Sociable Objects

Robert Faludi

April 2007

A thesis in the Interactive Telecommunications Program submitted to the faculty of the Tisch School of the Arts in partial fulfillment of the requirements for the degree of Master of Professional Studies at New York University

Sponsor: ___________________________ Nancy Hechinger

Reader: ________________________________
# TABLE OF CONTENTS

*Abstract/Thesis Statement* 3

*Research* 4

*Overview* 12

*Rationale* 12

*Goals* 14

*Audience* 15

*Environmental scenario* 16

*Core features* 16

*Media and technology proposed* 16

*Ballpark budget* 18

*Criteria for success* 19

*Project Narrative: the story of Sociable Objects* 20

**PRIOR KNOWLEDGE** 20

*INTERACTIVE TELECOMMUNICATIONS* 22

*PRINTED CIRCUIT BOARDS* 23

*C PROGRAMMING FOR ARDUINO* 25

*XBee RADIOS AND MESH NETWORKING* 28

*PROGRAMMING LIBRARIES* 31

*TEACHING* 38

**Process** 38

*Objects Database* 39

*Broadcast Clock* 40

*Slinky Metronome* 41

*Botanicalls* 45

**Results and Conclusions** 49

*Challenges* 50

*Successes* 51

*Future Directions* 52

**References** 53

**Other Documentation** 55
Abstract/Thesis Statement

The Sociable Objects project aims to explore how clocks, plants, communications devices, toys, artworks and wearables might work together to enhance their utility to people. My thesis will investigate the ways in which objects can share their data, learn about each other and perform intelligent behaviors based upon each other's states. I call them “sociable objects.” Modern devices sense and generate information, but they tend to keep this information local. The information sharing potential between devices has not been fully exploited. Objects usually don't share a language. Most importantly, they aren’t asking each other the right questions. Sociable objects will behave differently. They will be well-informed, friendly and aware of their context through communication with each other. I’ll create my own sociable objects, and I’ll begin the process of networking and socializing objects created by others.

The Sociable Objects Project has four major components:

1. The physical design and creation of the objects
2. The technical design of protocols, APIs and networking methodologies to support object interactions
3. Research into existing work regarding device-level communications.
4. Social networking to inform, influence and convince other makers to integrate sociable communications in their own creations. The end product will be a diverse menagerie of seemingly unrelated devices that can react to and interact with each other.
Research

Donald Norman estimates that the average adult must be ready to cope with around 20,000 different everyday objects (Norman, 1988). We are awash in a sea of things, and some of these are in some way animate. Active devices and objects surround us. From toasters to doorknobs, faxes to family sedans, light bulbs to library cards, we have created a world of artifacts that move, transform and manipulate information.

Modern artifacts seem poised on the brink of dramatically extended helpfulness. In the last century electrical power granted devices the power for autonomous motion, and clever mechanisms enabled those motions to follow simple ordered procedures. Washing machines knew to agitate before they spun dry. Toasters can (and still do) mechanically sense the transformation from bread to breakfast, then physically present us with the gloriously browned results. Of course, in the last 30 years, the computer revolution has eclipsed these mechanical computations with electrical ones that are both highly sophisticated and stunningly cheap. The average home contains 40-50 microcontrollers (Ganssle Group, ). These tiny chips bring the computer revolution to a tremendous variety of devices. There are 50 even in the cheapest automobile (Bauer, ). There are four in my wallet. These microcontrollers don’t quite bring sentience, but they certainly allow the incorporation of complex logical operations.

Another revolution underway adds communications to the mix. Devices are joining networks that allow them to commune with each other as well as with humans. Many of
the newest technologies spread a wireless web across our habitats, offices and schools. WiFi signals put Internet connectivity into every nook and cranny of 60 million homes and offices. Short-distance Bluetooth now comes pre-installed in 1 billion electronic gadgets. 100 percent of the world’s surface has satellite phone coverage, though most populated areas are pervasively served by less expensive mobile voice and data technologies such as GSM and CDMA. Hundreds of other wireless protocols support simpler devices like garage door openers, security systems and toys. Very recently, developments in mesh networking stand poised to release a new flood of communicative possibility. While most of the preceding technologies allow only point-to-point or hierarchical connectivity, mesh networks democratically support unmediated conversations between all devices that share a protocol. And there are a lot of protocols to choose from. Wikipedia lists 98 distinct protocols for mesh networking (Wikipedia contributors, ). While the race is hardly won yet, the current frontrunner is a widely adopted protocol called ZigBee. Like Bluetooth, ZigBee is designed for device interoperability between manufacturers, and is supported by a diverse body of engineers, semiconductor architects component producers and technology corporations. Unlike Bluetooth, ZigBee aims to create large networks of low-power devices, any one of which can exchange information with the others (Zigbee Alliance, ).

None of these wireless technologies describe a particular method for data exchange. They are generally agnostic as to the type of information carried. Some are clearly more suited to high-bandwidth applications such as video streaming, while others are designed to be best at handling short bursts of low-bandwidth information. Luckily, all have the
potential to carry communications in the current gold standard for data exchange, Extensible Markup Language (XML). XML is designed to facilitate the transfer of data across vastly disparate information systems, and to do so in a human-readable format (Wikipedia contributors, ). Because it is simple, generalized and extensible, XML is a natural choice for devices that need to share information.

Despite the wide base of technological advances available to modern-day devices, most fall short of being anything like “smart.” Weiser’s seminal 1991 paper on ubiquitous computing (aka. ubicomp) describes a world in which there are “hundreds of computers per room.” (Weiser, ) However, in his 2005 book Everyware, Adam Greenfield notes, “As a culture, we have so far been unable to craft high-technology artifacts that embody an understanding of the subtlety and richness of everyday life." (Greenfield, 2006)[p3] What is going on? Weiser’s vision of technological advances has largely come true, but his descriptions of a seamless web of invisible computing that supports our every move has clearly failed to occur. Greenfield adds, “[despite] a coordinated suite of devices and user interfaces, sensor grids, software architecture, and ad hoc mesh network strategies...very few of the people working in ubicomp or its tributaries seem to have quite gotten how all these pieces would fit together.” Clearly this is the puzzle that must be solved if devices are ever going to be smart (Greenfield, 2006)[p.16].

The ubiquitous computing movement may not have produced smart objects, but over the last 25 years it has undeniably produced a lot of interesting papers. While the variety of subjects and approaches is diverse, there’s a clear trend toward a certain type of
interaction. Much of the ubicomp literature involves objects that either receive
information from humans, or sense environmental information and report it directly to
them (Dey, Schmidt, & McCarthy, 2003). Input and output are clearly valuable
components, and the work done in these areas is laudable. From a sensor-laden room
capable of providing 150 hours of video and 200,000 sensor readings per day (Ubicomp,
p.193) to a Lava Lamp that associates eye movements with voice commands (Ubicomp,
[AuraLamp], to a wall-sized interactive LED display, there are few unturned stones.
However, despite the development of pervasive networking, a recent survey of the
literature revealed relatively little work that relates to information exchange between
objects or on-board processing. Humans are still tasked with the major role in
information transfer and processing. This is nothing new.

One of the first technological systems for connecting devices was developed in 1975. The
X10 system uses interior residential power lines to create a remote control network for
household appliances and devices. More than thirty years after its creation, X10 is still
the prevailing technology for this type of home networking, with millions of units in use
worldwide (Wikipedia contributors, ). Its ease of installation—most modules simply need
to be plugged in—along with its simple on/off/dim command set and relatively low cost,
have endowed it with a lasting popularity. Yet it is still relatively rare to come across a
house where X10 is in use. The system is mostly popular with home hobbyists because
even in its simplicity, the time and energy needed to utilize the remote control network is
outweighed by the modest value gained from having a nightstand controller for home
lighting. The trouble doesn’t lie so much in installation as it does in everyday use. It’s
hard to control so many devices directly. First there’s the spatial relationship of which remote switch corresponds to which lamp or appliance. Add a time element, and things become even more challenging. Consider the difficulties involved in coordinating events in advance. X10 allows for central timers and computers that can activate the home’s lights before its owner arrives in the evening. It can ensure that the coffee maker clicks on at 7:25, just before the coffee drinker awakens. And let’s stop there because accurately accomplishing even those two tasks turns out to be nearly impossible challenges. Nobody arrives home at exactly the same time. In fact given a bell-curve probability of arrival, the lights will go on wastefully early half the time, and uselessly late the other half. Some times they’ll be on all night even though the owner never arrives to need them. Coffee will show up on a schedule that ignores presence, sickness, insomnia and preference that is true to Ray Bradbury’s haunting depiction of a post-apocalyptic robotic home that carries on, though its human hosts have perished (Bradbury, 1958).

Of course, X10 is just one example of a technology that fails to put the pieces together for intelligent behavior. After 30 years of VCRs flashing 12:00, the example has become a cliché. Yet how many clocks in the world still do not set themselves? Office phones ring endlessly when office desks are clearly unoccupied. Cars moronically lock when the keys remain in the ignition. Loaded ferryboats will still unquestioningly run full speed into the shore. Our devices are annoying, they are inconvenient and worst of all they are dangerous. Our devices are anti-social.
An often-repeated design rule for human computer interfaces is that a computer should never ask a human for information that it already knows. For example, even in command-line interfaces, it should never be necessary to type the full directory path to run a program. The computer already knows all of its paths and can easily keep track of programs as they are installed. The user should only be bothered when the computer lacks the resources to take action on its own. As with any rule, this one is constantly violated. Order anything online and type New York in the city field. Chances are you’ll still have to tell the computer that New York City is in New York State. Yet if you mistakenly enter Alabama, the web site will respond later that your address is incorrect. There’s only one New York City, and the computer knew that to begin with, so why was it wasting your time? To a reasonable degree, the devices around us try to follow this rule, at least internally. Not only does the microwave oven beep when its timer has run out, it also shuts itself off. (If it’s hard to imagine how such an obvious interconnection could be ignored, consider regular ovens. Most will happily beep while they burn your dinner, or set fire to your home.) Air conditioners and refrigerators automatically cycle their compressors based upon internal thermostat readings. We are blissfully free from redundant confirmation requests such as, “room cold, continue chilling?” This bliss does not extend to information that’s readily available, but outside of the object’s casing. How many smoke detectors sound false alarms when a single crumb vaporizes in the toaster coils? The toaster knows that it is on, and in use. Motion detectors in the home security system know that someone is moving around in the kitchen. Everything is fine, but the smoke detector is incapable of sensing that. Rather than emitting a pleasant cautionary chime for a minute or two, it jumps right into the ear-piercing alarm that invariably
results in battery removal, rendering it useless. The smoke detector doesn’t exchange information with other devices. It is non-social. In fact it is also anti-social because its resulting poor behavior annoys humans. And what if that battery removal later ends up as an entry in a coroner’s report? When non-social object behavior turns anti-social would it be alarmist to label it sociopathic?

To solve this problem, devices need more than human partners. It is vitally important that they also communicate with each other. Only with access to the collective pool of information already present in other artifacts will devices have the raw material required for intelligent behavior. Several new technologies and systems are beginning to investigate this sociable realm. The TinyOS is an operating system for embedded network sensors developed by U.C. Berkeley’s EECS department and released as open source software (TinyOS, ). It is used in “motes” which were also developed at Berkeley as part of the Smart Dust Project. Named after dust motes, these tiny microcontrollers originally conveyed sensor data using a system of miniature mirrors to reflect a laser generated by a base station(Warneke, Last, Liebowitz, & Pister, 2001). Motes have been commercialized by Crossbow and now communicate using radio frequencies (Crossbow Technology, ). The Smart-Its project from Europe creates small microcontroller—embedded devices that are attached to other objects. Once attached, the Smart-Its adds sensing, information processing and communications capabilities to the original object or device (Gellersen, 2005). One very intriguing use of Smart-Its was the creation of industrial chemical containers that are able to independently detect potentially hazardous juxtapositions and issue alerts, without the use of any external infrastructure (Davies,
Mynatt, & Siio, 2004)[p.250]. The containers perform only rudimentary information processing and communication, but the results could some day be quantified in lives saved. These new technologies are a start, but more important is continuing to develop a new way of thinking about devices and their discourse so that the promise of a helpful and intelligent environment can become a reality.

Objects that can think and communicate may seem a futuristic fantasy. However, as Greenfield points out, "In a sense this is only a return to a much older tradition. For most of our sojourn on this planet, human beings have understood the physical world as a place intensely invested with consciousness and agency; the idea that the world is alive, that the objects therein are sentient and can be transacted with, is old and deep and so common to all the cultures of humanity that it may as well be called universal." [p.118] It is with respect for this ancient sense of the world that I undertake Sociable Objects and investigate whether someday all 20,000 everyday objects can behave in the intelligent fashion that would be most useful to humans.
Overview
When I first experimented with TCP/IP protocols in 1993, I boldly predicted that web pages and emails were really only the tip of the iceberg for information sharing, and that within a year or two we would see physical things start to share with and learn from a worldwide information network. Almost immediately, Cambridge University put a coffee pot on the web ((University of Cambridge, ). and my timeline appeared pessimistic. However while many other objects have appeared on the Internet, it has been a trickle rather than my predicted flood. Modern devices remain disconnected and uninformed. Mostly, I believe this is because the information sharing potential between these objects has not been fully exploited. My research has focused on investigating the possibilities for sharing information and took a deeper look at what appliances, sensors and artworks should be saying to each other, so they can better serve people. I built a number of devices that share their information and allow other things to trigger their behaviors. These include a clock that broadcasts real time, a lamp that brings natural light to dark or windowless rooms, a public metronome to coordinate multi-device electronic performances, and a plant care system that demonstrates their Sociable Object utility. I also began to coordinate the sharing of information and behaviors from devices and artworks made by other people.

Rationale
There is an ongoing debate in cognitive psychology about the sources of knowledge. Initially it was assumed that all the things we know about the world are learned, and though recent research has added a role for innate knowledge, direct learning is still
vastly important. People see, hear, feel and smell things that teach them about the world. Furthermore, they react to those stimuli with behaviors, including some that directly affect the input stream. One thing that makes humans smart is that we integrate information from a wide variety of sources. In fact one of the largest sections of the brain (and one of the least well understood) is the parietal lobe’s sensory integration area. Here discrete information from the five senses is combined to create a comprehensive picture of the world that guides our reactions and behaviors. People with damage to the sensory integration area have tremendous difficulty comprehending the world and managing their behaviors. When diverse information is not available or combined, we have trouble coping.

Humans are also highly social. We rely upon information transmitted by others even when we cannot see with our own eyes. In fact, much of what we know about the world comes from what we are told by others. We know about the back of the moon, the bottom of the ocean and the interior of atoms, though nearly none of us have experienced these things first-hand. Intelligent behavior is rooted in the integration of the direct sensory input with remote sources that we at least implicitly trust.

Humans tend to anthropomorphize active objects, but these objects behave nothing like us. In fact our modern devices cannot serve even as a sketchbook model of the human cognitive and behavioral system. They lack integral structures. Devices tend to get all their information from one source. For example, a smoke detector simply evaluates a sensor module that looks for particulate matter in the air, often mistaking innocuous
matter for toxic gas. Automatic flush toilets react to a single detector, which is sometimes fooled if the user leans forward. Even devices that take information from two or more internal sensor sources are unable to query the neighboring devices which hold pertinent information. Their behaviors are annoying and un-human because they lack access to the data that could help them act “smart.”

**Goals**
My intent is to prove that when devices exchange the right kinds of information and commands they appear more intelligent in their actions, and that the results can be more helpful to humans. The project should eventually produce publishable works that inspire other researchers to take my results as a jumping-off point for future investigations. I would also like the project to continue on the floor-wide test bed at NYU’s ITP, with new students continuing to use its infrastructure to share object information on an ongoing basis. Finally, I would like an everyday audience to understand the project and easily “get” why this new paradigm of object interaction would improve their devices and supplement their lives. I have created five Sociable Objects and about five more are in various stages of development. They will serve widely diverse purposes, use varied networking technologies and run from very small to wall-sized installations. I would also like to engage a number of other students in my work. Several are already participating. These students will electronically publish information generated by their own projects, and use data generated in others’ projects for new purposes. The end result should be enhancements for everyone involved, and some surprising repurposing of data that reveals how diverse information can be integrated into new knowledge.
Audience
The Sociable Objects project will engage a lot of different groups. It will clearly affect the ITP community because they will be viewers, users and participants. Objects will be displayed on the floor. For devices with a utilitarian purpose, the ITP community will be the likely users. My goal is to involve as many makers as possible. Anyone with a project that obtains data and is willing to share it, or creates a behavior that they are willing to have remotely triggered, can be a bona fide research participant. Beyond the ITP community, I hope to influence researchers at other institutions to consider, test and extend my work. To this end I’ll participate in conferences and eventually submit the results of my research for post-graduate publication. In the interest of reaching the broadest possible audience, many of my Sociable Objects, and those of my fellow makers will be whimsical and entertaining, as well as technically creative, so that they are worth of inclusion in the popular press. While I’d like for everyone who has contact with the project to fully understand it, I’d be satisfied if most viewers understand the basics, as long as a select few comprehend what I believe are the far-reaching implications.

There are no immediate plans for any specific business venture related to this project, however a successful outcome would clearly have commercial applications. Home automation, office efficiency and industrial monitoring systems could all benefit from improved device behaviors. It will be important to continue relating this model system to potential real-world applications.
**Environmental scenario**
The project will mostly take place on the ITP floor, and involve gallery and works-in-progress displays. Because much of the technical interaction and communications will be unseen, online documentation will be essential. A web site with detailed protocols and a complete database of participating projects will be available. Signage on each participating project should direct viewers there for further information.

**Core features**
The Sociable Objects project will feature open publishing of variables and methods (data sensed and behaviors available) between objects and devices. This publishing will provide a system of enhanced contextual information and interaction possibilities for the devices that participate. The project will be a model system for larger real-world data sharing. The database of participating objects—with complete documentation of their variables, methods, protocols, addressing information and contact information—will portray the available data landscape. That database will be put forth as a model for a larger system—one which engineers and manufacturers could use to plan, share and communicate for better and more sophisticated device interactions. Every object that participates will use external information or request external behaviors that enhance their utility or aesthetic value to humans.

**Media and technology proposed**
To investigate the Sociable Objects concept I will connect ITP projects that generate and sense data, amalgamating the various feeds into a network of information that enhances all projects that use it. While these objects will not display human intelligence, they
hopefully will approach a more thick-witted level of logic, one that can take advantage of contextual information because it has been made available. Various technologies will be useful in this endeavor. Each object will generate information using sensors connected to a microcontroller. The primary communications link between objects will be ZigBee radios, specifically the XBee brand radio from Maxstream that is currently in popular use on the ITP floor. Several data feeds will use eXxtensible Markup Language (XML) formatting to standardize communications while leaving ample opportunity for flexibility and extension. Proving that objects can be sociable, even when they don’t directly share technology is important to the project. To that end, other communications devices will eventually come into play including worldwide (GSM) cell phone modules, TCP/IP Internet communications via embedded XPort brand devices, Bluetooth radio links and perhaps even simple IR communication. To promote and enable participation in the project, I created the initial web site using a publicly editable PmWiki database to store all the project and protocol information. This site will include areas for code samples and discussion. It can easily be extended to serve other needs as they arise.

My current list of Sociable Objects includes:

- a real-time clock that provides accurate time of day to any requesting device
- a metronome that gives a constant beat to any musical project that wants to coordinate with others
- a ZigBee-to-Internet gateway that allows wireless devices to access online resources, and each other over long distances
- a lamp that brings natural daylight to dark and windowless spaces, using a rooftop
sensor

- the Botanicalls project [ref], enhanced by time, motion and internal interactions

I also plan to create:

- an interactive wall for full-body expression
- a home barcode and Radio Frequency IDentification (RFID) scanner with a simple interface, for easy capture of item-related events and desires
- a riff on Tom Igoe’s email clock (Igoe,) that can be worn around the neck, rapper-style

Other people have already expressed interest in joining the system. Their projects include:

- a lamp that tracks with the sun across the sky
- a robotic dog that responds to the proximity of the viewer
- a series of light sculptures that interact with each other, using motion detectors
- an set of interactive kinetic sea-grass sculptures that react to human presence
- The Roof-link system that brings solar panel data to the ITP floor
- an automated photo booth
- a swarm of interactive robotic evil babies that react to and interact with people

**Ballpark budget**

The typical Sociable Object that I am planning can be created for less than $200. The radio and microcontroller electronics will typically run under $40 per device. Certain items will be more expensive. Several sensors cost upwards of $100 and I plan to create
at least one object that uses a cell phone module costing $225. Some savings is available that will help counterbalance these expenses. Because sociable devices share information with each other, each does not need to house all the sensors and mechanisms that are used. This should result in some cost-savings over non-communicative versions. Wherever possible I intend to reuse components. Great materials can be obtained from second-hand stores, then hacked and modified as needed. Reuse is both economical and environmentally preferable. The documentation and publicity infrastructure on the web will be created with open-source free software, and will be hosted on existing servers.

Criteria for success
1. Results - My primary goal is to fully explore the question: Can openly sharing information between objects enhance their utility to people?
2. Participation – It is vital that at least three other people participate in the project to highlight the possibilities for data integration. Ideally ten or more participants will become engaged and benefit from the Sociable Objects infrastructure.
3. Complexity - Levels of value and complexity that illustrate Metcalfe’s Law\(^1\) are desirable. Simply put, the system should become much more useful as it grows.
4. Sharing – Because this is a research project, it is important that the results do not become buried. Wide dissemination of the outcome is essential, both through publications and teaching.

\(^1\) the value of a network is proportional to the square of the number of members \(\frac{n(n-1)}{2}\).
Project Narrative: the story of Sociable Objects

Before I could begin making Sociable Objects, I had to learn a lot of technical skills. I knew I would need to create electronic devices, program their behaviors, master wireless communications and devise protocols for inter-object data transmission. There was also a lot that I didn't know that I needed until the project got started, including creation of printed circuit boards, memory management, the creation of programming libraries for wireless communications, and plenty of patience.

PRIOR KNOWLEDGE

Before starting at ITP, I did have some technical background. I had run a computer company that did networking and web application development for start-ups and other progressive businesses. So I knew a fair amount about how computers communicate. Most of what I brought to ITP was about wired network communications, including TCP/IP—the digital language that runs the Internet. I had worked a bit with higher-level protocols that computer applications use to talk to each other, including email, post office protocol, file-transfer and web protocols (SMTP, POP3, FTP and HTTP respectively). Knowing about how these structured communications occur was a great leg up to understanding new wireless communications. The similarities of client/server call-response, packaging data in an envelope of addressing and description meta-data, terse commands and error correction structures all fed my comprehension of new methods and protocol standards. Because I had been involved with setting up and
maintaining small business systems beforehand, the mostly-obsolete task of configuring modems still occupied part of my permanent memory. Relatively obscure technical methods have a way of repeatedly reappearing. Even though analog phone lines, huge mainframes and dumb terminals are fading into the past, the methodology used in their communications keeps cropping up in brand new products. I thought I had left behind things like parity bits, AT commands, hardware handshaking, and serial communications, but the radios used in the Sociable Objects project were about to bring all of those seemingly-obsolete methods roaring back.²

Outside of a class in BASIC in high school, and a some simple scripting during the years running my consulting business, I never had much programming background or experience. Just before starting at ITP, I did a Masters degree in Cognitive Psychology. It turns out that these days psychology is very little about Freud and much more about neuroscience, statistics and of all things: programming. To keep my head above water, I was compelled to struggle through the coding of experiments in Matlab. Matlab is a mathematical environment for running simulations (which I was just learning) in matrix algebra (which I didn’t understand). It uses a homegrown language called M-code. M-code takes a little bit from a variety of standard computer languages and includes some experimental ideas of its own. For two-and-a-half years I wrangled with modifying code that I hadn't written, reading obscure documentation and my own misconceptions about structured programming. Along the way, I was unknowingly gaining familiarity with some standard syntax and developing my debugging skills. Painful as it was to learn by jabbing in the dark, this exposure would come in handy later.

² I often wonder if someday to operate a fantastically futuristic system, we’ll need to call on our ancestral memories of formatting floppy discs.
Growing up, through hobby projects, home telephone wiring and late-night lamp repair, I had an opportunity to do a bit of soldering and create some sparks while learning how simple electrical systems work. However I had never designed a working circuit or programmed a microcontroller. Even so, my basic initial background in computers and electricity would help accelerate me through the learning process and leave more room for attempting the more advanced techniques I would need for the Sociable Objects project.

**INTERACTIVE TELECOMMUNICATIONS**

The core curriculum at ITP includes one programming class and one in the electronics of physical computing. Both are structured similarly in that they eschew theory in favor of rapid prototyping and practical hands-on experience. Concepts that are commonly taught to engineering students over the course of two or three years are packed into a single semester. The intent is that students will gain a broad working knowledge that they can put into practice right away. The trade-off is that this is really only a starting platform for further exploration. Students don't emerge as computer scientists or electrical engineers so deeper skills must be self-acquired.

At ITP my Matlab struggles paid off almost immediately. Formal training in an organized classroom took the disparate chunks of knowledge that I already had, and began stringing them together like beads in a necklace. Arrays, functions, loops, variables, strings, classes, objects and threads all emerged of the helter-skelter jumble I had survived over the past two years and began to make sense. Several complex projects came--not effortlessly--but with remarkable fluidity. Simultaneously tackling lower-level
electronics brought out my passion for logically organized thinking matched with
creative methodology. Things not only had to work, they had to work for people. My
further studies therefore included interaction design, ethnography, advanced
microcontrollers and a strong series of small-group social projects, all of which employed
increasingly advanced networking architectures.

All of these courses emphasized iterative prototyping. They also inculcated an
appreciation for development shortcuts. The prototyping process thrives on innovative
tactics. Rather than slowly design perfect electronics and robust computing structures,
good prototyping encourages tricks to move the process along at lightning pace. After all,
the best way to learn is to get a project out in front of people. Prototypes aren’t meant to
last forever. They’re intended to create a brief experience that results in learning.
Sociable Objects is heavily informed by this spirit. Instead of focusing on designing
sophisticated application programming interfaces, or technically adroit communications
protocols, the project would center around creating logically helpful device behaviors.
How these behaviors came to pass, and even how robust they were over the long run was
not nearly as important as whether they were useful and an inspiration for further
research. Or at least that was the plan. Despite ITP’s focus on rapid prototyping, there’s
only so much useful information that can be gained if your project fails to function. It
turned out that technical work would end up playing a major role.

PRINTED CIRCUIT BOARDS
Electronic circuits are typically designed on “breadboards.” These are slotted
prototyping grids that allow for connections between components without the use of
permanent solder. While many projects can undergo early testing in this state, anything
that needs to travel, or endure repeated physical stresses of almost any kind, will be
subject to frequent misbehavior or outright failure if left in the breadboard state. An
intermediate step is to use “perfboard” which is a generic substrate for soldering together
electronic components. Perfboarding a project can be time-consuming, and when multiple
copies of a circuit are required, each must be individually crafted by hand. A better
solution is creating printed circuit boards or PCBs. These have several distinct
advantages. PCBs are typically designed in a CAD program on a computer and the
finished product is manufactured based upon digital output. Therefore they are robust,
repeatable and very inexpensive in large quantities. As soon as I knew that I would be
designing electronic devices for my thesis, I suspected that I would need to learn how to
make PCBs.

There are a variety of different CAD programs for creating schematics and circuit
boards. Many prototypers in our community prefer Eagle CAD³, a cross-platform
program that can be obtained for free, then later expanded into a full commercial version
if desired. Eagle CAD is notoriously confounding at first. The interface breaks rules and
conventions of human interface design that have been around for over twenty years.
Simple operations like cut and paste seem half backwards and three-quarters inverted.
Still, the price is right and substantial libraries of components are available, along with
plentiful local expertise. I spent a few days going through the entire tutorial and was
ready to being making my first schematic and circuit. It took quite a while the first time
through to get a design that I was happy with. Selecting components, fighting the

³ http://www.cadsoft.de/
interface, researching terminology and dealing with an arcane ordering process consumed far more time than I expected. Worse yet, when the board arrived it turned out that I had made at least two mistakes, which meant that the circuit could not function exactly the way it was printed. I feared my first masterpiece would end up as a coaster.

Luckily, several electronics manufacturers offer detailed tutorials on the PCB production process. One, from Spark Fun⁴ electronics, showed detailed methods for testing, rescuing and iterating through flawed designs on the way to working and robust versions. For example, by simply scraping away a bad connection with the side of a knife, and running a short wire across the bottom of the board, I could fix my errors and test whether the rest of the design was sound. In fact, the end result was a rock-solid board with minor cosmetic blemishes. Bolstered by the resiliency of the process, I went on to create several more PCBs. Each one has improved on the prior designs and I am now able to accurately create custom circuits that look and work similar to those built by people trained to know what they are doing.

**C PROGRAMMING FOR ARDUINO**

My ITP education provided background in programming micro-controllers in two languages, PICBasic PRO⁵ for PIC family of microprocessors and Arduino⁶ for the ARM ATMEGA8 and ATMEGA168 micro-controller environment. Since the summer of 2006, ITP has transitioned to teaching the Arduino environment as its main physical computing development platform. Arduino is a set of libraries that overlay the C and C++ languages.

---

⁶ [http://www.arduino.cc/](http://www.arduino.cc/)
It is intended to initially mask the more difficult processes of working in those environments directly. The Arduino command set is limited to a very small and very specific set of functions, all of which relate directly to operation of the ARM microprocessor. A major strength of the Arduino system is that, while knowledge of C is not initially required, applications that need extended features can be expanded directly by writing C code alongside the basic Arduino functions. For the Sociable Objects project it was clear that I would need to be able to create features and use structures that were not supported by Arduino. Therefore I set about teaching myself the basics of computer programming in C. Because I had both the struggle with Matlab and the revelations of ITP under my belt, I hoped it would be easy enough to teach myself a third language without significant outside support. This turned out to be true. I purchased SAMS Learn C in 21 Days and tried to go through a lesson every day for three weeks. The journey from Contents to Index ended up taking a month, and I gained even more insights into the programming process. Both Processing’s Java language, and Matlab’s M-Code have their historical roots in C. Structures and functions that seemed intentionally backwards or arbitrary when I learned them turned out to be mostly a result of legacy. A surprising number of bewildering syntactic and organizational features in the latter languages reveled themselves to be either continuations of existing C structures or direct reactions to long-term problems latent in the C system. Understanding these roots provided me with wonderful context to comprehend and parse the later languages’ syntax and architecture. More directly though, a formal self-education in C allowed me to begin creating the interactions that the Sociable Objects would require. A thorough study of the basic C concepts gave me detailed grounding in dealing with different types of variables.
It rapidly became clear that poor variable management might send an otherwise functional program into a mysterious death-spiral. I had already seen similarly unpredictable behavior in Arduino programs and was now prepared to look for integers that were being overflowed or variables that had not been properly initialized—the latter being especially dangerous because in C an uninitialized variable might be holding any number at all, and happily inject garbage into your program at random. The other two major take-aways were pointer handling, string functions and memory management. It had initially puzzled me that Arduino did not offer any type of string variables. Strings are containers for lines of text and much of what radio communication devices do is transfer text and text-related messages from place to place. It turned out that when C was developed, no strings were ever included. Instead, anything having to do with words was relegated to arrays of characters (arrays are like lists). This primitive approach continues to this day and has vast repercussions for novice programmers. Strings are fairly easy to understand, but array management is more abstract and daunting to the new programmer. In C, arrays are intimately tied in with pointers—one of the most arcane features of the C language. Pointers are methods of referring directly to memory locations. Sort of. In the course of coming to understand them, I read from my book, the Internet and various other publications (often during late-evening bookstore runs). Not one of these sources provides an entirely clear answer on its own. But by using all of them I slowly came into a working knowledge of the topic—or enough of one to begin making mistakes and eventually fixing them.

With a phone directory-sized coursebook and extensive practice behind me, I felt ready to tackle creating my own C code outside of the safe space provided by the
Arduino project. All that learning and the planned exercises were only half of the battle. The sole way to fully become conversant and proficient in programming is to work through real problems in code. So at the end of my self-study, I was actually just beginning the real learning process.

**XBee RADIOS AND MESH NETWORKING**

The goal of the Sociable Objects project is to share information; therefore a robust system for data communications is essential. Because many devices are mobile and battery operated, a wireless system with modest power requirements, standardized protocols and excellent reliability was needed to form the underlying platform. Small devices require small radios, but there are plenty of systems to choose from. Bluetooth radios offer standards, but only 11 can share a network simultaneously. WiFi Internet connections don’t have this limit, but the radio modules are both expensive and power-hungry. For both systems, configuring and running networks poses major difficulties. WiFi radios really require a base station of some sort and must be configured by hand in most cases to communicate with that base, making device deployment a technical chore. Bluetooth’s limitations include extensive discovery times that delay the initial delivery of time-sensitive information, fairly high cost and severe addressing limits. Luckily for this project, another wireless communications protocol had just emerged.

The ZigBee protocol is a wireless communications standard for mesh networking. It is built on top of the poorly monikered 802.15.4 protocol. In broad terms, 802.15.4 does all the local heavy lifting of information delivery while ZigBee handles larger organizational issues and long-distance routing so that data can cross many radios to
reach its destination. The concept of addressed communications and of mesh networking
has been around for many years, but ZigBee is the first major protocol to bring them
together in a unified industry standard. It is being rapidly adopted for industrial uses,
agricultural systems, home networking applications and scientific sensing systems. A
quick overview of the ZigBee features was all it took to commit to using it as the major
Sociable Objects platform. It turned out that my early commitment spread rapidly to the
electronic artists, artisans and tinkerers at ITP. Within six months of my first ZigBee
radio acquisition, there were over 250 in use on the ITP floor.

The particular brand of ZigBee radio that I selected is the XBee brand produced
by Maxstream\(^7\) in Lindon, Utah. This particular radio has a number of features that are
useful in prototyping and education. While most ZigBee brands require embedded C
programming for even the most basic communications, the XBee radios support
transparent driverless communications. They feature an “AT” command set for relatively
simple configuration. The AT commands are a legacy of analog modem communications.
They were developed by Hayes as a user-friendly method for issuing instructions to
modems directly from the computer keyboard without the need for any additional
programming resources. The upshot of all this is that XBees can be set up with unique
addresses and put to work doing paired data communications in a matter of minutes.
Additional commands and sophisticated strategies can be learned as needed, with the
culmination being their “API” mode. In that mode, information is sent and delivered in
highly structured formats similar to those available on other brands of ZigBee radios. The
quick initial setup paired with the deep full feature set is a natural match for prototypers

\(^7\) [http://www.maxstream.net/](http://www.maxstream.net/)
who are often more interested in linking two things together than learning the intricacies of data transfer protocols.

I set about learning everything I could about the XBee radios. The first step of reading the entire product manual from cover to cover was accomplished in a few sittings. Certain features were better explained than others. There was no problem in comprehending how to do basic communications, whereas the more advanced coordination and cyclic sleep modes required repeated re-readings and hands-on experimentation to really understand. I decide to set up a simple test between two PIC microcontrollers so that closing a switch on one would, via the XBees, light an LED on the other. In the spirit of good documentation, I took some pictures, then began writing some notes for myself. Over the course of an evening these notes turned into a more extensive guide and finally into a full-fledged tutorial on getting basic data communications running via XBee radios. I put the tutorial online[^8] and fairly soon had a little local readership that was training themselves with my instructions. I translated to PIC Basic Pro code to Arduino, and found that some of the functions required strings, pointers and my first real excursion into coding in C. Because this code was going to be published online as a practical example, I not only needed to write the C portions, I also had to properly explain what I was doing and how I was accomplishing it. This is one of the best ways to really learn because in penning exact definitions, I was forced to wholly consider what I had coded and why. Improvements resulted directly. I eliminated several spurious methods and came to understand a few nascent strategies intuitively by way of explaining them to potential novices. One code example led to another. I experimented

[^8]: http://itp.faludi.com/meshnetworking/xbee/
with remotely programming the Arduino, using the XBee radio directly as a sensor\textsuperscript{9}, attempted to measure distances exactly using signal strength (it’s impossible) and began working with other students to explore the more complex modes. Jeff LeBlanc and I tested power consumption and sleep modes\textsuperscript{10}. With Andrew Schneider we created a Max/MSP decoder for the direct sensing mode. Pollie Barden became probably the first person to try doing both input and output on multiple XBees without a microcontroller.

Along the way, I began to recognize that the transparent mode with the AT commands was terrific for many basic uses, but became cumbersome when sophisticated and rapid communications were needed. I realized that creating special libraries for the structured API mode was the only way to gain the full potential of the radios for rapidly interconnecting diverse devices. These code libraries would be essential for the Sociable Objects project, and would certainly also be useful to the growing population of XBee artists and prototypers.

\textit{PROGRAMMING LIBRARIES}

The idea behind structured communications is leave nothing ambiguous. When XBee radios are put into API mode, they begin to communicate in a way that allows the receiver to carefully unpack a tremendous amount of useful information in a logical hierarchy. Each transmission is enclosed in a packet of data that includes an identifier for its beginning, length, type and a calculation that can be made to ensure the data was not damaged in transmission. Depending on the type of information being communicated, additional layers of meta-data will wrap around each important byte to fully describe its

\textsuperscript{9} http://itp.faludi.com/blog/archives/2006/12/xbee_direct_io.html
\textsuperscript{10} http://itp.nyu.edu/~jl2515/sustain/xbee.htm
origin, context and meaning. Nesting information in this way greatly increases the efficiency and value of each communication. However it also adds a substantial amount of complexity for both the sender and the recipient. Rather than write and debug original code for each device that needs to parse the structured API format, it makes sense to create generalized programming libraries of functions that automatically take the received packets and put them properly into useful variables. Functional libraries create significant efficiencies for the original programmer, and well written ones offer novices easy access to advanced features, without having to know much if anything about how the code or communications happen in the first place. Because no code libraries for the XBee’s API mode existed when the Sociable Objects project began, everything had to be written from scratch.

I decided to begin by writing a set of functions in Arduino and C so that structured packets could be sent to and received from the XBee radios directly by the microcontrollers. Microcontrollers offer a challenging programming environment. The best chip available for Arduino is the AVR ATMEGA168. It can hold 16K of code and 1K of dynamic data. This represents about one-two-millionth of what my slightly underpowered laptop computer can handle. A double-spaced page contains about two kilobytes of data. My chip could only store half of that. Working within these limits meant gaining a deep understanding of memory management and clever strategies for minimizing storage space while not limiting functionality. During program development, immediate feedback is highly desirable. It is best to run code as soon as a few lines have been added, so that bugs are easily traceable to a small body of known changes. Text feedback in the way of debugging output helps to understand what might be going on
inside the program, so that behaviors and misbehaviors are more easily traceable to their source. Microcontrollers put several barriers into this process. Code cannot be run directly in the Arduino development environment. To test what had been written, it must first be downloaded to the microcontroller, which needs to be connected and prepped in several different ways. Each code download effectively takes about a minute. While this is not a long time, when multiplied over the thousands of revision cycles in typical programming work it can add hours or days to the process. The original Arduino platform provided no debugging output at all if the serial port was already in use, and the XBee needed that port. So I began by flying blind, and guessing what the output could be. That was nearly impossible so I installed the rudimentary Software Serial\textsuperscript{11} features created here at ITP. These were an enormous help because for the first time I was able to get some idea of what was going on inside the chip. I augmented the basic code with functions to properly manage strings. Now, rather than just single letter output, the libraries would now produce real English words (or the fragments thereof that would fit in the chip’s memory). Eventually, these features were recreated in the official Arduino libraries. While that meant revising all of my code for the slightly different syntax, it was worth it to be able to create libraries and examples that would work on everyone’s standard setup.

Four major tasks needed to be accomplished: sending commands, receiving feedback, sending data and receiving data. The first step required the basic tools for compiling a correct packet of information that would be understood by the XBee radio. Maxstream’s documentation is very complete; still understanding the details took quite a

\textsuperscript{11} http://www.arduino.cc/en/Tutorial/SoftwareSerial
bit of patience, especially the first time around. There was nobody to explain the tricky parts or clear up misconceptions, so some things had to be learned by trial and error. For example I spent a good hour on debugging the checksum calculation that tells whether each packet has been properly received. Finally I threw up my hands and went back to fully re-read the documentation, where I noticed that the final result of all the operations was not what I needed. (The final result had to be subtracted from 255 (or hex FF) before it would match what the radio expected.) While missing a single operation seems like a trivial error, there was no feedback about why the radio was not accepting my packet, so only combing through the code with documentation in hand finally revealed that the bug was on my end and not theirs. With the checksum problem finally fixed, I was able to send the radio configuration commands. I knew this was working because I could see the radio react properly, but the microcontroller wasn’t yet getting any feedback on the outcome of its requests. The second task was to read back the answer coming from the XBee. This code built on the sending code. The checksum issue had been cleared up so the next major hurdle was properly detecting input and handling variable numbers of packets that are returned in the response.

It’s worth explaining here that programming is not solely a logical operation. There’s a great deal of emotional fluctuation to be endured in the process. Joy, confusion, depression, euphoria and most especially frustration are frequent visitors to the mind of the coder. Each wave brings with it some bonuses and some challenges. It can be very difficult to creatively solve problems when one’s affect has gone deeply negative. Unfortunately these are the times that creativity is most needed. Instead the coder will have a tendency to try the same solution repeatedly, or make stabs in the dark that don’t
necessarily match the evidence at hand. In contrast, the up times created by solving tricky bugs and wielding workable code may bring excessive zeal to the process. This is often the root cause for over-engineering—a situation where simple specific solutions are replaced by hugely generalized intricate code objects which tend to fail mysteriously later, bringing the emotional cycle full circle. In the case of writing the incoming packet parsing code, I was subject to this entire cyclic process. Having made the radio respond to my commands, I euphorically plunged into a brilliant structure that could be used by any process in any program for any application. It turned out to be a mess. The structures created were unstable, violated important memory restrictions and to top it all off, they were totally unnecessary. In the dark inner times that followed, I focused on fixing the over-generalized code. When this failed a glum midnight walk turned out to be the setting for an epiphany. The code didn’t need to be fixed. It needed to be deleted and replaced with a far simpler structure. The replacement would be less flexible, but it didn’t at all matter because it would work perfectly—and it did. Pretty soon accurate responses were flowing back from the local radio. With these two transactions under my belt, I was ready to start on the third: sending actual data out to remote receivers.

The first two steps had been useful exercises, but provided little immediate advantages over configuring with AT commands. This third step of sending real data in structured packets would bring an enormous advantage. It meant that every packet sent could be addressed to a different recipient. In addition, a response received back from the radio would allow my program to know if the message had been received by the remote radio. The addressing process would be trivial, as opposed to the laborious method of switching back and forth constantly into command mode if the API were not being used.
Knowing whether responses had been received allowed for creation of the well-informed behaviors that one would expect from a Sociable Object. I had learned my lesson about over-engineering and set out to make a simple outgoing package, along with a neat function to receive the response. Each step in this process built on the prior one, so much of what had been a circuitous journey the first time was a now a familiar tromp down a well-cleared path. This is the groove of programming that makes it all worthwhile, the “flow state” of coding where each problem is a marvelous little puzzle that’s quickly solved and cleverly incorporated into the larger purpose of the code. Seeing packets going out to whichever radio I had addressed them to was deeply satisfying. I knew that the next step of receiving this information back would be the hardest to tackle so I took the success in the third step as inspiration which hopefully would pull me through the difficult fourth.

Receiving information in a structured format, wrapped up with meta-data about its source was the major reason for creating the XBee API libraries. Knowing where information comes from tells you a lot about how to handle that information. This is as true in structured computer communications as it is in human interactions. After all, news of upcoming layoffs means a lot coming from reliable Rachel in finance, but nothing at all if it comes from incompetent Eliot in shipping. In the same way, we almost always care about the source, as well as the content of incoming data. The most practical reason is for responses to incoming messages. If we don’t know the source there’s no way to address a reply. With the earlier informing my process, I launched into designing a simplified reception function. Minor struggles aside, this went well, but only at first. My self-training and early experience with string management in C was used, but then
overwhelmed. I had to stop, re-read and re-research all the methods so that I fully understood their structures and implications. At the same time, I was dealing with more memory management issues, but wisely decided to strip out other parts of the program and deal with one situation at a time. [As it turns out, I’m only now developing the proper tools needed for easily working near the memory limits of the Arduino.] Revisiting string management filled in some important blanks. I started to receive data in little broken spurts, then with some more fixes things began running smoothly. The structure wouldn’t work in every possible arrangement, but it was fine for most and could be improved and generalized more later as needed. Now when data was received, the microcontroller knew immediately where it was coming from, how strong the signal was, who to respond to and whether the data received exactly matched the data sent. No other radio technology used at ITP to date had provided so much useful information. Plenty of previously unattainable transactions were now possible. Senders knew immediately whether their data had been correctly received. This is useful in any application where the remote unit may travel out of range or whenever information needs to be resent if it is not correctly received the first time. Recipients knew where their data came from and could react accordingly, including making asynchronous responses directly to the sender. Thanks to the enveloping structure, all of these types of communications could be mixed together with command requests and responses without the microcontroller losing track.

Because the libraries are so useful, I commented them extensively and plan to wrap them up for online publication in the near future.
TEACHING

In the course of learning all these things, I discovered that I wasn't the only person at ITP who wanted to know them. Over the last nine months ITP students have bought over 250 XBee radios to begin networking their projects. And so I was drawn into teaching, mostly informally, at the same time as I was learning. I carried on student-to-student late-night "DriveBy" information sessions. I organized several weekend hands-on workshops. I gave a few informational class presentations. Once on 20 minutes notice, I was called on to teach an Introductory Physical Computing class. Mostly though, I did one-on-one tutorials and help sessions with almost every student using the radios. While time-consuming, this has been one of the most rewarding aspects of the project. I’ve been exposed to dozens of different applications for radio networking. The call to help with so many different projects meant that I had to use my knowledge over and over in different contexts. I often ran into situations that required new learning, and of course some students investigated features that I had glossed over, showing me tricks that came in handy down the line. Through responses to my blog posts I’ve communicated with people in the U.S., Canada, Europe and Asia about wireless networking. When I chose the name Sociable Objects, I did not realize how much the social part was also going to apply to me. The process has vastly informed my understanding of the technology, its applications and its broad usefulness to others. I hope to continue with teaching in the future.

Process

While the learning never stops, at some point the building has to begin in earnest. My goal was to make as many Sociable Objects as was possible given the limited timeframe. I also wanted to create documentation of both my own and other students’
Sociable Objects work. With the six or so weeks available to me for building, I set about doing both of these jobs.

**Objects Database**

Early on I decided that Sociable Objects’ documentation would be in the form of a database of projects, makers, data and protocols. The drawback of databases is that they make poor sketchbooks. Field and table structure tend to restrict creative ideas from coming forward because they will require significant organizational changes. A better way to get started on a documentation project is to create a site where records can be entered free-form, and later solidified into a formal data structure when it is clear what the fields, tables and formats should be created. I settled on using a wiki to begin storing the information. A wiki is simply a set of editable web pages. Any user can create a page about their project and document its shared or sharable features. I downloaded, installed and configured PMWiki\(^{12}\), which is widely used for physical computing at ITP. Of course, as it is with all nascent projects, the bulk of the initial documentation had to be created by the person running the project. So I went about meeting with makers and writing up all the networked projects that I could find. Initially I thought this task would be daunting, but it turned out that it generally only requires 10 or 20 minutes per meeting. The wiki, at [http://sociableobjects.com](http://sociableobjects.com) contains the following information about each project: name, location, contacts, web site, picture, description, public variables (data), public methods (behaviors), private variables and methods, current users, ideas for new uses, technical addressing info, technical protocol info, example code and restrictions on use.

---

\(^{12}\) [http://www.pmichaud.com/wiki/PmWiki/PmWiki](http://www.pmichaud.com/wiki/PmWiki/PmWiki)
Broadcast Clock

The very first sociable object to grace the ITP floor was the broadcast real-time clock that I created. The idea of the clock was to provide a service to the ITP floor both as a direct physical source of time information and as a basic network service that almost any other device could easily use. The very first version of the clock was developed in PIC Basic Pro, using a DS1307 real-time clock chip on a convenient breakout board from Sparkfun electronics. Its time signal can be picked up by any project on the floor that incorporates standard 802.15.4 radios and used for anything from time display to inter-project synchronization. An updated version added the ability to connect to an Internet gateway and from there poll the time servers at the National Institute of Standards to keep the broadcasts accurate, and to coordinate with Daylight Savings Time changes automatically. The automated update feature is really a form of object-to-object communication that creates the foundation of the Sociable Object philosophy. Why should a device bother people by requiring twice-yearly maintenance and monthly adjustments when another device on the network already offers the information needed to avoid those annoyance interactions?

To pick up the signal at ITP, other devices simply set their network address correctly and send out a “GET” command. The clock returns a time string response in the following format: *20061003143227 where the year is 2006, month is October, day is the 3rd, hour is 14 (2 p.m.), minute is 32 and second is 27. The clock also broadcasts the same machine-readable time string every ten seconds to all 65,534 addresses on one subnet, and in human readable format “Tuesday, October 3rd, 2006, 2:32 PM” on another. Projects that are not able to transmit GET requests can passively listen on these broadcast subnets for time signals.
**Slinky Metronome**

In December of 2006, I met Bill Buxton while he was visiting ITP. Bill has a vast background in electronic music and is currently Principal Researcher at Microsoft Research\(^{13}\). Bill asked about my work, and comprehended it immediately. He suggested that a great service for Sociable Objects would be some kind of shared chronographic code, so that musical projects could play along together in synchrony. He likened it to the SMPTE time code that applies a unique sequential ID to each frame of a motion picture. I loved the idea because, really it could be applied to any project doing time-based work. Electronic paintings could change along with robotic drums in synchronization with kinetic sculptures. All that was needed was a basic time code infrastructure. The broadcast clock could technically serve that purpose, but the type of information that it sent out was too linear. Most music has strong cyclic components. There’s a regular beat of some sort, measures that contain the beats and typically phrases and larger structures that repeat themselves in regular patterns. Of course, not all music is exactly like this, but the kinds that would be appropriate for coordinated orchestration are much more likely to share this cyclic arrangement. I decided that the best way to start out would be to build a metronome to keep regular time.

Metronomes are periodic mechanisms. They require some kind of physical or electronic oscillator to run them at a constant rate. Because I was building with a microchip, electronic oscillation was going to be available. It’s built right onto every microcontroller, to keep instructions running in sync. However it seemed important to expose the timekeeping mechanism physically so that this device was not an invisible hand but rather an obvious participant that viewers could interact with. I first considered

\(^{13}\) http://www.billbuxton.com/
hacking a regular metronome. This could provide the timekeeping mechanism and structural support in a unit that already worked. Still, while existing metronomes had mechanical timing they hid that timing from the viewer. So taking the idea to a further level of abstraction, I considered pendulums. Pendulums are used in clocks because of a unique physical property. No matter how far they swing, the time it takes for them to go back and forth remains the same. They swing faster in exact proportion to how far they travel. In physics terms, the period remains constant because velocity changes with amplitude. In the course of my final semester class in Mechanisms, the instructor Dustyn Roberts reminded us that mechanical springs act exactly as pendulums do. That’s why they are used in timing mechanisms in wristwatches. I loved the idea of using a big spring as the pendulum, and the biggest spring that came to mind was a Slinky toy. Slinkys are common, resilient and cost $3.99. I knew that a hanging Slinky would bounce up and down at a constant rate. All that was left was to figure out how to measure its movements.

A variety of schemes for measurement were considered. Magnets at the end of the spring could be paired with static Hall effect (magnetic) sensors to measure their movement. Hall effect sensors are cheap, but I had never worked with one and didn’t know if they would be stable over the ten-foot distances that the Slinky could reach at full extension. Piezo and other vibration sensors were considered, but the Slinky probably didn’t shake enough at its maximum accelerations because it ideally moves in a perfect sine wave pattern. My choice was to try an accelerometer. I had never worked directly with one before but had helped other students with projects that included them. This just one example of how teaching and learning can go hand-in-hand. Physics classes
being part of my more distant past, I needed to do research on what the acceleration pattern of a moving spring would be like. This required really understanding what acceleration is. It’s not the same as velocity or speed. Those are changes in position over time. Acceleration is one step further. It is defined as changes in velocity over time. For example when an airplane takes off, speed is increasing and passengers are pushed back into their seats. At cruising altitude however, even though speed is near 600 mph, acceleration is close to zero because that speed doesn’t change. Passengers can easily move around the cabin on foot. In the Slinky, the maximum accelerations are—somewhat counter intuitively—the moments of zero velocity. These occur at the top and bottom of the bounce. The middle of the oscillation is similar to the forces in the cruising jet.

Velocity is high but changes are low so acceleration goes momentarily to zero. My accelerometer would measure all this by hanging on the end of the Slinky as it moved up and down. By adding a microcontroller and radio to the package, it could broadcast the results out to any other device. The Slinky’s constant period would set a regular beat while it was moving, and the microcontroller could maintain this regular beat after eventually the Slinky settled into motionlessness.

I created a prototype of the electronics and used a smaller spring toy as my test case. Because the smaller slinky would not support the weight of my prototype, not to mention the laptop computer that needed to be attached for development, I only fastened the accelerometer to the end of the spring, and ran a wire to the rest of the package. This setup worked well, and I was able to learn the best ways to work with the accelerometer going from basic readouts, to finding zero points, to tracking transitions from positive to negative acceleration as the momentary trigger for my beats. I created a broadcast format
in 4/4 time to indicate which beat of the four we are currently on, as well as a uniquely identified measure so that instruments or electronic artworks can know where they are in the cycle and also where they are in linear time. Several other problems needed to be solved for the project to work right. I had to calibrate the accelerometer properly.

Readouts changed depending upon its exact orientation. This is because gravity is always providing a constant acceleration force that needs to be subtracted from the readings. Typically, I first made some complex attempts at calculating an average reading (the grand average of all readings will cancel each other out and be the calibrated zero point). Alas, memory limitations on the microcontroller restricted my ability to capture large data sets. Typically, again, I found there was a much simpler solution. It was very easy to tell when the Slinky had been motionless for a time, and take a simple reading of that value as a close estimation of the zero point. For a variety of reasons this isn’t necessarily as precise a calibration, but for my purposes that turned out to be totally irrelevant. Good enough in this case was identical to perfect. The other major challenge was to bring the weight of the package down low enough so that it could be supported by the Slinky. Slinkys are extremely elastic, so they will stretch to great lengths with minimal application of force. This makes them fun toys, and tough structures for electronic sensing and transmission packages. I decided that the only way to get everything small and light enough was to design my own printed circuit board so that no excess prototyping baggage would be riding on the end of my metronome. Single PCBs are expensive, so I only wanted to do the order once and get it right the first time. I designed, checked, rechecked and held my breath. Typically it takes a few tries to get a circuit board right, and I still pretty new at this. Luckily it worked. My board arrived and with
everything soldered into place the lights lit and beats were transmitted. I added code for running a small sound output buzzer and linked that to a hardware switch so it could be totally silenced as desired. I also added an output light so that function would be visible even when the sound was off. The entire package including battery turned out light enough to ride the end of the Slinky and a broadcast metronome was born.

Botanicalls

Botanicalls is one of my previous projects, created in partnership with Kati London, Rebecca Bray and Kate Hartman. It’s a system that allows plants to place phone calls for human help. When a plant on the Botanicalls network needs water or more light, it can call a person and ask for exactly what it needs. When people phone the plants, they orient callers to their habits and characteristics. While this may seem plenty sociable, it really isn’t in a Sociable Objects sense. The plants are contacting people, and the people are contacting the plants, but the system information is kept internal and never shared with any other devices. The plants will call for light even when it’s just been cloudy for a few days. They call day or night, without any regard to whether someone is awake or around to take the call. For this part of the project, I set out to change all that, and demonstrate how Sociable Objects can be better behaved objects.

Here’s the scenario, as it plays out in the Botanicalls installation set up in the ITP shared public areas:

Fiddle-leaf fig is low on light. Its light level for the last two days is only 30% of optimal and its Botanicalls event sensor is triggered. But is it time to make a phone call? As a Sociable Object device, its decisions don't need to rely on single data points. The fig begins making contact with the devices around it. First it checks with nearby Botanicalls plants. If they haven't gotten enough light either recently, the call for help will be postponed because it's probably been cloudy and
that's not a reason to disturb humans with a useless phone call. However in this case the spider plant, cuban oregano and prayer plant all respond that their light levels have been close to optimal. So it's not cloudy. In fact the fiddle-leaf fig also makes contact with the light sensor in the solar panel on the roof, which is monitoring its battery charging levels, but sociable enough to share its data with whomever asks. Sure enough, there has been plenty of sun. Now it's appropriate to make a call. Is anyone there to answer? Fiddle-leaf fig contacts a real-time clock and discovers that it is 4 p.m. Someone is likely to be around. A quick check-in with some nearby art projects that happen to use motion detectors confirms this. People are moving around the space, and at 4 p.m. they're probably students who will answer the phone. A call is placed and answered by a student, who heeds the request for light and moves the fiddle-leaf fig out of its shady corner and into a sunny window.

With that neat scenario in mind, I ventured once again back into the messy, magical world of microcontroller programming. This would be the first major test of my API. Even though the original Botanicalls worked with XBee radios in transparent/command mode, it hadn’t been stunningly reliable. Sometimes calls would fail to go through. Other times they’d be placed repeatedly because errors in the call transaction were apparently frequent. The idea was to use API mode to reduce the possibility of these errors, and keep the system more stable. The second important reason to use API mode was that the interchange between plants absolutely required it. Every plant needed the ability to ask all the others for light status. And once asked, each plant had to know whom to answer back. Transparent mode would not directly allow for this type of transaction. I glibly installed the freshly minted and essentially untested API libraries and gave a go at making the system run.
Initially things went pretty smoothly. I had created printed circuit boards for each Botanicalls plant. Laying out, designing, ordering and testing an initial prototype had shaken out a few minor errors. The full run of the boards was terrific. I spent a week soldering up the components to 15 of them in repeated short sessions. It was very pleasant to do and each board came out totally reliable. My prior experiences mounting projects on perf board were not so lucky. While all the projects worked eventually, each setup was a unique labor of love that had to be rigorously testing and debugged separately. The PCBs made a huge difference because I no longer really needed to check the electronics if something went wrong. I could concentrate on the code. I also created a new in-circuit programming setup for the Botanicalls PCBs, so that I could load code on them directly.

Sending commands to the XBee through my new API libraries went very smoothly. After correcting a minor typo here and there, I had the system seemingly working. There was just one nasty bug to track down. All of my commands worked, except when I tried to set an address to zero. Zero is completely valid so it was unclear what the problem might be. An hour or two later, not surprisingly after taking a short break, I found the problem in a sub-function that counted zero bytes for zero, which is technically true, but functionally damaging. As it is with most bugs, once found, it only took a few minutes to fix. After that sending commands was going swimmingly and I moved on to sending data.

The send data function took much more work because in the process of sending information, I also needed to receive responses. While this had worked in my test setup superficially, it had been designed in such a way that managed memory very poorly.
There were several different things wrong in the code, and unfortunately that made it exponentially harder to debug. Over the course of two frustrating days, I made very, very slow headway. Combing through the code I’d eventually recognize a problem, figure out obvious results, try uploading and re-running the program and not see a fix because several different bugs were interacting. It was like picking apart tangled fishing wire, which was doubly frustrating because my program looked pretty clean and neat. Slowly though, things began to work a little. I read up on memory yet again, wrote some additional functions to reveal how it was being used, and added a bunch of code to deal with the issues. Now my code didn’t look so clean but it had a distinct advantage. It ran without crashing. I took a day off from writing as I steeled myself for the major event—writing the call/response for plants to check each other’s light levels.

I first sketched out call/response on paper so I had some idea of a good approach. Then I put on a pot of coffee and typed in my first pass at the call/response function. Almost never does code run the first time—especially totally new concepts. Yet for some reason, this code worked immediately. When the plant needed light, it checked all of the others first before making a call. The darn things were just dying to be social. I knew it! Of course, there were still some logical behaviors to clean up such as making sure that the plants didn’t call to thank for being moved if they hadn’t complained in the first place. But that was easy and the power of these library functions began coming through strong. Once they had been properly debugged, they made writing new code trivial. I gleefully set about setting up longer-term tests of the new function, which continued to behave just perfectly.
Following my watershed with the inter-plant conversation functions, putting in the clock-checking code was a snap. I had written up some functions to check the clock for a previous project and these translated well to the new environment. I added a few niceties that I had discovered since, just to keep things buttoned up. Code to read in the last use of the metronome went similarly smoothly. A little back-and-forth was all that was required to make everything socialize properly. At this point, Botanicalls was doing great, but I’ve noticed that it is running out of room for new code. Only a little more socialization will fit on the chip. For this project that’s not much of an issue, however for others it is worth noting that complex interactions are always eventually limited by the amount of code storage space on the microcontroller. Significantly more intricate behaviors might require a vastly different structure—either putting a miniaturized full-scale computer into the device, or off-loading tasks to remote processors.

**Results and Conclusions**

The overall goal of this project has been to explore the ways in which objects can share their internal information with other devices, take into account more of their context, and therefore become more helpful to humans. There are times when I feel like I’ve only just begun the project. There seem to be so many different ways in which shared information could be helpful to devices that it may take a team of researchers to fully investigate the possibilities. On the other hand, a lot of work has already been accomplished:

1. Various Sociable Objects devices have been designed, brought into being and tested in real-world situations.
2. Several different protocols, one major API library and a variety of networking methods were created and employed to enable information sharing.

3. Existing methods and research were documented, along with a variety of schemes in which data is available to be easily incorporated by new devices as well as modified to fit existing ones.

4. A body of documentation and a methodology for creating more was developed and implemented so that makers can find and incorporate useful information feeds and behavioral requests into their own creations.

A variety of different Sociable Objects are now deployed, tested and communicating with each other. In the fall of 2007, ITP will offer a class that allows and encourages further development of new Sociable Objects. Because I’ll be leading that class, I should have an opportunity to continue this project with a team of creative students. There’s no doubt that they will enrich the venture and I hope I can help them employ the Sociable Object concept to empower their artwork and interactive ideas.

**Challenges**

Throughout the course of this project I faced several important challenges. While I kept assuring everyone that it wasn’t a project about programming or protocols, there was certainly a lot of programming protocols to be done. The uneven pace of code development made time management difficult. Sometimes the hour put aside for one little task would mushroom into a sleepless night. On the other hand, entire days that had been set aside for what I predicted to be a major imbroglio with the code turned out to be a simple 30 minute typing session that resulted in bugless perfection. Programming anything interactive means keeping track of not just one device, but the reactions and
interactions of many at the same time. I kept needing to simplify and keep my eye on the
goal of the communication, rather than become embroiled in its form. There was also the
simple difficulty of finding a quiet place to spread out my tools and noodle with the tasks.
A small New York City apartment and hectic crowded school lab meant a lot of my time
was spent packing, unpacking and moving from place to place in search of focus and
elbow room. Finally, I had a lot of things to learn and there’s still plenty that I don’t
know. Incomplete knowledge is certainly a challenge, but to me it has been a huge
opportunity because my understanding of technical and conceptual issues has had
numerous occasions to grow.

Successes

Learning was one of the most important outcomes of this thesis project. I now
really feel like I have a handle on C programming, radio communication, device
interaction protocols, development strategies and documentation methods. I’m far better
equipped to continue the investigation of Sociable Objects, and to think more deeply
about communication in general.

I also created a lot of stuff. There’s a wiki filled with Sociable Objects projects for
students and other ITPers to browse. A full set of libraries for the XBee radio has been
developed and should soon be ready for an initial release to the maker community. A
number of Sociable Objects have been completed, and several others are in various stages
of completion. Example code for call/response output & input, broadcast output & input,
and the XBee’s microcontroller-independent I/O mode have been produced for Arduino,
Processing and Max/MSP.
Most importantly, I have working Sociable Objects interactions that function well. They act as proof that sharing simple information between devices can result in useful behaviors that otherwise would not have been possible. The concept really works.

**Future Directions**

Moving forward I would like to continue building Sociable Objects. I think it’s very important to increase participation in the project. My fellow students have been very receptive to publishing the information latent in their creations for use by others. I believe that there is some critical level of available resources to be attained, at which point it will be easier and cheaper to utilize shared information than to create redundant data sources or ignore potentially useful behavioral requests. The Collaborative Mesh Networking class will make the fall of 2007 an important opportunity to jump-start this interaction. I’d also like to create project alliances with other classes, certainly with the New Interfaces for Musical Expression and Network Objects courses, but also with any other course that can use information and triggered behaviors to make a statement, tell a story or enable helpful actions.

Future research will include full mesh networking as a platform for sharing information interactively over larger distances and in complicated environments like outdoor urban spaces or between moving vehicles. There’s also a great opportunity in the Ubiquitous computing literature for research that generates findings about inter-object communications of any kind. Hopefully these ventures, and the growing participation of other Sociable Object enthusiasts will start making room in the world for devices that can share and language, ask each other the right questions and enhance their utility to people.

* * *
References

Bauer, P. EETimes.com - Recession-weary chip makers turn to auto
http://www.eetimes.com/story/OEG20030519S0042


Igoe, T. *Email clock*. http://www.tigoe.net/emailclock/mailclock1.shtml


TinyOS. *TinyOS Community Forum || An open-source OS for the networked sensor regime*. http://www.tinyos.net/


Other Documentation

Schematics, board designs and code for the Broadcast time clock, Slinky Metronome and Botanicalls are attached.
DEFINE OSC 20
Include "modedefs.bas"
' Set up Debug Out
DEFINE DEBUG_REG PORTC
DEFINE DEBUG_BIT 4
DEFINE DEBUG_BAUD 9600
DEFINE DEBUG_MODE 0

TRISB = %11100010

' set vars for clock runMode
'RTC pins
SDA var PORTD.0 ' I2C data pin
SCL var PORTD.1 ' I2C clock pin
SQW var PORTC.3

RTCSec var byte ' Seconds
RTCMin var byte ' Minutes
RTCHour var byte ' Hours
RTCDay var byte ' Weekday
RTCDay var byte ' Day
RTCMonth var byte ' Months
RTCYear var byte ' Year
RTCCtrl var byte ' Control
'conversion Variables
DecimalYears var byte
DecimalMonths var byte
DecimalDays var byte
DecimalWeekdays var byte
'conversion Variables
amPm var bit
DecimalMinutes var byte
DecimalSeconds var byte
address var byte
dataHours var byte
dataMinutes var byte
dataSeconds var byte
currentPAN var word
currentAddr VAR WORD
currentDestination VAR WORD
nistString var byte[48]
nistYear VAR BYTE
nistMonth var byte
nistDay var byte
nistHour var byte
nistMinute var byte
nistSecond var byte
nistDST var byte
dayTimeSuccess var BIT
offset VAR BYTE
lastDayTimeCheck VAR BYTE
lastBroadcast VAR BYTE
serMode var WORD
serMode = 84 'inverted: 16468 ttl: 84
'end vars for clock runMode

'Define serial output pins
TX var portc.6
RX var portc.7
RTS VAR portc.5
speakTX VAR portd.4
speakRX VAR portd.5
OUTPUT speakTX
INPUT speakRX

TESTLIGHT var PORTB.7
\ var byte
INPUTSwitch VAR portb.6
INPUT inputSwitch
lastSwitch VAR BIT

GOSUB blinkTwice
GOSUB setupXbee
GOSUB subgetTime
GOSUB subDayTime

serout2 speakTX, serMode, ["the clock is starting ",13]

'main loop
main:
GOSUB subListener ' check for time requests and respond
if lastDayTimeCheck != DecimalHours THEN ' set clock hourly via network
  GOSUB subDayTime
ENDIF
If lastBroadcast != DecimalSeconds / 10 THEN ' broadcast time frequently
  GOSUB humanTime
  GOSUB computerTime
  lastBroadcast = DecimalSeconds /10
ENDIF
DEBUG "*20",DEC2 DecimalYears,DEC2 DecimalMonths,_
  DEC2 DecimalDays,DEC2 DecimalHours,DEC2 DecimalMinutes,_
  DEC2 DecimalSeconds,10,13
if inputSwitch = 1 && lastSwitch = 0 THEN ' speak the time when requested
  GOSUB speakTime
ENDIF
lastSwitch = inputSwitch
goto main
'end main loop

subListener:
  ' wait for GET string on PAN C, and respond with computer coded time format
  currentPAN = $C
  GOSUB setPAN
  currentAddr = $0
  GOSUB setAddr
  currentDestination = $1
  GOSUB setDestination
  ' blocking function that times out after 300 milliseconds
  LOW RTS
  SERIN2 RX, serMode, 300, endListener, [WAIT ("GET")]
  HIGH RTS
  gosub subgetTime
  GOSUB sendCodedTime
  endListener: ' broadcast human readable formatted time on PAN CC
RETURN

humanTime: ' broadcast human readable formatted time on PAN CC
  currentPAN = $CCC
  GOSUB setPAN
  currentAddr = $0
  GOSUB setAddr
  currentDestination = $FFFF
GOSUB setDestination
gosub subgetTime
GOSUB sendFormattedTime
RETURN

computerTime: ' broadcast human readable formatted time on PAN CC
    currentPAN = $CC
    GOSUB setPAN
    currentAddr = $0
    GOSUB setAddr
    currentDestination = $FFFFFF
    GOSUB setDestination
    gosub subgetTime
    GOSUB sendCodedTime
    RETURN

subDayTime:
debug "daytime start"
    'change to XPort PAN & Address
    currentPAN = $7777
    GOSUB setPAN
    currentAddr = $0
    GOSUB setAddr
    currentDestination = $1
    GOSUB setDestination
    'open connection to NIST on port 13
    'read time into string
    GOSUB getDayTime
    if dayTimeSuccess == 1 then
        debug "daytime gotten"
        'parse string into time variables & set zone for daylight savings
        GOSUB parseDayTime
        'set clock
        GOSUB subSetClock
    ELSE
        DEBUG "daytime failed"
    endif
    lastDayTimeCheck = DecimalHours
    'change everything back to regular mode
    currentPAN = $CC
    GOSUB setPAN
    currentAddr = $1
    GOSUB setAddr
    currentDestination = $FFFF
    GOSUB setDestination
    RETURN

subgetTime:
    I2CRead SDA,SCL,$D0,$00,[RTCSec,RTCMin,RTCHour,RTCDay,RTCMonth,RTCYear,RTCCtrl]
    gosub bcdtodec
    return

bcdtodec:
    'Convert BCD to decimal... assume variable RTCSeconds holds the time in BCD format...
    'Years
    DecimalYears=RTCYear & $F0
    DecimalYears=DecimalYears>>4
    DecimalYears=DecimalYears*10
    DecimalYears=DecimalYears+(RTCYear & $0F)
    'Months
    DecimalMonths=RTCMonth & $10
    DecimalMonths=DecimalMonths>>4
DecimalMonths = DecimalMonths * 10
DecimalMonths = DecimalMonths + (RTCMonth & $0F)

'Days
DecimalDays = RTCDay & $30
DecimalDays = DecimalDays >> 4
DecimalDays = DecimalDays * 10
DecimalDays = DecimalDays + (RTCDay & $0F)

'Day of Weeks
DecimalWeekdays = (RTCWDay & $0F)

'Hours
DecimalHours = RTCHour & $30
DecimalHours = DecimalHours >> 4
DecimalHours = DecimalHours * 10
DecimalHours = DecimalHours + (RTCHour & $0F)

'convert from 24 hr time to 12
if (DecimalHours >= 13) then
    Decimal12Hours = DecimalHours - 12
    amPm = 1
else
    Decimal12Hours = DecimalHours
    amPm = 0
endif

'Minutes
DecimalMinutes = RTCMin & $70
DecimalMinutes = DecimalMinutes >> 4
DecimalMinutes = DecimalMinutes * 10
DecimalMinutes = DecimalMinutes + (RTCMin & $0F)

'Seconds
DecimalSeconds = RTCsec & $70
DecimalSeconds = DecimalSeconds >> 4
DecimalSeconds = DecimalSeconds * 10
DecimalSeconds = DecimalSeconds + (RTCSec & $0F)

RTCHour = (nistHour + 24 - offset) DIG 1
RTCHour = RTCHour << 4
RTCHour = RTCHour + ((nistHour + 24 - offset) DIG 0)
return

'convert decimals to BCD to send to clock chip
decimaltobcdLOW:
RTCSec = nistSecond DIG 1
RTCsec = RTCsec << 4
RTCsec = RTCsec + (nistSecond DIG 0)

RTCMin = nistMinute DIG 1
RTCMin = RTCMin << 4
RTCMin = RTCMin + (nistMinute DIG 0)
return
decimaltobcdHIGH:
RTCHour = (nistHour - offset) DIG 1
RTCHour = RTCHour << 4
RTCHour = RTCHour + ((nistHour - offset) DIG 0)

RTCDay = nistDay DIG 1
RTCDay = RTCDay << 4
RTCDay = RTCDay + (nistDay DIG 0)

RTCMonth = nistMonth DIG 1
RTCMonth = RTCMonth << 4
RTCMonth = RTCMonth + (nistMonth DIG 0)

RTCYear = nistYear DIG 1
RTCYear = RTCYear << 4
RTCYear = RTCYear + (nistYear DIG 0)

return

sendCodedTime:
    Serout2 TX, serMode, [*20", DEC2 DecimalYears, DEC2 DecimalMonths, 
                      DEC2 DecimalDays, DEC2 DecimalHours, DEC2 DecimalMinutes, 
                      DEC2 DecimalSeconds, 10, 13]
RETURN

sendFormattedTime:
    GOSUB outputWeekday
    Serout2 TX, serMode, [", "]
    GOSUB outputMonth
    Serout2 TX, serMode, [", DEC2 DecimalDays, " 20", DEC2 DecimalYears]
    SEROUT2 TX, serMode, [", dec2 DecimalHours, ":", dec2 DecimalMinutes, ":", 
                      dec2 DecimalSeconds, 10, 13]
RETURN

speakTime:
    GOSUB speakWeekday
    Serout2 speakTX, serMode, [" "]
    GOSUB speakMonth
    Serout2 speakTX, serMode, [" "]
    Serout2 speakTX, serMode, [DEC DecimalDays]
    GOSUB speakOrdinal
    Serout2 speakTX, serMode, [" ", 13]
    pause 1300
    Serout2 speakTX, serMode, [" 20", DEC2 DecimalYears, "]", 13]
    pause 1400
    SEROUT2 speakTX, serMode, [dec Decimal12Hours, "]
IF DecimalMinutes < 10 THEN
    SEROUT2 speakTX, serMode, ["oh "]
ENDIF
IF DecimalMinutes > 0 THEN
    SEROUT2 speakTX, serMode, [dec DecimalMinutes, "]", 13]
ELSE
    SEROUT2 speakTX, serMode, ["clock ", 13]
ENDIF
pause 1200
IF amPm == 1 THEN
    SEROUT2 speakTX, serMode, ["PM ", 13]
ELSE
    SEROUT2 speakTX, serMode, ["AM ", 13]
ENDIF
RETURN

subsetClock:
' Write command
  '   RTCSec = $00 ' Seconds
  '   RTCMin = $31 ' Minutes
  '   RTCHour = $12 ' Hours
  '   RTCDay = $04 ' Weekday
  '   RTCDay = $01 ' Day
  '   RTCMonth=$11 ' Months
  '   RTCYear=$06 ' Year
  '   RTCCtrl=$10 ' Control preset to output 1 second 'Tick' on SQWpin
I2CWrite SDA, SCL, $D0, $00, [RTCSec, RTCMin, RTCHour, RTCWDay, RTCDay, RTCMonth, RTCYear, RTCCtrl]
Pause 1000
return

outputMonth:
  select case DecimalMonths
  case 1
    Serout2 TX, serMode, ["January"]
  case 2
    Serout2 TX, serMode, ["February"]
  case 3
    Serout2 TX, serMode, ["March"]
  case 4
    Serout2 TX, serMode, ["April"]
  case 5
    Serout2 TX, serMode, ["May"]
  case 6
    Serout2 TX, serMode, ["June"]
  case 7
    Serout2 TX, serMode, ["July"]
  case 8
    Serout2 TX, serMode, ["August"]
  case 9
    Serout2 TX, serMode, ["September"]
  case 10
    Serout2 TX, serMode, ["October"]
  case 11
    Serout2 TX, serMode, ["November"]
  case 12
    Serout2 TX, serMode, ["December"]
  end Select
RETURN

outputWeekday:
  select case DecimalWeekdays
  case 1
    Serout2 TX, serMode, ["Sunday"]
  case 2
    Serout2 TX, serMode, ["Monday"]
  case 3
    Serout2 TX, serMode, ["Tuesday"]
  case 4
    Serout2 TX, serMode, ["Wednesday"]
  case 5
    Serout2 TX, serMode, ["Thursday"]
  case 6
    Serout2 TX, serMode, ["Friday"]
  case 7
    Serout2 TX, serMode, ["Saturday"]
  end Select
RETURN

speakMonth:
  select case DecimalMonths
  case 1
    Serout2 speakTX, serMode, ["January"]
  case 2
    Serout2 speakTX, serMode, ["February"]
  case 3
    Serout2 speakTX, serMode, ["March"]
  case 4
    Serout2 speakTX, serMode, ["April"]
  case 5
    Serout2 speakTX, serMode, ["May"]
case 6
    Serout2 speakTX,serMode,"June"
case 7
    Serout2 speakTX,serMode,"July"
case 8
    Serout2 speakTX,serMode,"August"
case 9
    Serout2 speakTX,serMode,"September"
case 10
    Serout2 speakTX,serMode,"October"
case 11
    Serout2 speakTX,serMode,"November"
case 12
    Serout2 TX,serMode,"December"
end Select
RETURN

speakWeekday:
    select case DecimalWeekdays
    case 1
        Serout2 speakTX,serMode,"Sunday"
case 2
        Serout2 speakTX,serMode,"Monday"
case 3
        Serout2 speakTX,serMode,"Tuesday"
case 4
        Serout2 speakTX,serMode,"Wednesday"
case 5
        Serout2 speakTX,serMode,"Thursday"
case 6
        Serout2 speakTX,serMode,"Friday"
case 7
        Serout2 speakTX,serMode,"Saturday"
end Select
RETURN

speakOrdinal:
    select case DecimalDays
    case 1
        Serout2 speakTX,serMode,"st"
case 2
        Serout2 speakTX,serMode,"nd"
case 3
        Serout2 speakTX,serMode,"rd"
case 21
        Serout2 speakTX,serMode,"st"
case 22
        Serout2 speakTX,serMode,"nd"
case 23
        Serout2 speakTX,serMode,"rd"
case 31
        Serout2 speakTX,serMode,"st"
CASE ELSE
    SEROUT2 speakTX,serMode,"th"
end Select
RETURN

setupXbee:
    configure: ' label to jump back to if configuration times out

    ' blink status light twice on startup
    GOSUB blinkTwice

    ' for some reason it seems to help to send an arbitrary character first
    ' then pause for the guard time before requesting command mode
serout2 TX, serMode, ["X"]
pause 1100

' put the XBee in command mode
serout2 TX, serMode, ["++"]

' wait for a response from the XBee for 2000 ms, or start
' over at the configure label if no valid response comes
LOW RTS
SERIN2 RX, serMode, 2000, configure, [WAIT ("OK")]
HIGH RTS

' set the Guard Time to 50 ms (same as hex 0x32)
serout2 TX, serMode, ["ATGT32,"]

' set the RTS pin to use flow control so that the XBee doesn't send unless PIC is listening
serout2 TX, serMode, ["D61,"]

' exit command mode
serout2 TX, serMode, ["CN",13]

' wait for a response from the XBee for 2000 ms, or start
' over at the configure label if no valid response comes
LOW RTS
SERIN2 RX, serMode, 2000, configure, [WAIT ("OK")]
HIGH RTS

RETURN

setPAN: ' label to jump back to if configuration times out
' pause for the guard time before requesting command mode
pause 65

' put the XBee in command mode
serout2 TX, serMode, ["++"]

' wait for a response from the XBee for 2000 ms, or start
' over at the configure label if no valid response comes
LOW RTS
SERIN2 RX, serMode, 2000, setPAN, [WAIT ("OK")]
HIGH RTS

' set the PAN (personal area network) ID number
serout2 TX, serMode, ["ATID",HEX currentPAN,","]

' exit command mode
serout2 TX, serMode, ["CN",13]

' wait for a response from the XBee for 2000 ms, or start
' over at the configure label if no valid response comes
LOW RTS
SERIN2 RX, serMode, 2000, setPAN, [WAIT ("OK")]
HIGH RTS

RETURN

setAddr:
' pause for the guard time before requesting command mode
pause 65

' put the XBee in command mode
serout2 TX, serMode, ["++"]

' wait for a response from the XBee for 2000 ms, or start
' over at the configure label if no valid response comes
LOW RTS
SERIN2 RX, serMode, 2000, setPAN, [WAIT ("OK")]
HIGH RTS

' set the local address
serout2 TX, serMode, ["ATMY",HEX currentAddr,","]
' exit command mode
serout2 TX, serMode, ["CN",13]

' wait for a response from the XBee for 2000 ms, or start
' over at the configure label if no valid response comes
LOW RTS
SERIN2 RX, serMode, 2000, setAddr, [WAIT ("OK")]
HIGH RTS

return

setDestination:
' pause for the guard time before requesting command mode
pause 65
' put the XBee in command mode
serout2 TX, serMode, ["+++"]

' wait for a response from the XBee for 2000 ms, or start
' over at the configure label if no valid response comes
LOW RTS
SERIN2 RX, serMode, 2000, setPAN, [WAIT ("OK")]
HIGH RTS

' set the local address
serout2 TX, serMode, ["ATDH0,DL",HEX currentDestination",""

' exit command mode
serout2 TX, serMode, ["CN",13]

' wait for a response from the XBee for 2000 ms, or start
' over at the configure label if no valid response comes
LOW RTS
SERIN2 RX, serMode, 2000, setDestination, [WAIT ("OK")]
HIGH RTS

RETURN

getDayTime:
' send connection string to xport
' time.nist.gov on port 13 (more info at http://tf.nist.gov/service/its.htm)
serout2 TX, serMode, ["C192.43.244.18/13",13]
ddebug " sent connection request... "
' check for C and fail if you don't get it
LOW RTS
SERIN2 portc.7, serMode, 2000, dayTimeFail, [WAIT ("C")]

' get and parse the time string from NIST
' for some reason, the RTC expects these values in the
' HEX version of decimal numbers. Weird, but it works.
SERIN2 RX, serMode, 10000, dayTimeFail, [WAIT (" "), dec2 nistYear]
SERIN2 RX, serMode, 2000, dayTimeFail, [WAIT ("-"), dec2 nistMonth]
SERIN2 RX, serMode, 2000, dayTimeFail, [WAIT ("-"), dec2 nistDay]
SERIN2 RX, serMode, 2000, dayTimeFail, [WAIT (" "), dec2 nistHour]
SERIN2 RX, serMode, 2000, dayTimeFail, [WAIT (":"), dec2 nistMinute]
SERIN2 RX, serMode, 2000, dayTimeFail, [WAIT (":"), dec2 nistSecond]
SERIN2 RX, serMode, 2000, dayTimeFail, [WAIT (" "), dec2 nistDST]
HIGH RTS

ddebug " year:"
ddebug dec nistYear
ddebug " month:"
ddebug dec nistMonth
ddebug " day:"
ddebug dec nistDay
ddebug " hour:"
ddebug dec nistHour
ddebug " minute:"
ddebug dec nistMinute
ddebug " second:"
ddebug dec nistSecond
ddebug " DST:"
debug dec nistDST

dayTimeSuccess = 1 ' if we made it through this code then things worked
dayTimeDone: ' we'll end up here if things failed
RETURN

dayTimeFail: ' this code will run on daytime lookup failure
dayTimeSuccess = 0
GOSUB blinkTwice
GOSUB blinkTwice
GOSUB blinkTwice
GOTO dayTimeDone
RETURN

parseDayTime: ' this function parses UTC into Eastern Time with Daylight Savings if needed
IF nistDST > 51 || nistDST == 0 || nistDST == 1 THEN
    offset = 5 'standard time
ELSE '(if nistDST < 50 && nistDST > 1) || nistDST==50 || nistDST==51 then
    offset = 4 'daylight time
ENDIF
IF (nistHour + 24 - offset) < 24 THEN
    gosub decimaltobcdLOW
    ' to simplify the code, prevent having to subtractively
    ' calculate hour, day, month and year
    ' at times when UTC of these is later than local
ELSE
    gosub decimaltobcdLOW
    gosub decimaltobcdHIGH
ENDIF
RETURN

blink:
    HIGH testLight
    pause 250
    Low testLight
    Pause 250
RETURN

blinkTwice: 'testlight
for i = 1 to 2
    HIGH testLight
    pause 50
    Low testLight
    Pause 250
next i
'end testlight
RETURN
int ledPin = 5;    // declare pin for the LED
int buzzerPin = 4;
//int Xpin = 0;    // slider variable connected to analog pin 0
//int Ypin = 1;    // slider variable connected to analog pin 1
int Zpin = 2;
//int Xval = 0;     // variable to read the value from the analog pin 0
//int Yval = 1;     // variable to read the value from the analog pin 1
int Zval = 0;
int lastZval = 0;
int noiseAllowance = 10;
int threshold = 25;
int calibrationPoint = 750; // initial guess at a value which represents stillness
int ctr=0;
long thisTick = 0;
long lastTick = 0;
boolean positivePhase = true;
live tickAverage = 1300;
int tickArray[10];
int tickArraySize = 10;
long tickCounter = 0;
long sendTime = 0;
int beatCount = 4;
int beatsPerTick = 2;
int beat = 0;
long timeCode = 0;

void setup()
{
  pinMode(ledPin, OUTPUT);    // initializes LED output
  pinMode(buzzerPin, OUTPUT); // initialize buzzer output
  Serial.begin(9600);          // turn on the serial port
  for(int i = 0; i < tickArraySize; i++) {
    tickArray[i] = 0;
  }
  blinkLED(ledPin,2,100);
  setupXBee(

  }

void loop()
{
  Zval = analogRead(Zpin); // read the accelerometer Z axis
  if (Zval > lastZval-noiseAllowance & & Zval < lastZval+noiseAllowance) {
    if (ctr < 10000) {
      ctr++;
    }
    else if (ctr == 10000) {
      calibrationPoint = Zval;
      ctr=0;
    }
  } else {
    ctr=0;
  }

  long now = millis(); // mark the current time
  if (Zval < calibrationPoint-threshold & positivePhase == true) {
    //mark entry into negative phase of cycle;
positivePhase = false;
}
// if phase has gone positive from negative
if (Zval > calibrationPoint + threshold && positivePhase == false) {
    thisTick = now - lastTick; // calculate the tick length
    lastTick = now; // set the last tick value to use next time
    positivePhase = true;
    // and if this tick is of some reasonable length (not too short or long)
    if (thisTick > tickAverage * .6 && thisTick < tickAverage * 1.6) {
        buzz(buzzerPin, 2500, 100); // buzz the buzzer at 2500Hz for 100 milliseconds
        blinkLED(ledPin,1,50);
        for (int i = tickArraySize-1; i > 0; i--) {
            tickArray[i] = tickArray[i-1]; // move each array member to the left
        }
        tickArray[0] = thisTick; // insert the latest tick into the array
        tickCounter++;
        tickAverage = 0;
        for (int i = 0; i < constrain(tickCounter,0,tickArraySize); i++) {
            tickAverage = tickAverage + tickArray[i];
        }
        tickAverage = tickAverage / constrain(tickCounter,0,tickArraySize);
        //Serial.print("avg: ");
        //Serial.println(tickAverage);
    }
}
if (millis() - sendTime > tickAverage / beatsPerTick) {
    Serial.print("measure: ");
    Serial.print(timeCode);
    Serial.print(" beat: ");
    Serial.print(beat + 1);
    Serial.print(" (avg: ");
    Serial.print(tickAverage);
    sendTime = millis();
    beat = (beat + 1) % beatCount;
    if (beat == beatCount - 1) {
        timeCode++;
    }
    Serial.print(" ctr: ");
    Serial.print(ctr,DEC);
    Serial.print(" calibration: ");
    Serial.print(calibrationPoint);
    Serial.println("\n");
}
lastZval = Zval;
}

void setupXBee() {
    boolean success = false;
    int ctr = 0;
    while (success == false && ctr < 100) {
        // blink the status LED
        blinkLED(ledPin, 2, 250);
        // an arbitrary byte to wake up the XBee
        Serial.print("X");
        //delay(1100);
        // put the XBee in command mode
        Serial.print("+++\n");
        delay(1100);
        // wait for a response from the XBee for 2000 ms, or start
        // over with setup if no valid response comes
        // set the PAN (personal area network) ID number
        // set the MY (16 bit address)
// set the Destination High to 0x0
// set the Destination Low (16 bit address)
// exit command mode (using a Serial.printLN to end the command with a linefeed)
Serial.println("ATRS,ID3333,MY64,DH0,DLS,CLN");
if (checkFor("OK", 1000)) {
    // if an OK was received then continue
    success = true;
} else {
    success = false;
}
ctr++;
}

//////////////////////////////// UTILITY FUNCTIONS //////////////////////////////////////////
// this function checks for a specific response on the serial port
// it accepts a string to look for and a timeout in milliseconds
int checkFor(char* desiredResponse, int timeout) {
    int result = 0;
    int length = 40;
    char incomingResponse[41];
    memset(incomingResponse, 0, length); // initialize all incomingResponse string positions to null
    char inByte = NULL;
    long startTime = millis(); // makes the start time = to now
    char* ptr_incomingResponse = incomingResponse;
    // while we haven't timed out or gotten back the string that we are looking for
    while (millis() - startTime < timeout && strstr(incomingResponse, desiredResponse) == NULL) { // strstr compares strings
        if (Serial.available() > 0) { // if there are any bytes waiting to be read
            inByte = Serial.read(); // read one byte
            if (incomingResponse > ptr_incomingResponse - length) { // if we haven't read in 80 characters yet
                *ptr_incomingResponse = inByte; // put the byte into the current position in the string
                ptr_incomingResponse++; // advance to the next position in the string
            } else {
                // move the last char to be next to last, and so forth until we reach the end of the array.
                for (int i = 0; i < length; i++) {
                    incomingResponse[i] = incomingResponse[i + 1];
                }
                incomingResponse[length - 1] = inByte; // put the byte into the current position in the string
            }
        } else {
            // move the last char to be next to last, and so forth until we reach the end of the array.
            for (int i = 0; i < length; i++) {
                incomingResponse[i] = incomingResponse[i + 1];
            }
            incomingResponse[length - 1] = inByte; // put the byte into the current position in the string
        }
        if (strstr(incomingResponse, desiredResponse) != NULL) { // if the desired string is found
            result = 1;
        } else {
            result = 0;
        }
        return result;
    }
}

// this function blinks an LED light as many times as requested
void blinkLED(int targetPin, int numBlinks, int blinkRate) {
    for (int i = 0; i < numBlinks; i++) {
        digitalWrite(targetPin, HIGH); // sets the LED on
        delay(blinkRate); // waits for a blinkRate milliseconds
        digitalWrite(targetPin, LOW); // sets the LED off
        delay(blinkRate);
    }
}
void buzz(int targetPin, long frequency, long length) {
    long delayValue = 1000000/frequency/2; // calculate the delay value between transitions
    // 1 second's worth of microseconds, divided by the frequency, then split in half since
    // there are two phases to each cycle
    long numCycles = frequency * length/ 1000; // calculate the number of cycles for proper timing
    // multiply frequency, which is really cycles per second, by the number of seconds to
    // get the total number of cycles to produce
    for (long i=0; i < numCycles; i++) { // for the calculated length of time...
        digitalWrite(targetPin,HIGH); // write the buzzer pin high to push out the diaphram
        delayMicroseconds(delayValue); // wait for the calculated delay value
        digitalWrite(targetPin,LOW); // write the buzzer pin low to pull back the diaphram
        delayMicroseconds(delayValue); // wait again for the calculated delay value
    }
}
/* This program helps a plant to make a phone call. 
* Rob Faludi with additional code from various public examples 
* http://www.botanicalls.com 
* A project with Kati London, Rob Faludi, Kate Hartman and Rebecca Bray */

char version[] = "1.83";

boolean demoMode = false; // set this to true for short timeouts and no data uploads
boolean sociable = false; // set this to true to use the Sociable Objects code
byte XPortAddr = 0x64;

char* getData(char* dataIn, int timeout);

byte plantID; // unique plant identifier

// time intervals for regular use
// THESE WILL BE MODIFIED IN THE SETUP FUNCTION IF DEMO MODE IS ON
int wateredTimeInterval = 30; // watering event interval in seconds
unsigned int lightTimeInterval = 18*60*60; // light intervals in seconds (for hours use h*60*60)
unsigned int moistTimeInterval = 18*60*60; // moisture interval in seconds
int dataInterval = 1800; // data sending for SQL, interval in seconds

// the positive increase in moisture that defines a watering event
int wateringCriteria = 100;

int lightNumSamples = 10; // number of sampled light values at the determined intervals
int moistNumSamples = 10; // number of sampled moisture values at the determined intervals

// most desirable state of wetness during a watering event--PLANT SPECIFIC VALUE
int desiredSoakLevel[13] = {
  600, 455, 280, 660, 700, 700, 720, 700, 460, 600, 600, 550};
// sufficiently dried out to be watered---PLANT SPECIFIC VALUE
int minMoist[13] = {
  450, 350, 1, 550, 400, 590, 500, 610, 500, 300, 560, 560, 330};
// Unhealthfully dried out, desperately needs water---PLANT SPECIFIC VALUE
int maxDry[13] = {
  400, 280, 0, 490, 320, 500, 400, 510, 400, 200, 490, 490, 200};
int lightMinimum = 96; // generally desired light level. Plant specific light threshold TBD

// set up the moist array of n values---replace 10 w/96
int moistValues[10];

int moistArraySize = moistNumSamples; // size of the soil moisture array
int lightArraySize = lightNumSamples; // size of the light array
int lightValues[10]; // set up the light array of n values---replace 10 w/96

// pins for PCB layout
byte ledPin = 13; // generic status LED
byte moistLED = 9; // LED which indicates the plant needs water
byte sleepPin = 4; // bring this pin HIGH to make the XBee sleep and LOW to wake it up (connects to pin 9 on XBee)
byte ID1 = 5, ID2 = 6, ID3 = 7, ID4 = 8;
byte moistPin = 0; // moisture input is on analog pin 0
byte lightPin = 1; // photocell input is on analog port 1
byte batteryPin = 12; // monitor battery level. active is high

// limits the maximum number and frequency of calls
int callSpacing = 20; // minimum seconds between each call
long lastCallTime = 0; // holds the last time we made a call
int callLimit = 30000; // maximum number of attempted http connects to call
int calls = 0; // holds the current number of attempted http connects
int lastWatered = 255; // this holds the last value of watering event check, set high to ensure descent
int lastMoist = 255; // this holds the last value of the moisture check, set high to ensure descent
int lastLight = 255; // this holds the last value of the light check, set high to ensure descent

// sociable objects section
boolean neglected = false; // true if others had light when you didn't
int lastSender = 0; // global variable for the address of the most recent XBee sender

// structure for holding time values
typedef struct {
    int year;
    byte month;
    byte day;
    byte hour;
    byte minute;
    byte second;
} ClockTime;

ClockTime time;

int callSpacing = 10; // minimum seconds between each call

void setup () {
    /// for(int i = 0; i < lightArraySize; i++) {
    ///     lightValues[i] = 1023;
    /// }
    plantID = (digitalRead(ID1)) + (digitalRead(ID2) << 1) + (digitalRead(ID3) << 2) + (digitalRead(ID4) << 3);
    for(int i = 0; i < moistArraySize; i++) {
        moistValues[i] = 0;
    }
    // start up the serial connection with 9600-8-n-1-true (non-inverted):
    Serial.begin(9600);
    debug.begin(9600); // start software serial process
    pinMode(ledPin, OUTPUT);
    pinMode(moistLED, OUTPUT);
    pinMode(sleepPin, OUTPUT);
    pinMode(ID1, INPUT);
    pinMode(ID2, INPUT);
    pinMode(ID3, INPUT);
    pinMode(ID4, INPUT);
    pinMode(debugRX, INPUT);
    pinMode(debugTX, OUTPUT);
    pinMode(batteryPin, INPUT);
    /// debugging serial output///////
    debug.print("start v");
    debug.print(version);
    debug.println(plantID, DEC);
    
    // time intervals for demo mode
    if (demoMode == true) {
        wateredTimeInterval = 5; // watering event interval in seconds
        lightTimeInterval = 5; // light intervals in seconds
        moistTimeInterval = 5; // moisture interval in seconds
        dataInterval = 3600; // data sending for SQL, interval in seconds
        int callSpacing = 10; // minimum seconds between each call
        debug.println("Demo");
    }
    lastCallTime = 5000 - ((callSpacing + 1) * 1000); // special calculation to immediately enable calls
}
digitalWrite(sleepPin, LOW);
delay(14); // 14 ms delay allows XBee to wake up from sleep mode
setAPIMode(); // puts the XBee into API packetized mode
setupXBee(); // initialize the proper settings on the XBee radio module
digitalWrite(sleepPin, HIGH);
dump.print(memoryTest());
dump.println("b free");
}

void loop () {
if (demoMode == false) {
   // dataSend(); // send moisture data to mySQL database
}

byte watered = wateringEventCheck(); // returns a value dependent upon watering events
byte moist = moistureCheck(); // returns a value dependent upon moisture level
byte light = lightCheck(lightMinimum); // returns a value dependent upon light level
if (millis() - lastCallTime > (callSpacing+1) * 1000 && calls < callLimit) {
   //if we're not too close to the last call and we haven't made too many calls...
   if (watered > lastWatered) { //if there's been a positive change
      byte ctr = 0;
      dump.println("WCall");
      // place a call request, using the value returned by watering event check
      while (httpRequest("128.122.151.44","80",watered) == false & ctr < 3) {
         ctr++;
      }
      lastWatered = watered; //log the current moisture value so you can compare it to the next value that comes in
      calls++; //add one to the count of the number of phone call attempts made
      lastCallTime = millis();
   }
   else if (moist > lastMoist) { //if there's been a positive change
      byte ctr = 0;
      dump.println("MCall");
      while (httpRequest("128.122.151.44","80",moist) == false & ctr < 3) {
         ctr++;
      }
      lastMoist = moist; //log the current moisture value so you can compare it to the next value that comes in
      calls++; //add one to the count of the number of phone calls made
      lastCallTime = millis();
   }
   else if (light != lastLight) { //if there has been a change
      byte ctr = 0;
      dump.println("LCall");
      if (sociable == true & light < lastLight) { // if the sociable objects mode is enabled and light is lacking
         neglected = checkSunny();
         if (neglected == true) {
            while (httpRequest("128.122.151.44","80",light) == false & ctr < 3) {
               ctr++;
            }
         }
      }
      else if (sociable == false || neglected == true) { // if the sociable objects mode is disabled or if complaint was made
         while (httpRequest("128.122.151.44","80",light) == false & ctr < 3) {
            ctr++;
         }
      }
      lastLight = light;
}
calls++;//add one to the count of the number of phone calls made
lastCallTime = millis();
}
}

lastWatered = watered;//log the current watering value so you can compare it to the next value that comes in
lastMoist = moist;//log the current moisture value so you can compare it to the next value that comes in
lastLight = light;//log the light value so you can compare it to the next value that comes in

switch (moist) {
    case 2: // min moist
        blinkLED(moistLED, 1, 200);
        break;
    case 6: // max dry
        blinkLED(moistLED, 1, 50);
        break;
    default:
        int brightness = analogRead(moistPin);
        analogWrite(moistLED, brightness); // otherwise display a steady LED with brightness mapped to moisture
        break;
    }

    if (sociable == true) { // if the sociable objects mode is enabled
        digitalWrite(sleepPin, LOW);
        respondToRequests();
    }
}

// function for checking if a watering event has occurred
byte wateringEventCheck() {
    // compare moisture now to moisture one minute ago
    // if level has changed by +x amount, then watering has occurred
    // byte returnValue = 0;
    static unsigned long lastMeasure; // time we last measured the watering event moisture variable
    static int wateredValue = 1023; // initialize the watering event value with saturation, to prevent an initial false positive
    int lastWateredValue = wateredValue; // update a variable with the last watering event moisture value taken
    if((millis() - lastMeasure) / 1000 > wateredTimeInterval) {
        blinkLED(ledPin, 2, 25); // led shows that project is still running
        wateredValue = analogRead(moistPin); // read the current watering event moisture value
        lastMeasure = millis();
        debug.print("W: ");
        debug.println(wateredValue,DEC);
        debug.print("L: ");
        debug.println(lastWateredValue,DEC);
        if (wateredValue > lastWateredValue + wateringCriteria) {
            if (lastWateredValue > minMoist[plantID]) {
                returnValue = 5; // you watered me when I didn't need it
            } else if (wateredValue < desiredSoakLevel[plantID]) {
                returnValue = 4; // I wasn't watered enough
            } else {
                returnValue = 3; // thanks for watering me
            }
        }
    }
}
else {
    returnValue = 0; // otherwise no watering event was detected during this function call
}
return returnValue;
}

//function for checking soil moisture against threshold
//function for moving average with soil moisture--adapted from the light code
byte moistureCheck() {
    static byte returnValue = 0;
    static int counter = 1; //init static counter
    static unsigned long lastMeasure; // time we last measured the soil moisture variable
    int moistAverage = 0; // init soil moisture average
    if((millis() - lastMeasure) / 1000 > (moistTimeInterval / moistNumSamples)) {
        for(int i = moistArraySize - 1; i > 0; i--) {
            moistValues[i] = moistValues[i-1]; //move the first measurement to be the second one, and so forth
        }
        moistValues[0] = analogRead(moistPin); //take a measurement and put it in the first place
        lastMeasure = millis();
        int moistTotal = 0; // create a little local int for an average of the moistValues array
        for(int i = 0; i < moistArraySize; i++) { //average the measurements (but not the nulls)
            moistTotal = moistTotal + moistValues[i]; //in order to make the average we need to add them first
        }
        if(counter<moistArraySize) {
            moistAverage = moistTotal/counter;
            counter++; //this will add to the counter each time we've gone through the function
        }
        else {
            moistAverage = moistTotal/moistArraySize; //here we are taking the total of the current light
            readings and finding the average by dividing by the array size
        }
        //lastMeasure = millis();
        debug.print("m: ");
        debug.println(moistAverage,DEC);
    }
    //return values
    if (moistAverage < maxDry[plantID]) {
        returnValue = 6; //we are in an urgent state of maximum dryness. please water this instant!
    } else if (moistAverage <= minMoist[plantID]) { //we are dry, make a phone call that requests to be watered
        returnValue = 2;
    } else { 
        returnValue = 0;
    }
    return returnValue;
}

//function for checking light level WE NEED TO ADJUST THE 1 AND 0 SETTING FOR EACH PLANT BASED ON REAL
//LIGHT TESTING
byte lightCheck(int desiredAverage) {
    static byte returnValue = 9;
    static int counter = 1; //init static counter
    static unsigned long lastMeasure; // time we last measured the light variable
    // reset lastMeasure if clock register overflows http://www.arduino.cc/en/Reference/Millis
    if(lastMeasure > millis()) lastMeasure = millis();
    int lightAverage = 0; // init light average
    if((millis() - lastMeasure) / 1000 > (lightTimeInterval / lightNumSamples)) {
        for(int i = lightArraySize - 1; i > 0; i--) {
            lightValues[i] = lightValues[i-1]; //move the first measurement to be the second one, and so forth
        }
        lightValues[0] = analogRead(lightPin); //take a measurement and put it in the first place
        lastMeasure = millis();
        int lightTotal = 0; // create a little local int for an average of the lightValues array
        for(int i = 0; i < lightArraySize; i++) { //average the measurements (but not the nulls)
            lightTotal = lightTotal + lightValues[i]; //in order to make the average we need to add them first
        }
        if(counter<lightArraySize) {
            lightAverage = lightTotal/counter;
            counter++; //this will add to the counter each time we've gone through the function
        }
        else {
            lightAverage = lightTotal/lightArraySize; //here we are taking the total of the current light
            readings and finding the average by dividing by the array size
        }
        //lastMeasure = millis();
        debug.print("l: ");
        debug.println(lightAverage,DEC);
    }
    //return values
    if (lightAverage < maxLight[plantID]) {
        returnValue = 6; //we are in an urgent state of maximum lightness. please get sun this instant!
    } else if (lightAverage <= minLight[plantID]) { //we are dry, make a phone call that requests to be watered
        returnValue = 2;
    } else {
        returnValue = 0;
    }
    return returnValue;
}
lightValues[0] = analogRead(lightPin); // take a measurement and put it in the first place
lastMeasure = millis();
int lightTotal = 0; // create a little local int for an average of the lightValues array
for(int i = 0; i < lightArraySize; i++) {// average the measurements (but not the nulls)
lightTotal = lightTotal + lightValues[i]; // in order to make the average we need to add them first
}
if(counter<lightArraySize) {
lightAverage = lightTotal/counter;
counter++;// this will add to the counter each time we've gone through the function
}
else {
lightAverage = lightTotal/lightArraySize;// here we are taking the total of the current light readings and finding the average by dividing by the array size
}
lastMeasure = millis();
dump.print("l: ");
dump.println(lightAverage,DEC);
if (lightAverage < desiredAverage) { // average minus padding is too low
    returnValue = 7;
} else if (lightAverage > desiredAverage) { // average plus padding is too high
    returnValue = 9;
}
return returnValue;
// now we've created an average light read

boolean deviceConnect(char* ipAddress, char* port) {
digitalWrite(sleepPin,LOW);
delay(14); // 14 ms delay allows XBee to wake up from sleep mode
boolean success = false;
byte ctr=0;
while (success == false && ctr < 3) { // try to make a XPort connection upload 20 times
    Serial.flush();
    sendData("C", XPortAddr); // instructs the XPort that a Connect message is coming
    sendData(ipAddress, XPortAddr); // IP address of the server to connect to
    sendData("/", XPortAddr); // slash to indicate port number is coming
    sendData(port, XPortAddr);// TCP/IP port number to connect to
    sendData("\n", XPortAddr); // end of line to finish the request
    if (checkFor("C", 3000)) { // if a connect response was received from the XPort then continue
        debug.println("true");
        success = true;
    } else {
        debug.println("false");
        success = false; // otherwise go back and try setup again
    }
    ctr++;
}
return success;
}
/*
dataSend() {
    static unsigned long lastSend=0; // time we last measured the watering event moisture variable
    // jitter the interval by plant ID so that plants don't all try to connect at exactly the same time
    if(millis() - lastSend) / 1000 > dataInterval+plantID) {
        if (deviceConnect("128.122.253.189","80") == true) {
            httpDataSend();
            lastSend = millis();
        }
void httpDataSend() {
    digitalWrite(sleepPin, LOW);
    delay(14); // 14 ms delay allows XBee to wake up from sleep mode
    boolean success = false;
    byte ctr = 0;
    while (success == false && ctr < 6) { // try to make a database upload six times
        // Make HTTP GET request
        Serial.flush(); // clear the serial port for new incoming data
        Serial.print("GET /~kl892/bcalls.php?a=sensordata&id=");
        if (plantID < 100) Serial.print("0");
        if (plantID < 10) Serial.print("0");
        Serial.print(plantID, DEC);
        Serial.print("&t=M&v=");
        Serial.print(analogRead(moistPin), DEC);
        Serial.print(" HTTP/1.1");
        Serial.print("\n");
        //Serial.print("HOST:itp.nyu.edu\n\n"); // server's hostname
        if (checkFor("SUCC",3000)) { // check for the "ok" response from the server
            debug.println("dbOK");
            success = true;
        } else {
            debug.println("dbBad");
            success = false;
            // If we are in Monitor Mode (and we don't know why this happens) then attempt to quit out of it
            // Serial.flush(); // clear the serial port for incoming data
            // Serial.print("\n"); // send some line feeds
            // if (checkFor("9>", 5000)) { // see if we get an error prompt from Monitor Mode
            //    Serial.print("QU");  // if we sense Monitor Mode then send the quit command
            //    debug.println("TryQuit");
            // }
            ctr++; //
        }
    }
    delay(5000); // wait for XBee's RSSI light to go out, otherwise it stays on and uses power continuously
    digitalWrite(sleepPin, HIGH);
}
*

// +++++++++++
// Send an HTTP GET request
boolean httpRequest(char* ipAddress, char* port, int messageType) {
    boolean success = false;
    digitalWrite(sleepPin, LOW);
    delay(14); // 14 ms delay allows XBee to wake up from sleep mode
    connectIPGate();
    if (deviceConnect(ipAddress, port) == true) {
        // char phone [] = "12129896888"; // Hartman: 19178417494, London: 19174342135, Faludi 12129896888,
        Lounge: 12129981877
        // LearningWorlds: 16464424409
        Serial.flush(); // clear serial before the next request
        sendData("GET /~kh928/bcall.php?af=c", XPortAddr);
        // Serial.print(phone);
        // Serial.print("&audiofile=calltype");
        char messageString[2];
        itoa(messageType, messageString, 10);
        sendData(messageString, XPortAddr);
        sendData("_id", XPortAddr);
        if (plantID < 100) sendData("0", XPortAddr);
        if (plantID < 10) sendData("0", XPortAddr);
char plantString[2];
itoa(plantID, plantString, 10);
sendData(plantString, XPortAddr);
// sendData(" HTTP/1.1", XPortAddr);
sendData("\n", XPortAddr);
// sendData("HOST:asterisk.itp.tsoa.nyu.edu\n\n", XPortAddr); // server's hostname
if (checkFor("Faludi",3000)) { // check for the "ok" message in the server's response
    debug.println("CallOK");
    success = true;
    Serial.flush(); // clear the serial port of extraneous packets
} else {
    debug.println("CallBad");
    success = false;
    quitMonitorMode();
}
else {
    debug.println("ConBad");
    success = false;
    quitMonitorMode();
}
setupXBee();
delay(5000); // wait for XBee's RSSI light to go out, otherwise it stays on and uses power continuously
digitalWrite(sleepPin,HIGH);
return success;
}

void quitMonitorMode() {
    // If we are in Monitor Mode then attempt to quit out of it
    Serial.flush(); //++++++++++++++++
    sendData("\n", XPortAddr); // send a line feed
    if (checkFor("9>", 500)) { // see if we get back an error prompt from Monitor Mode
        sendData("RS\n", XPortAddr); // if we sense Monitor Mode then send the quit command
        debug.println("TryRset");
    }
}

void connectIPGate() {
    boolean outcome = false;
    while (outcome == false) {
        outcome = sendCommand("ID", 0x7777);
    }
    outcome = false;
    while (outcome == false) {
        outcome = sendCommand("MY", plantID);
    }
}

void setupXBee() {
    // send commands using the sendCommand() function
    boolean outcome = false;
    while (outcome == false) {
        outcome = sendCommand("RE", 0x0); // The RE command resets the XBee to factory defaults
        // it ignores any parameters, but the current API requires one so we send a zero which is ignored
    }
    while (outcome == false) {
        outcome = sendCommand("ID", 0x7777);
    }
    outcome = false;
    while (outcome == false) {
        outcome = sendCommand("MY", plantID);
    }
}
false;
while (outcome == false) {
    outcome = sendCommand("DH", 0x0);
}  
outcome = false;
while (outcome == false) {
    outcome = sendCommand("DL", 0x64);
}  
outcome = false;
while (outcome == false) {
    outcome = sendCommand("SM", 0x1);
}  

boolean checkSunny() {
    // this function checks with neighboring plants to see if they are satisfied with their light levels
    // if neighbors are satisfied, but you are not, then it's appropriate to make a complaint call
    int Y=0,N=0; // yes and no response counters for satisfaction with light levels
    int timeout = 500; // timeout for checking each neighbor
    boolean result = true; // assume that it's sunny to begin with
    // ask each neighbor if they are satisfied with their light levels
    for (int i=0; i < 16; i++) {
        digitalWrite(sleepPin, LOW);
        delay(14);
        sendData("L", i); // send out an Light level data request
        char* incomingResponse;
        incomingResponse = getData(incomingResponse, timeout);
        if (incomingResponse != NULL) { // if we didn't read a bad packet
            if (strstr(incomingResponse,"Y") != NULL ) { // if the desired string is found
                Y++;
            } else if (strstr(incomingResponse,"N") != NULL ) { // if the desired string is found
                N++;
            } else {
                debug.println("bad response");
            }
        } else {
            debug.println("bad packet");
        }
        free(incomingResponse); // free the string's allocated memory for reuse
    }  
    // evaluate responses
    if (Y >= N) {
        result = true; // if more happy than not, or no responses, assume that it's sunny
    } else {
        result = false; // otherwise it seems to be cloudy
    }
    delay(5000); // wait for XBee's RSSI light to go out, otherwise it stays on and uses power continuously
    digitalWrite(sleepPin, HIGH);
    return result;
}

void respondToRequests() {
    if (Serial.available() > 0) {
        char* incomingResponse;
        incomingResponse = getData(incomingResponse, 1000);
        if (incomingResponse != NULL) { // if we didn't read a bad packet
            if (strstr(incomingResponse,"L") != NULL ) { // if this is a light level request
                debug.print("Lyes"); // indicate this was a light request
            } else {
                debug.println("bad packet");
            }
        } else {
            debug.println("bad packet");
        }
    }
}
byte light = lightCheck(lightMinimum); // returns a value dependent upon light level
if (light == 9) { // if we're happy with our light level
    sendData("Y", lastSender);
} else if (light == 7) {
    sendData("N", lastSender);
} else {
    debug.print("Lno"); // indicate this wasn't a light request
}
free(incomingResponse); // free the string's allocated memory for reuse

void timeCheck() {
    Serial.flush(); // clear the serial buffer before reading new data
    char timeString[15]; // create a string to hold the time value when it's read
    memset(timeString, '\0', 15); // initialize that string to all NULL characters
    byte inByte = '\0'; // declare and initialize a byte to read in serial data
    while(inByte != '*') {
        inByte = Serial.read(); // read data and wait for an asterisk character
    }
    boolean timeStringValid = true; // declare and initialize a variable to track whether the string has
    all valid characters
    while(Serial.available() < 14) {
        ; //wait for enough data to be available (14 characters of time string), while doing nothing else
    }
    for (int i=0; i < 14; i++) {
        timeString[i] = Serial.read(); // reach each time string character into a character array
        if(timeString[i] < '0' || timeString[i] > '9') {
            timeStringValid = false; // if any character is bad then the whole string is bad
        }
    }
    if (timeStringValid == true) {
        char yearString[5]; // create a string to hold the year part of the string
        memset(yearString, '\0', 5); // initialize that string to all NULL characters
        strncpy(yearString, timeString, 4); // copy the first four characters of timeString into the year
        time.year = atoi(yearString); // convert ASCII year string to integer and store in the year integer
        variable
        char monthString[3]; // create a string to hold the month part of the string
        memset(monthString, '\0', 3); // initialize that string to all NULL characters
        strncpy(monthString, timeString+4, 2); // skip four characters, then copy the next two of timeString
        into the month string
        time.month = atoi(monthString); // convert ASCII month string to integer and store in the month
        integer variable
        char dayString[3];
        memset(dayString, '\0', 3);
        strncpy(dayString, timeString+6, 2);
        time.day = atoi(dayString);
        char hourString[3];
        memset(hourString, '\0', 3);
        strncpy(hourString, timeString+8, 2);
        time.hour = atoi(hourString);
        char minuteString[3];
        memset(minuteString, '\0', 3);
        strncpy(minuteString, timeString+10, 2);
        time.minute = atoi(minuteString);
char secondString[3];
memset(secondString, '0', 3);
strncpy(secondString, timeString+12, 2);
time.second = atoi(secondString);
}
}

UTILITY FUNCTIONS

// this function checks for a specific response on the serial port
// it accepts a string to look for and a timeout in milliseconds
boolean checkFor(char* desiredResponse, int timeout) {
  boolean result = false;
  char* incomingResponse;
incomingResponse = getData(incomingResponse, timeout);
if (incomingResponse != NULL) {
  if (strstr(incomingResponse, desiredResponse) != NULL) {
    result = true;
  } else {
    result = false; // return false if the strings don't match
  }
} else {
  result = false; // return false if we got a bad packet
}
free(incomingResponse); // free the string's allocated memory for reuse
return result;
}

int memoryTest() {
  int byteCounter = 0; // initialize a counter
  byte* byteArray; // create a pointer to a byte array
  // More on pointers here: http://en.wikipedia.org/wiki/Pointer#C_pointers
  // use the malloc function to repeatedly attempt allocating a certain number of bytes to memory
  while ((byteArray = (byte*) malloc (byteCounter * sizeof(byte))) != NULL) {
    byteCounter++; // if allocation was successful, then up the count for the next try
  }
free(byteArray); // also free memory after the function finishes
return byteCounter; // send back the highest number of bytes successfully allocated
}

void blinkLED(byte targetPin, int numBlinks, int blinkRate) {
  for (int i=0; i<numBlinks; i++) {
    digitalWrite(targetPin, HIGH); // sets the LED on
    delay(blinkRate); // waits for a blinkRate milliseconds
    digitalWrite(targetPin, LOW); // sets the LED off
    delay(blinkRate);
  }
}

FUNCTIONS

// this function puts together a command packet
boolean sendCommand(char* command, unsigned long data) {
  static byte frameID = 0;
  static byte* packet = (byte*) malloc(64 * sizeof(byte));
  packet[0] = frameID;
  // More on the packet here: http://example.com
  // use the serialSend function to transmit the packet
  // More on serialSend here: http://example.com
  free(packet); // free the allocated memory for reuse
  return true;
}
frameID++; // add one to the frame ID each time, okay to overflow
if (frameID == 0) frameID = 1; // skip zero because this disables status response packets
byte checksum = 0xFF; // checksums are the hex FF minus the total of all bytes
// calculate length of the packet
int dataLength = countBytes(data);
int length = (dataLength + 4); // data length + API id + frame id + two command bytes
// send the packet
Serial.print(0x7E, BYTE); // send start delimiter
Serial.print(0x0, BYTE);  // send length MSB (always zero because packets can't be more than 100 bytes)
Serial.print(length, BYTE); // send length LSB
Serial.print(0x8, BYTE); // send API command identifier
checksum = checksum - 0x8;
Serial.print(frameID, BYTE); // send frame ID (set to 0 if no response is required)
checksum = checksum - frameID;
Serial.print(command);   // send two-character AT command
for (int i=0; i < strlen(command); i++) {
  checksum = checksum - command[i]; // add in the AT command bytes
}
// DIVIDE DATA INTO BYTES AND ITERATE THROUGH EACH ONE
for (int i = dataLength; i > 0; i--) {
  byte dataByte = data >> 8 * (i-1); // shift over one byte at a time, MSB first
  Serial.print(dataByte, BYTE); // send command data
  checksum = checksum - dataByte;
}
Serial.print(checksum); // send checksum
return getCommandResults();

// function that receives the results of a command request
boolean getCommandResults() { // MAYBE ADD FRAME ID AS AN ARGUMENT HERE?
  int packetLength = checkHeader(500); // check for a start byte with a 1/2 second timeout
  // debug.print("packetLength: ");
  // debug.println(packetLength);
  if (getIdentifier(0x88)) { // if this is indeed the results of a command
    byte frameID = Serial.read(); // get the frame ID we're receiving information about
    // debug.print("frameID: ");
    // debug.println(frameID);
    // right now we're not doing anything with the frame ID, so this is a formality
    char commandReceived[3];
    // debug.print("commandReceived: ");
    for (int i=0; i < 2; i++) {
      commandReceived[i] = Serial.read(); // get the AT command we're receiving
      // debug.print(commandReceived[i]);
      // (right now we're not doing anything with the AT command info, so this is a formality)
    }
    // debug.println("" Untitle");
    boolean status = !Serial.read(); // OK is equal to zero in the API, so invert this value when
    reporting it
    byte checksum = Serial.read(); // read in the checksum. We ignore this for now
    return status;
  } else {
    return false; // if Identifier indicates wrong packet then give negative feedback
  }
}

// function to check for the Header bit, and read in the two following length bits
int checkHeader(int timeout) { // timeout is in milliseconds
  long startTime = millis();
  int length = 0;
  int inByte = 0;
  // during the timeout period, if we haven't gotten the start byte yet...
  while (((millis() - startTime) < timeout) && (inByte != 0x7E)) {
    if (Serial.available() > 0) { // if a byte is waiting in the buffer
// function that puts together a data packet and transmits it, using 16-bit addressing
boolean sendData(char* data, int destinationAddr) {
    static byte frameID = 0;
    frameID++; // add one to the frame ID each time, okay to overflow
    if (frameID == 0) frameID = 1; // skip zero because this disables status response packets
    byte checksum = 0xFF; // checksums are the hex FF minus the total of all bytes
    // calculate length of the packet
    int dataLength = strlen(data);
    int length = (dataLength + 5); // data length + API id + frame id + two address bytes + options byte
    // send the packet
    Serial.print(0x7E, BYTE); // send start delimiter
    Serial.print(0x0, BYTE);  // send length MSB (always zero because packets can't be more than 100 bytes)
    Serial.print(length, BYTE); // send length LSB
    Serial.print(0x1, BYTE); // send API command identifier
    checksum = checksum - 0x1;
    Serial.print(frameID, BYTE); // send frame ID (set to 0 if no response is required)
    checksum = checksum - frameID;
    for (int i = 2; i > 0; i--) {
        byte destinationByte = destinationAddr >> 8 * (i-1); // shift over one byte at a time, MSB first
        Serial.print(destinationByte, BYTE);  // send destination address
        checksum = checksum - destinationByte;
    }
    checksum = checksum - options; // add in the AT command bytes
    Serial.print(checksum); // send checksum
    return getTXstatus();
}

// function that receives data which was sent using 16-bit addressing
char* getData(char* dataIn, int timeout) {
    int packetLength = checkHeader(timeout); // check for a start byte with a timeout
    debug.print("pktLen: ");
    debug.println(packetLength, DEC);
    if (packetLength > 0 && getIdentifier(0x81)) { // if we didn't get an error from check header and this is indeed a data rx packet
        if((dataIn = (char*) malloc(packetLength - 5+1)) == NULL) { // attempt to allocate string memory
            debug.println("mallocNO"); // report if allocation fails
        } else {
            debug.println("mallocOK"); // otherwise report that allocation succeeded
        }
        memset(dataIn,0,packetLength); // initialize all incomingResponse string positions to null
        byte addrMSB = Serial.read(); // get the MSB of the source address
        byte addrLSB = Serial.read(); // get the LSB of the source address
        // debug.print("MSB: ");
        // debug.println(MSB);
        // debug.print("LSB: ");
        // debug.println(LSB);
        lastSender = (addrMSB << 8 ) + (addrLSB); // update global variable with this sender's address
        byte RSSI = Serial.read(); // get the Received Signal Strength Indicator value
        // debug.print("RSSI: ");
        // debug.println(RSSI);
        byte options = Serial.read(); // get the options. 0 = reserved, 1 = Addr broadcast, 2 = PAN broadcast, 3 - 7 = reserved
        // debug.print("options: ");
        // debug.println(options);
        long startTime = millis();
    }
    for (int i=0; i < (packetLength - 5); i++) {
        dataIn[i] = Serial.read(); // get a byte of data
    }
    return 0;
}
debug.print(dataIn[i]);
}
}

byte checksum = Serial.read();
// debug.print("checksum: ");
// debug.println(checksum);
return(dataIn);
}

else {
    return NULL; // if this packet is bad then give negative feedback
}
}

// puts the XBee in API mode
void setAPIMode() {
    delay(14); // 14 ms delay allows XBee to wake up from sleep mode
    boolean success = false;
    while(success == false) {
        // strobe led
        delay(1100);
        // put the XBee in command mode
        Serial.print("+++\n");
        delay(1100);
        // wait for a response from the XBee for 2000 ms, or start over if no valid response comes
        Serial.println("ATAP1\n");
        delay(1100);
        // select API Mode
        Serial.println("ATCN\n");
        // exit command mode (note that we use Serial.println here to issue a linefeed that completes the
        // command sequence)
        Serial.flush(); // remove any prior "OK" responses from the serial buffer
        Serial.println("ATCN");
        long startTime = millis();
        while (((millis() - startTime) < 500) && Serial.available() < 2) ; //wait for two characters to come
        if (Serial.read() == 'O') { // test for the first letter of "OK"
            success = true;
            Serial.flush();
        }
    }
}

// this function counts the number of bytes in a long
int countBytes (long myLong) {
    int length=0;
    do {
        myLong = myLong >> 8; // shift one byte over
        length++; // and add this byte to the count
    } while (myLong != 0); // as long as there's still a non-zero number remaining
    return length;
}