8_t* AtCommandRequest::getCommandValue() {
    return _commandValue;
}

uint8_t AtCommandRequest::getCommandValueLength() {
    return _commandValueLength;
}

void AtCommandRequest::setCommand(uint8_t* command) {
    _command = command;
}

void AtCommandRequest::setCommandValue(uint8_t* value) {
    _commandValue = value;
}
void AtCommandRequest::setCommandValueLength(uint8_t length) {
    _commandValueLength = length;
}

uint8_t AtCommandRequest::getFrameData(uint8_t pos) {
    if (pos == 0) {
        return _command[0];
    } else if (pos == 1) {
        return _command[1];
    } else {
        return _commandValue[pos - AT_COMMAND_API_LENGTH];
    }
}

void AtCommandRequest::clearCommandValue() {
    _commandValue = NULL;
    _commandValueLength = 0;
}

//void AtCommandRequest::reset() {
//    XBeeRequest::reset();
//}

uint8_t AtCommandRequest::getFrameDataLength() {
    // command is 2 byte + length of value
    return AT_COMMAND_API_LENGTH + _commandValueLength;
}

XBeeAddress64 RemoteAtCommandRequest::broadcastAddress64 = XBeeAddress64(0x0, BROADCAST_ADDRESS);

RemoteAtCommandRequest::RemoteAtCommandRequest() : AtCommandRequest(NULL, NULL, 0) {
    _remoteAddress16 = 0;
    _applyChanges = false;
    setApiId(REMOTE_AT_REQUEST);
}

RemoteAtCommandRequest::RemoteAtCommandRequest(uint16_t remoteAddress16, uint8_t *command, uint8_t *commandValue, uint8_t commandValueLength) :
    AtCommandRequest(command, commandValue, commandValueLength) {
    _remoteAddress64 = broadcastAddress64;
    _remoteAddress16 = remoteAddress16;
    _applyChanges = true;
    setApiId(REMOTE_AT_REQUEST);
RemoteAtCommandRequest::RemoteAtCommandRequest(uint16_t remoteAddress16, uint8_t *command) : AtCommandRequest(command, NULL, 0) {
    _remoteAddress64 = broadcastAddress64;
    _remoteAddress16 = remoteAddress16;
    _applyChanges = false;
    setApiId(REMOTE_AT_REQUEST);
}

RemoteAtCommandRequest::RemoteAtCommandRequest(XBeeAddress64 &remoteAddress64, uint8_t *command, uint8_t *commandValue, uint8_t commandValueLength) :
            AtCommandRequest(command, commandValue, commandValueLength) {
    _remoteAddress64 = remoteAddress64;
            // don't worry.. works for series 1 too!
    _remoteAddress16 = ZB_BROADCAST_ADDRESS;
    _applyChanges = true;
    setApiId(REMOTE_AT_REQUEST);
}

RemoteAtCommandRequest::RemoteAtCommandRequest(XBeeAddress64 &remoteAddress64, uint8_t *command) : AtCommandRequest(command, NULL, 0) {
    _remoteAddress64 = remoteAddress64;
    _remoteAddress16 = ZB_BROADCAST_ADDRESS;
    _applyChanges = false;
    setApiId(REMOTE_AT_REQUEST);
}

uint16_t RemoteAtCommandRequest::getRemoteAddress16() { 
    return _remoteAddress16;
}

void RemoteAtCommandRequest::setRemoteAddress16(uint16_t remoteAddress16) {
    _remoteAddress16 = remoteAddress16;
}

XBeeAddress64& RemoteAtCommandRequest::getRemoteAddress64() {
    return _remoteAddress64;
}

void RemoteAtCommandRequest::setRemoteAddress64(XBeeAddress64 &remoteAddress64) {
    _remoteAddress64 = remoteAddress64;
}

bool RemoteAtCommandRequest::getApplyChanges() {
    return _applyChanges;
}
void RemoteAtCommandRequest::setApplyChanges(bool applyChanges) {
  _applyChanges = applyChanges;
}

uint8_t RemoteAtCommandRequest::getFrameData(uint8_t pos) {
  if (pos == 0) {
    return (_remoteAddress64.getMsb() >> 24) & 0xff;
  } else if (pos == 1) {
    return (_remoteAddress64.getMsb() >> 16) & 0xff;
  } else if (pos == 2) {
    return (_remoteAddress64.getMsb() >> 8) & 0xff;
  } else if (pos == 3) {
    return _remoteAddress64.getMsb() & 0xff;
  } else if (pos == 4) {
    return (_remoteAddress64.getLsb() >> 24) & 0xff;
  } else if (pos == 5) {
    return (_remoteAddress64.getLsb() >> 16) & 0xff;
  } else if (pos == 6) {
    return (_remoteAddress64.getLsb() >> 8) & 0xff;
  } else if (pos == 7) {
    return _remoteAddress64.getLsb() & 0xff;
  } else if (pos == 8) {
    return (_remoteAddress16 >> 8) & 0xff;
  } else if (pos == 9) {
    return _remoteAddress16 & 0xff;
  } else if (pos == 10) {
    return _applyChanges ? 2 : 0;
  } else if (pos == 11) {
    return getCommand()[0];
  } else if (pos == 12) {
    return getCommand()[1];
  } else {
    return getCommandValue()[pos - REMOTE_AT_COMMAND_API_LENGTH];
  }
}

uint8_t RemoteAtCommandRequest::getFrameDataLength() {
  return REMOTE_AT_COMMAND_API_LENGTH + getCommandValueLength();
}

// TODO
//GenericRequest::GenericRequest(uint8_t* frame, uint8_t len, uint8_t apiId):
XBeeRequest(apidl, *(frame), len) {
  // _frame = frame;
  //}
void XBee::send(XBeeRequest &request) {
    // the new new deal

    sendByte(START_BYTE, false);

    // send length
    uint8_t msbLen = ((request.getFrameDataLength() + 2) >> 8) & 0xff;
    uint8_t lsbLen = (request.getFrameDataLength() + 2) & 0xff;

    sendByte(msbLen, true);
    sendByte(lsbLen, true);

    // api id
    sendByte(request.getApiId(), true);
    sendByte(request.getFrameId(), true);

    uint8_t checksum = 0;

    // compute checksum, start at api id
    checksum += request.getApiId();
    checksum += request.getFrameId();

    //std::cout << "frame length is " << static_cast<unsigned int>(request.getFrameDataLength()) << std::endl;

    for (int i = 0; i < request.getFrameDataLength(); i++) {
        // std::cout << "sending byte [" << static_cast<unsigned int>(i) << "] " << std::endl;
        sendByte(request.getFrameData(i), true);
        checksum += request.getFrameData(i);
    }

    // perform 2s complement
    checksum = 0xff - checksum;

    // std::cout << "checksum is " << static_cast<unsigned int>(checksum) << std::endl;

    // send checksum
    sendByte(checksum, true);

    // send packet (Note: prior to Arduino 1.0 this flushed the incoming buffer, which of course
    // was not so great)
    flush();
}

void XBee::sendByte(uint8_t b, bool escape) {
    if (escape && (b == START_BYTE || b == ESCAPE || b == XON || b == XOFF)) {
        // std::cout << "escaping byte [" << toHexString(b) << "] " << std::endl;
    }
}
write(ESCAPE);
write(b ^ 0x20);
} else {
    write(b);
}
}
Xbee.h

/*
 * Xbee.h
 *
 * Created: 4/16/2012 7:22:15 PM
 * Author: Aditya
 */

#ifndef XBEE_H_
#define XBEE_H_

/**
 * Copyright (c) 2009 Andrew Rapp. All rights reserved.
 *
 * This file is part of XBee-Arduino.
 *
 * XBee-Arduino is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * XBee-Arduino is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with XBee-Arduino. If not, see <http://www.gnu.org/licenses/>.
 */

#if defined(ARDUINO) && ARDUINO >= 100
  //include "Arduino.h"
#else
  //include "WProgram.h"
#endif

//include "defines.h"
#include <inttypes.h>

//#define SERIES_1
#define SERIES_2

// set to ATAP value of XBee. AP=2 is recommended
#define ATAP 2
#define START_BYTE 0x7e
#define ESCAPE 0x7d
#define XON 0x11
#define XOFF 0x13

// This value determines the size of the byte array for receiving RX packets
// Most users won't be dealing with packets this large so you can adjust this
// value to reduce memory consumption. But, remember that
// if a RX packet exceeds this size, it cannot be parsed!

// This value is determined by the largest packet size (100 byte payload + 64-bit address + option
// byte and rssi byte) of a series 1 radio
#define MAX_FRAME_DATA_SIZE 110

#define BROADCAST_ADDRESS 0xffff
#define ZB_BROADCAST_ADDRESS 0xfffe

// the non-variable length of the frame data (not including frame id or api id or variable data size
// (e.g. payload, at command set value)
#define ZB_TX_API_LENGTH 12
#define TX_16_API_LENGTH 3
#define TX_64_API_LENGTH 9
#define AT_COMMAND_API_LENGTH 2
#define REMOTE_AT_COMMAND_API_LENGTH 13
// start/length(2)/api/frameid/checksum bytes
#define PACKET_OVERHEAD_LENGTH 6
// api is always the third byte in packet
#define API_ID_INDEX 3

// frame position of rssi byte
#define RX_16_RSSI_OFFSET 2
#define RX_64_RSSI_OFFSET 8

#define DEFAULT_FRAME_ID 1
#define NO_RESPONSE_FRAME_ID 0

// TODO put in tx16 class
#define ACK_OPTION 0
#define DISABLE_ACK_OPTION 1
#define BROADCAST_OPTION 4

// RX options
#define ZB_PACKET_ACKNOWLEDGED 0x01
#define ZB_BROADCAST_PACKET 0x02

// not everything is implemented!
/**
* Api Id constants
  */
#define TX_64_REQUEST 0x0
#define TX_16_REQUEST 0x1
#define AT_COMMAND_REQUEST 0x08
#define AT_COMMAND_QUEUE_REQUEST 0x09
#define REMOTE_AT_REQUEST 0x17
#define ZB_TX_REQUEST 0x10
#define ZB_EXPLICIT_TX_REQUEST 0x11
#define RX_64_RESPONSE 0x80
#define RX_16_RESPONSE 0x81
#define RX_64_IO_RESPONSE 0x82
#define RX_16_IO_RESPONSE 0x83
#define AT_RESPONSE 0x88
#define TX_STATUS_RESPONSE 0x89
#define MODEM_STATUS_RESPONSE 0x8a
#define ZB_RX_RESPONSE 0x90
#define ZB_EXPLICIT_RX_RESPONSE 0x91
#define ZB_TX_STATUS_RESPONSE 0x9b
#define ZB_IO_SAMPLE_RESPONSE 0x92
#define ZB_IO_NODE_IDENTIFIER_RESPONSE 0x95
#define AT_COMMAND_RESPONSE 0x88
#define REMOTE_AT_COMMAND_RESPONSE 0x97

/*
  * TX STATUS constants
  */
#define SUCCESS 0x0
#define CCA_FAILURE 0x2
#define INVALID_DESTINATION_ENDPOINT_SUCCESS 0x15
#define NETWORK_ACK_FAILURE 0x21
#define NOT_JOINED_TO_NETWORK 0x22
#define SELF_ADDRESSED 0x23
#define ADDRESS_NOT_FOUND 0x24
#define ROUTE_NOT_FOUND 0x25
#define PAYLOAD_TOO_LARGE 0x74

// modem status
#define HARDWARE_RESET 0
#define WATCHDOG_TIMER_RESET 1
#define ASSOCIATED 2
#define DISASSOCIATED 3
#define SYNCHRONIZATION_LOST 4
#define COORDINATOR_REALIGNMENT 5
#define COORDINATOR_STARTED 6
#define ZB_BROADCAST_RADIUS_MAX_HOPS 0
#define ZB_TX_UNICAST 0
#define ZB_TX_BROADCAST 8

#define AT_OK 0
#define AT_ERROR 1
#define AT_INVALID_COMMAND 2
#define AT_INVALID_PARAMETER 3
#define AT_NO_RESPONSE 4

#define NO_ERROR 0
#define CHECKSUM_FAILURE 1
#define PACKET_EXCEEDS_BYTE ARRAY_LENGTH 2
#define UNEXPECTED_START_BYTE 3

/**
 * The super class of all XBee responses (RX packets)
 * Users should never attempt to create an instance of this class; instead
 * create an instance of a subclass
 * It is recommend to reuse subclasses to conserve memory
 */
class XBeeResponse {
public:
    //static const int MODEM_STATUS = 0x8a;
    /**
     * Default constructor
     */
    XBeeResponse();
    /**
     * Returns Api Id of the response
     */
    uint8_t getApiId();
    void setApiId(uint8_t apiId);
    /**
     * Returns the MSB length of the packet
     */
    uint8_t getMsbLength();
    void setMsbLength(uint8_t msbLength);
    /**
     * Returns the LSB length of the packet
     */
    uint8_t getLsbLength();
    void setLsbLength(uint8_t lsbLength);
    /**
     * Returns the packet checksum
     */
    uint8_t getChecksum();
    void setChecksum(uint8_t checksum);
/**
  * Returns the length of the frame data: all bytes after the api id, and prior to the checksum
  * Note up to release 0.1.2, this was incorrectly including the checksum in the length.
  */
uint8_t getFrameDataLength();
void setFrameData(uint8_t* frameDataPtr);

/**
  * Returns the buffer that contains the response.
  * Starts with byte that follows API ID and includes all bytes prior to the checksum
  * Length is specified by getFrameDataLength()
  * Note: Unlike Digi's definition of the frame data, this does not start with the API ID..
  * The reason for this is all responses include an API ID, whereas my frame data
  * includes only the API specific data.
  */
uint8_t* getFrameData();

void setFrameLength(uint8_t frameLength);
// to support future 65535 byte packets I guess
/**
  * Returns the length of the packet
  */
uint16_t getPacketLength();

/**
  * Resets the response to default values
  */
void reset();

/**
  * Initializes the response
  */
void init();

#ifdef SERIES_2
/**
  * Call with instance of ZBTxStatusResponse class only if getApiId() ==
  * ZB_TX_STATUS_RESPONSE
  * to populate response
  */
void getZBTxStatusResponse(XBeeResponse &response);

/**
  * Call with instance of ZBRxResponse class only if getApiId() == ZB_RX_RESPONSE
  * to populate response
  */
void getZBRxResponse(XBeeResponse &response);

/**
  * Call with instance of ZBRxIoSampleResponse class only if getApiId() ==
  * ZB_IO_SAMPLE_RESPONSE
  * to populate response
  */
void getZBRxIoSampleResponse(XBeeResponse &response);

#endif
### Series 1

```c
#define SERIES_1

/** *
 * Call with instance of TxStatusResponse only if getApiId() == TX_STATUS_RESPONSE
 */
void getTxStatusResponse(XBeeResponse &response);

/** *
 * Call with instance of Rx16Response only if getApiId() == RX_16_RESPONSE
 */
void getRx16Response(XBeeResponse &response);

/** *
 * Call with instance of Rx64Response only if getApiId() == RX_64_RESPONSE
 */
void getRx64Response(XBeeResponse &response);

/** *
 * Call with instance of Rx16IoSampleResponse only if getApiId() == RX_16_IO_RESPONSE
 */
void getRx16IoSampleResponse(XBeeResponse &response);

/** *
 * Call with instance of Rx64IoSampleResponse only if getApiId() == RX_64_IO_RESPONSE
 */
void getRx64IoSampleResponse(XBeeResponse &response);

/** *
 * Call with instance of AtCommandResponse only if getApiId() == AT_COMMAND_RESPONSE
 */
void getAtCommandResponse(XBeeResponse &responses);

/** *
 * Call with instance of RemoteAtCommandResponse only if getApiId() == REMOTE_AT_COMMAND_RESPONSE
 */
void getRemoteAtCommandResponse(XBeeResponse &response);

/** *
 * Call with instance of ModemStatusResponse only if getApiId() == MODEM_STATUS_RESPONSE
 */
void getModemStatusResponse(XBeeResponse &response);

/** *
 * Returns true if the response has been successfully parsed and is complete and ready for use
 */
bool isAvailable();

void setAvailable(bool complete);
```
* Returns true if the response contains errors
*/
bool isError();
/**
 * Returns an error code, or zero, if successful.
 * Error codes include: CHECKSUM_FAILURE,
 * PACKET_EXCEEDS_BYTE_ARRAY_LENGTH, UNEXPECTED_START_BYTE
 */
uint8_t getErrorCode();
void setErrorCode(uint8_t errorCode);

protected:
// pointer to frameData
uint8_t* _frameDataPtr;

private:
void setCommon(XBeeResponse &target);
uint8_t _apiId;
uint8_t _msbLength;
uint8_t _lsbLength;
uint8_t _checksum;
uint8_t _frameLength;
bool _complete;
uint8_t _errorCode;

};

class XBeeAddress {
public:
    XBeeAddress();
};

/**
 * Represents a 64-bit XBee Address
 */
class XBeeAddress64 : public XBeeAddress {
public:
    XBeeAddress64(uint32_t msb, uint32_t lsb);
    XBeeAddress64();
    uint32_t getMsb();
    uint32_t getLsb();
    void setMsb(uint32_t msb);
    void setLsb(uint32_t lsb);

private:
    uint32_t _msb;
    uint32_t _lsb;

};

//class XBeeAddress16 : public XBeeAddress {
//public:
//    XBeeAddress16(uint16_t addr);

};
/**
 * This class is extended by all Responses that include a frame id
 */
class FrameIdResponse : public XBeeResponse {
public:
  FrameIdResponse();
  uint8_t getFrameId();
private:
  uint8_t _frameId;
};

/**
 * Common functionality for both Series 1 and 2 data RX data packets
 */
class RxDataResponse : public XBeeResponse {
public:
  RxDataResponse();
  /**
   * Returns the specified index of the payload. The index may be 0 to getDataLength() - 1
   * This method is deprecated; use uint8_t* getData()
   */
  uint8_t getData(int index);
  /**
   * Returns the payload array. This may be accessed from index 0 to getDataLength() - 1
   */
  uint8_t* getData();
  /**
   * Returns the length of the payload
   */
  virtual uint8_t getDataLength() = 0;
  /**
   * Returns the position in the frame data where the data begins
   */
  virtual uint8_t getDataOffset() = 0;
};

// getResponse to return the proper subclass:
// we maintain a pointer to each type of response, when a response is parsed, it is allocated only if
// can we allocate an object in a function?
#ifdef SERIES_2
/**
 * Represents a Series 2 TX status packet
 */
class ZBTxStatusResponse : public FrameIdResponse {
    public:
        ZBTxStatusResponse();
        uint16_t getRemoteAddress();
        uint8_t getTxRetryCount();
        uint8_t getDeliveryStatus();
        uint8_t getDiscoveryStatus();
        bool isSuccess();
};
/**
 * Represents a Series 2 RX packet
 */
class ZBRxResponse : public RxDataResponse {
    public:
        ZBRxResponse();
        XBeeAddress64& getRemoteAddress64();
        uint16_t getRemoteAddress16();
        uint8_t getOption();
        uint8_t getDataLength();
        // frame position where data starts
        uint8_t getDataOffset();
    private:
        XBeeAddress64 _remoteAddress64;
};
/**
 * Represents a Series 2 RX I/O Sample packet
 */
class ZBRxIoSampleResponse : public ZBRxResponse {
    public:
        ZBRxIoSampleResponse();
        bool containsAnalog();
        bool containsDigital();
        /**
         * Returns true if the pin is enabled
         */
        bool isAnalogEnabled(uint8_t pin);
        /**
         * Returns true if the pin is enabled
         */
        bool isDigitalEnabled(uint8_t pin);
        /**
         * Returns the 10-bit analog reading of the specified pin.
         */
uint16_t getAnalog(uint8_t pin);

bool isDigitalOn(uint8_t pin);

uint8_t getDigitalMaskMsb();
uint8_t getDigitalMaskLsb();
uint8_t getAnalogMask();

};

#ifdef SERIES_1
/**
 * Represents a Series 1 TX Status packet
 */
class TxStatusResponse : public FrameIdResponse {
    public:
        TxStatusResponse();
        uint8_t getStatus();
        bool isSuccess();
};

/**
 * Represents a Series 1 RX packet
 */
class RxResponse : public RxDataResponse {
    public:
        RxResponse();
        uint8_t getRssi();
        uint8_t getOption();
        bool isAddressBroadcast();
        bool isPanBroadcast();
        uint8_t getDataLength();
        uint8_t getDataOffset();
        virtual uint8_t getRssiOffset() = 0;
};

/**
 * Represents a Series 1 16-bit address RX packet
 */
class Rx16Response : public RxResponse {
    public:
        Rx16Response();

    };

#endif
uint8_t getRssiOffset();
uint16_t getRemoteAddress16();
protected:
    uint16_t _remoteAddress;
};

/**
 * Represents a Series 1 64-bit address RX packet
 */
class Rx64Response : public RxResponse {
public:
    Rx64Response();
    uint8_t getRssiOffset();
    XBeeAddress64& getRemoteAddress64();
private:
    XBeeAddress64 _remoteAddress;
};

/**
 * Represents a Series 1 RX I/O Sample packet
 */
class RxIoSampleBaseResponse : public RxResponse {
public:
    RxIoSampleBaseResponse();
    /**
     * Returns the number of samples in this packet
     */
    uint8_t getSampleSize();
    bool containsAnalog();
    bool containsDigital();
    /**
     * Returns true if the specified analog pin is enabled
     */
    bool isAnalogEnabled(uint8_t pin);
    /**
     * Returns true if the specified digital pin is enabled
     */
    bool isDigitalEnabled(uint8_t pin);
    /**
     * Returns the 10-bit analog reading of the specified pin.
     * Valid pins include ADC:0-5. Sample index starts at 0
     */
    uint16_t getAnalog(uint8_t pin, uint8_t sample);
    /**
     * Returns true if the specified pin is high/on.
     * Valid pins include DIO:0-8. Sample index starts at 0
     */
    bool isDigitalOn(uint8_t pin, uint8_t sample);
uint8_t getSampleOffset();
private:
};

class Rx16IoSampleResponse : public RxIoSampleBaseResponse {
public:
    Rx16IoSampleResponse();
    uint16_t getRemoteAddress16();
    uint8_t getRssiOffset();
};

class Rx64IoSampleResponse : public RxIoSampleBaseResponse {
public:
    Rx64IoSampleResponse();
    XBeeAddress64 getRemoteAddress64();
    uint8_t getRssiOffset();
private:
    XBeeAddress64 _remoteAddress;
};

#endif

/**
 * Represents a Modem Status RX packet
 */
class ModemStatusResponse : public XBeeResponse {
public:
    ModemStatusResponse();
    uint8_t getStatus();
};

/**
 * Represents an AT Command RX packet
 */
class AtCommandResponse : public FrameIdResponse {
public:
    AtCommandResponse();
    /**
     * Returns an array containing the two character command
     */
    uint8_t* getCommand();
    /**
     * Returns the command status code.
     * Zero represents a successful command
     */
    uint8_t getStatus();
    /**
* Returns an array containing the command value.
* This is only applicable to query commands.
 */
uint8_t* getValue();
/**
* Returns the length of the command value array.
*/
uint8_t getValueLength();
/**
* Returns true if status equals AT_OK
*/
bool isOk();
};

/**
* Represents a Remote AT Command RX packet
*/
class RemoteAtCommandResponse : public AtCommandResponse {
public:
    RemoteAtCommandResponse();
/**
* Returns an array containing the two character command
*/
uint8_t* getCommand();
/**
* Returns the command status code.
* Zero represents a successful command
*/
uint8_t getStatus();
/**
* Returns an array containing the command value.
* This is only applicable to query commands.
*/
uint8_t* getValue();
/**
* Returns the length of the command value array.
*/
uint8_t getValueLength();
/**
* Returns the 16-bit address of the remote radio
*/
uint16_t getRemoteAddress16();
/**
* Returns the 64-bit address of the remote radio
*/
XBeeAddress64& getRemoteAddress64();
/**
* Returns true if command was successful
/**
 * Super class of all XBee requests (TX packets)
 * Users should never create an instance of this class; instead use a subclass of this class
 * It is recommended to reuse Subclasses of the class to conserve memory
 * <p/>
 * This class allocates a buffer to
 */

class XBeeRequest {

public:

/**
 * Constructor
 * TODO make protected
 */
XBeeRequest(uint8_t apiId, uint8_t frameId);

/**
 * Sets the frame id. Must be between 1 and 255 inclusive to get a TX status response.
 */
void setFrameId(uint8_t frameId);

/**
 * Returns the frame id
 */
uint8_t getFrameId();

/**
 * Returns the API id
 */
uint8_t getApiId();

// setting = 0 makes this a pure virtual function, meaning the subclass must implement, like abstract in java

/**
 * Starting after the frame id (pos = 0) and up to but not including the checksum
 * Note: Unlike Digi’s definition of the frame data, this does not start with the API ID.
 * The reason for this is the API ID and Frame ID are common to all requests, whereas my definition of
 * frame data is only the API specific data.
 */
virtual uint8_t getFrameData(uint8_t pos) = 0;

/**
 * Returns the size of the api frame (not including frame id or api id or checksum).
 */
virtual uint8_t getFrameDataLength() = 0;

// void reset();
protected:
    void setApiId(uint8_t apiId);
private:
    uint8_t _apiId;
    uint8_t _frameId;
};

// TODO add reset/clear method since responses are often reused
/**
 * Primary interface for communicating with an XBee Radio.
 * This class provides methods for sending and receiving packets with an XBee radio via the
 * serial port.
 * The XBee radio must be configured in API (packet) mode (AP=2)
 * in order to use this software.
 * <p/>
 * Since this code is designed to run on a microcontroller, with only one thread, you are
 * responsible for reading the
 * data off the serial buffer in a timely manner. This involves a call to a variant of readPacket(...).
 * If your serial port is receiving data faster than you are reading, you can expect to lose packets.
 * Arduino only has a 128 byte serial buffer so it can easily overflow if two or more packets
 * arrive
 * without a call to readPacket(...)
 * <p/>
 * In order to conserve resources, this class only supports storing one response packet in memory
 * at a time.
 * This means that you must fully consume the packet prior to calling readPacket(...), because
 * calling
 * readPacket(...) overwrites the previous response.
 * <p/>
 * This class creates an array of size MAX_FRAME_DATA_SIZE for storing the response
 * packet. You may want
 * to adjust this value to conserve memory.
 * *
 * \author Andrew Rapp
 */
class XBee {
public:
    XBee();
    /**
     * Reads all available serial bytes until a packet is parsed, an error occurs, or the buffer is
     * empty.
     * You may call <i>xbee</i>.getResponse().isAvailable() after calling this method to
     * determine if
     * a packet is ready, or <i>xbee</i>.getResponse().isError() to determine if
     * an error occurred.
     * <p/>
     * This method should always return quickly since it does not wait for serial data to arrive.
     * You will want to use this method if you are doing other timely stuff in your loop, where
* a delay would cause problems.
* NOTE: calling this method resets the current response, so make sure you first consume
the

* current response
*/
void readPacket();
/**
 * Waits a maximum of <i>timeout</i> milliseconds for a response packet before timing
out; returns true if packet is read.
 * Returns false if timeout or error occurs.
 */
bool readPacket(int timeout);
/**
 * Reads until a packet is received or an error occurs.
 * Caution: use this carefully since if you don't get a response, your Arduino code will hang
on this
 * call forever!! often it's better to use a timeout: readPacket(int)
 */
void readPacketUntilAvailable();
/**
 * Starts the serial connection at the supplied baud rate
 */
void begin(long baud);
void getResponse(XBeeResponse &response);
/**
 * Returns a reference to the current response
 * Note: once readPacket is called again this response will be overwritten!
 */
XBeeResponse& getResponse();
/**
 * Sends a XBeeRequest (TX packet) out the serial port
 */
void send(XBeeRequest &request);
//uint8_t sendAndWaitForResponse(XBeeRequest &request, int timeout);
/**
 * Returns a sequential frame id between 1 and 255
 */
uint8_t getNextFrameId();
/**
 * Specify the serial port. Only relevant for Arduinos that support multiple serial ports (e.g.
Mega)
 * /
 //void setSerial(HardwareSerial &serial);

private:
bool available();
uint8_t read();
void flush();
void write(uint8_t val);
void sendByte(uint8_t b, bool escape);
void resetResponse();
XBeeResponse _response;
bool _escape;
// current packet position for response. just a state variable for packet parsing and has no relevance for the response otherwise
uint8_t _pos;
// last byte read
uint8_t b;
uint8_t _checksumTotal;
uint8_t _nextFrameId;
// buffer for incoming RX packets. holds only the api specific frame data, starting after the api id byte and prior to checksum
uint8_t _responseFrameData[MAX_FRAME_DATA_SIZE];
//HardwareSerial* _serial;

/**
 * All TX packets that support payloads extend this class
 */
class PayloadRequest : public XBeeRequest {
public:
  PayloadRequest(uint8_t apiId, uint8_t frameId, uint8_t *payload, uint8_t payloadLength);
  /**
   * Returns the payload of the packet, if not null
   */
  uint8_t* getPayload();
  /**
   * Sets the payload array
   */
  void setPayload(uint8_t* payloadPtr);
  /**
   * Returns the length of the payload array, as specified by the user.
   */
  uint8_t getPayloadLength();
  /**
   * Sets the length of the payload to include in the request. For example if the payload array
   * is 50 bytes and you only want the first 10 to be included in the packet, set the length to 10.
   * Length must be <= to the array length.
   */
  void setPayloadLength(uint8_t payloadLength);
private:
  uint8_t* _payloadPtr;
  uint8_t _payloadLength;
};

#ifdef SERIES_1

```cpp

```
/**
   * Represents a Series 1 TX packet that corresponds to Api Id: TX_16_REQUEST
   * <p/>
   * Be careful not to send a data array larger than the max packet size of your radio.
   * This class does not perform any validation of packet size and there will be no indication
   * if the packet is too large, other than you will not get a TX Status response.
   * The datasheet says 100 bytes is the maximum, although that could change in future firmware.
   */
class Tx16Request : public PayloadRequest {
public:
    Tx16Request(uint16_t addr16, uint8_t option, uint8_t *payload, uint8_t payloadLength, uint8_t frameId);
    /**
     * Creates a Unicast Tx16Request with the ACK option and DEFAULT_FRAME_ID
     */
    Tx16Request(uint16_t addr16, uint8_t *payload, uint8_t payloadLength);
    /**
     * Creates a default instance of this class. At a minimum you must specify
     * a payload, payload length and a destination address before sending this request.
     */
    Tx16Request();
    uint16_t getAddress16();
    void setAddress16(uint16_t addr16);
    uint8_t getOption();
    void setOption(uint8_t option);
    uint8_t getFrameData(uint8_t pos);
    uint8_t getFrameDataLength();
protected:
private:
    uint16_t _addr16;
    uint8_t _option;
};

/**
 * Represents a Series 1 TX packet that corresponds to Api Id: TX_64_REQUEST
 * <p/>
 * Be careful not to send a data array larger than the max packet size of your radio.
 * This class does not perform any validation of packet size and there will be no indication
 * if the packet is too large, other than you will not get a TX Status response.
 * The datasheet says 100 bytes is the maximum, although that could change in future firmware.
 */
class Tx64Request : public PayloadRequest {
public:
    Tx64Request(XBeeAddress64 &addr64, uint8_t option, uint8_t *payload, uint8_t payloadLength, uint8_t frameId);
    /**
     * Creates a unicast Tx64Request with the ACK option and DEFAULT_FRAME_ID
     */
    ...
Tx64Request(XBeeAddress64 &addr64, uint8_t *payload, uint8_t payloadLength);

* Creates a default instance of this class. At a minimum you must specify
* a payload, payload length and a destination address before sending this request.
*/

Tx64Request();
XBeeAddress64& getAddress64();
void setAddress64(XBeeAddress64& addr64);
// TODO move option to superclass
uint8_t getOption();
void setOption(uint8_t option);
uint8_t getFrameData(uint8_t pos);
uint8_t getFrameDataLength();

private:
    XBeeAddress64 _addr64;
    uint8_t _option;

};

#endif
#ifdef SERIES_2

/**
 * Represents a Series 2 TX packet that corresponds to Api Id: ZB_TX_REQUEST
 * 
 * Be careful not to send a data array larger than the max packet size of your radio.
 * This class does not perform any validation of packet size and there will be no indication
 * if the packet is too large, other than you will not get a TX Status response.
 * The datasheet says 72 bytes is the maximum for ZNet firmware and ZB Pro firmware provides
 * the ATNP command to get the max supported payload size. This command is useful since the
 * maximum payload size varies according to certain settings, such as encryption.
 * ZB Pro firmware provides a PAYLOAD_TOO_LARGE that is returned if payload size
 * exceeds the maximum.
 */

class ZBTxRequest : public PayloadRequest {

public:
    /**
     * Creates a unicast ZBTxRequest with the ACK option and DEFAULT_FRAME_ID
     */
    ZBTxRequest(XBeeAddress64 &addr64, uint8_t *payload, uint8_t payloadLength);
    ZBTxRequest(XBeeAddress64 &addr64, uint16_t addr16, uint8_t broadcastRadius, uint8_t 
option, uint8_t *payload, uint8_t payloadLength, uint8_t frameId);
    /**
     * Creates a default instance of this class. At a minimum you must specify
     * a payload, payload length and a destination address before sending this request.
     */

};

#endif
ZBTxRequest();
XBeeAddress64& getAddress64();
uint16_t getAddress16();
uint8_t getBroadcastRadius();
uint8_tgetOption();
void setAddress64(XBeeAddress64& addr64);
void setAddress16(uint16_t addr16);
void setBroadcastRadius(uint8_t broadcastRadius);
void setOption(uint8_t option);

protected:
    // declare virtual functions
    uint8_t getFrameData(uint8_t pos);
    uint8_t getFrameDataLength();

private:
    XBeeAddress64 _addr64;
    uint16_t _addr16;
    uint8_t _broadcastRadius;
    uint8_t _option;
};

#endif

/**
 * Represents an AT Command TX packet
 * The command is used to configure the serially connected XBee radio
 */
class AtCommandRequest : public XBeeRequest {
public:
    AtCommandRequest();
    AtCommandRequest(uint8_t *command);
    AtCommandRequest(uint8_t *command, uint8_t *commandValue, uint8_t commandValueLength);
    uint8_t getFrameData(uint8_t pos);
    uint8_t getFrameDataLength();
    uint8_t* getCommand();
    void setCommand(uint8_t* command);
    uint8_t* getCommandValue();
    void setCommandValue(uint8_t* command);
    uint8_t getCommandValueLength();
    void setCommandValueLength(uint8_t length);
    /**
     * Clears the optional commandValue and commandValueLength so that a query may be sent
     */
    void clearCommandValue();
    //void reset();

private:
    uint8_t _command;
class RemoteAtCommandRequest : public AtCommandRequest {
public:
    RemoteAtCommandRequest();
    /**
     * Creates a RemoteAtCommandRequest with 16-bit address to set a command.
     * 64-bit address defaults to broadcast and applyChanges is true.
     */
    RemoteAtCommandRequest(uint16_t remoteAddress16, uint8_t *command, uint8_t *commandValue, uint8_t commandValueLength);
    /**
     * Creates a RemoteAtCommandRequest with 16-bit address to query a command.
     * 64-bit address defaults to broadcast and applyChanges is true.
     */
    RemoteAtCommandRequest(uint16_t remoteAddress16, uint8_t *command);
    /**
     * Creates a RemoteAtCommandRequest with 64-bit address to set a command.
     * 16-bit address defaults to broadcast and applyChanges is true.
     */
    RemoteAtCommandRequest(XBeeAddress64 &remoteAddress64, uint8_t *command, uint8_t *commandValue, uint8_t commandValueLength);
    /**
     * Creates a RemoteAtCommandRequest with 16-bit address to query a command.
     * 16-bit address defaults to broadcast and applyChanges is true.
     */
    RemoteAtCommandRequest(XBeeAddress64 &remoteAddress64, uint8_t *command);
    uint16_t getRemoteAddress16();
    void setRemoteAddress16(uint16_t remoteAddress16);
    XBeeAddress64 & getRemoteAddress64();
    void setRemoteAddress64(XBeeAddress64 &remoteAddress64);
    bool getApplyChanges();
    void setApplyChanges(bool applyChanges);
    uint8_t getFrameData(uint8_t pos);
    uint8_t getFrameDataLength();
    static XBeeAddress64 broadcastAddress64;
    // static uint16_t broadcast16Address;
private:
    XBeeAddress64 _remoteAddress64;
    uint16_t _remoteAddress16;
    bool _applyChanges;
};
#endif /* XBEE_H */
3. Code for mac based server for database:

**Main.cpp**

```cpp
#include "ofMain.h"
#include "testApp.h"
#include "ofAppGlutWindow.h"

//===============================================
//===
testApp app;

int main( ){
    ofAppGlutWindow window;
    ofSetupOpenGL(&window, 100,100, OF_WINDOW); // <-------- setup the GL context
    // this kicks off the running of my app
    // can be OF_WINDOW or OF_FULLSCREEN
    // pass in width and height too:
    ofRunApp( &app);
}
```
testApp.cpp

#include "testApp.h"
#include "Xbee.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

XBee xbee = XBee();
XBeeresponse response = XBeeResponse();
// create reusable response objects for responses we expect to handle
ZBRxResponse rx = ZBRxResponse();
ModemStatusResponse msr = ModemStatusResponse();

typedef union DataAccess {
    DataPacket NewData;
    char buffer[sizeof(DataPacket)];
} DataAccess;

typedef union Addr {
    int Adr[2];
    double AD;
} Addr;

Addr newAddr;

DataAccess newPacket;
DataPacket *testData;

//----------------------------------------------------------------------
void testApp::setup(){
    ofSetVerticalSync(true);
    bSendSerialMessage = false;
    ofBackground(255,255,255);
    ofSetLogLevel(OF_LOG_NOTICE);

    //-------------------------------
    font.loadFont("DIN.otf", 64);

    serial.listDevices();
    vector <ofSerialDeviceInfo> deviceList = serial.getDeviceList();
serial.setup("/dev/tty.usbserial-A900UEJ6", 57600); //open the first device

// serial.setup("COM4");                     // windows example
// serial.setup("/dev/tty.usbserial-A4001JEC",9600); // mac osx example
// serial.setup("/dev/ttyUSB0", 9600);          //linux example

nTimesRead = 0;
nBytesRead = 0;
readTime = 0;
memset(bytesReadString, 0, 4);
ofSetLogLevel(OF_LOG_VERBOSE);
xbee.begin(57600);
ofLogVerbose() << "Started Serial Program\n";
//insertDB();
}

//-----------------------------------------------------------------------------------
void testApp::update(){
    char temp;
    //connect to the database

    xbee.readPacket(100);

    if (xbee.getResponse().isAvailable()) {
        // got something
        cout << "got something" << endl;
        if (xbee.getResponse().getApId() == ZB_RX_RESPONSE) {
            // got a zb rx packet
            cout << "zb pack" << endl;
            // now fill our zb rx class
            xbee.getResponse().getZBRxResponse(rx);

            if (rx.getOption() == ZB_PACKET_ACKNOWLEDGED) {
                // the sender got an ACK
                ofLogVerbose("ACK done proerly");
            } else {
                // we got it (obviously) but sender didn't get an ACK
                ofLogVerbose("ACK error");
            }
        }
    }
}
// set dataLed PWM to value of the first byte in the data

for(int i = 0; i < sizeof(DataPacket); i++){
    newPacket.buffer[i] = rx.getData(i);
}

XBeeAddress64 RxAddr;

RxAddr = rx.getRemoteAddress64();
newAddr.Adr[0] = RxAddr.getLsb();
newAddr.Adr[1] = RxAddr.getMsb();
insertDB();

} else if (xbee.getResponse().getApiId() == MODEM_STATUS_RESPONSE) {

}

}

void testApp::insertDB (void){

    vector<ofxMySQLField> vals;
    vector<string> readVals;

    char r1[16];
    char r2[30];
    char r3[30];
    char r4[80];
    char r5[30];
    char r6[30];
    char r7[30];
    char r8[30];

    time_t sec;
    struct tm * timeinfo;
    time(&sec);
    timeinfo = localtime ( &sec);

    for(int i = 0; i < newPacket.NewData.NumberOfLocations; i++){
sprintf(r2, " Location = %d ", newPacket.NewData.Location[i]);

readVals.clear();
db.ofxMySQL::getStrings(readVals, " lot1 ", " * ", r2);
int temp;

ofLogVerbose() << "read values = " << (int)readVals.size() << endl;

cout << "n Address full " << hex << (int)newAddr.AD << endl;
cout << "n Address h " << hex << (int)newAddr.Adr[0] << endl;
cout << "n Address l " << hex << (int)newAddr.Adr[1] << endl;

if(readVals.size() > 0){
    sprintf(r1, "%d", newPacket.NewData.Location[i]);
    sprintf(r2, " Location = %d ", newPacket.NewData.Location[i]);
    sprintf(r3, "%d", newPacket.NewData.LocationStatus[i]);
    strftime(r4,80,"%y-%m-%d %H:%M:%S",timeinfo);
    // temp =(int) (newAddr.AD >> 16);
    sprintf(r5, "x%x%x",newAddr.Adr[1],newAddr.Adr[0]);
    sprintf(r6, "%d", newPacket.NewData.BatteryStatus);
    sprintf(r7, "%d", newPacket.NewData.AuxBattStatus);
    cout <<"n r5 is " << r5 <<endl;

    vals.clear();
    //insert value for click event for col2
    ofxMySQLField f2(" Address ", r5);
    vals.push_back(f2);
    //insert value for click event for col2
    ofxMySQLField f3(" LocationStatus ", r3);
    vals.push_back(f3);
    //insert value for click event for col2
    ofxMySQLField f4(" BatteryStatus ", r6);
    vals.push_back(f4);
    ofxMySQLField f5(" AuxBattery ", r7);
    vals.push_back(f5);
    //insert value for click event for col2
    ofxMySQLField f6(" TimeIN ", r4);
    vals.push_back(f6);
//insert value for click event for col2
ofxMySQLField f7(" TimeOut ", r4);
vals.push_back(f7);

//insert value for click event for col2
ofxMySQLField f8(" LastUpdate ", r4);
vals.push_back(f8);

//insert value for click event for col2
ofxMySQLField f9(" Reserved ", " 0 ");
vals.push_back(f9);

//insert value for click event for col2
ofxMySQLField f10(" SkippedUpdate ", " 0 ");
vals.push_back(f10);

db.ofxMySQL::update(" lot1 ", vals, r2);
}

else{
    //do insert of the data
    sprintf(r1, "%d", newPacket.NewData.Location[i]);
sprintf(r2, " Location = %d ", newPacket.NewData.Location[i]);
sprintf(r3, "%d", newPacket.NewData.LocationStatus[i]);
strftime (r4, 80, "%y-%m-%d %H:%M:%S",timeinfo);
sprintf(r5, "x%\x%x",newAddr.Adr[1],newAddr.Adr[0]);
sprintf(r6, "%d", newPacket.NewData.BatteryStatus);
sprintf(r7, "%d", newPacket.NewData.AuxBattStatus);
vals.clear();

//insert value for click event for col1
ofxMySQLField f1("Location", r1);
vals.push_back(f1);

//insert value for click event for col2
ofxMySQLField f2("Address", r5);
vals.push_back(f2);

//insert value for click event for col2
ofxMySQLField f3("LocationStatus", r3);
vals.push_back(f3);

//insert value for click event for col2
ofxMySQLField f4("BatteryStatus", r6);
vals.push_back(f4);
//insert value for click event for col2
ofxMySQLField f5("AuxBattery", r7);
vals.push_back(f5);

//insert value for click event for col2
ofxMySQLField f6("TimeIN", r4);
vals.push_back(f6);

//insert value for click event for col2
ofxMySQLField f7("TimeOut", r4);
vals.push_back(f7);

//insert value for click event for col2
ofxMySQLField f8("LastUpdate", r4);
vals.push_back(f8);

//insert value for click event for col2
ofxMySQLField f9("Reserved", "'0'"蟻);
vals.push_back(f9);

//insert value for click event for col2
ofxMySQLField f10("SkippedUpdate", "'0'"蟻);
vals.push_back(f10);

db.ofxMySQL::insert("lot1", vals);

} } } } }

vector<string>::iterator fieldit;
vector<vector<string>> selectValues;
vector<vector<string>> ::iterator selectValuesIt;
vector<string> feildWanted;
string v1 = "*";
string v2 = "BatteryStatus";

feildWanted.push_back(v1);
// feildWanted.push_back(v2);

db.ofxMySQL::getStrings(selectValues, " lot1 ", feildWanted, " Location = 1000 ");

// selectValuesIt = selectValues.begin();
for(int i =0; i < selectValues.size(); ++i){
    for(int j = 0; j < selectValues[i].size(); ++j){
        cout << selectValues[i][j] << "t";
    }
cout << "\n";}
}

void testApp::draw(){
}

void testApp::keyPressed (int key){
}

void testApp::keyReleased(int key){
}

void testApp::mouseMoved(int x, int y){
}

void testApp::mouseDragged(int x, int y, int button){
}

void testApp::mousePressed(int x, int y, int button){
    bSendSerialMessage = true;
}

void testApp::mouseReleased(int x, int y, int button){
}

void testApp::windowResized(int w, int h){
}

void testApp::gotMessage(ofMessage msg){

}
void testApp::dragEvent(ofDragInfo dragInfo){
}

//--------------------------------------------------------------
#pragma once

#include "ofMain.h"
#include "ofxMySQL.h"

class testApp : public ofBaseApp{

public:
  void setup();
  void update();
  void draw();
  void keyPressed (int key);
  void keyReleased(int key);
  void mouseMoved(int x, int y);
  void mouseDragged(int x, int y, int button);
  void mousePressed(int x, int y, int button);
  void mouseReleased(int x, int y, int button);
  void windowResized(int w, int h);
  void dragEvent(ofDragInfo dragInfo);
  void gotMessage(ofMessage msg);
  void insertDB (void);

  ofTrueTypeFont font;

  bool bSendSerialMessage; // a flag for sending serial
  char bytesRead[3]; // data from serial, we will be trying to read 3 bytes
  char bytesReadString[4]; // a string needs a null terminator, so we need 3 + 1 bytes
  int nBytesRead; // how much did we read?
  int nTimesRead; // how many times did we read?
  float readTime; // when did we last read?

  ofxMySQL db;
  ofSerial serial;
};

#pragma pack(1)
typedef struct {
  int BaseAddrL;
  int BaseAddrH;
  char NumberOfLocations;

  ...
short int Location[5];
char LocationStatus[5];
short int BatteryStatus;
short int AuxBattStatus;
char Flags;
}DataPacket;

Note: XBee.xpp & Xbee.h are the same the code mentioned in the node.