

Programmable Remote Wake-Up Alarm

Ian Mowers

ECE499 – Electrical/Computer Engineering Capstone Project 3

Supervisor: Cherrice Traver

3/17/2016

REPORT SUMMARY

In this project, we are aiming to solve the issue of loud alarm clocks, which can cause problems for people who share rooms with others. To design and implement a solution, we consider previous technology that could aid us, such as past alarm clocks and newer headsets. We also examine the issues of developing the solution, as well as component restrictions. Manufacturing the final product, health, and societal impact are prime concerns. A goal for the initial prototype is for it to be compact and wearable, thus we need to ensure that the user's health is not adversely affected. These issues and constraints determine our requirements for the final design. Options for implementing the final product are also examined and compared on a basis of functionality and cost. The final design is a simple two part system, which consists of a smartphone running Android OS that signals a hearing-aid style receiver to vibrate. The vibration from the receiver will be enough to wake the user whilst not waking any others. It is intended to work within the perimeters of a typical bedroom (approximately 20 feet per wall), and be comfortable such that the user can sleep undisturbed. During the prototyping stage, the receiver was built into two separate circuits using different types of microcontrollers: an Arduino-based and PIC-based circuit. The two prototypes were then tested with a developed smartphone application. Only the Arduino circuit worked, while the PIC microcontroller had programming difficulties. In this implementation, the user can set the alarm using their smartphone running Android with Bluetooth capabilities and the receiver will vibrate a small motor.

Table of Contents

Report Summary	2
Table of Contents	3
Index of Figures and Tables	4
Introduction	5
Background	7
Design Requirements	10
Design Alternatives	16
Preliminary Proposed Design	19
Final Design and Implementation	23
Performance Estimates And Results	32
Production Schedule	35
Cost Analysis	37
User's Manual	38
Discussion, Conclusions and Recommendations	40
References	42
Appendix	43

Index of Figures and Tables

Figure 1: Artists' depiction of alarm clock problem	5
Figure 2: Basic block diagram of solution to problem	6
Figure 3: Graphic overview of functional decomposition	15
Figure 4: JY-MCU HC-06 Bluetooth transceiver picture	21
Figure 5: Arduino UNO R3 picture	21
Figure 6: Illustration of basic receiver design	22
Figure 7: Image of Vibration Motor used in circuits	23
Figure 8: Schematic of implemented Arduino-based circuit	24
Figure 9: Concept image of PIC used in circuit	25
Figure 10: Schematic of implemented PIC-based circuit	27
Figure 11: Image of PIC circuit on performance board	27
Figure 12: TimePicker on screen	30
Figure 13: Bluetooth device list	30
Figure 14: Logic flow of smartphone application operation process	32
Figure 15: Main menu for smartphone application	39
Figure 16: Screen image of alarm time with system clock	39
Table 1: Defined Behaviors and Constraints	10
Table 2: Approaches for Signaling Device, Advantages and Disadvantages	11-12
Table 3: Approaches for Receiver, Constraint Comparison	13-14
Table 4: Final Decisions on Design Requirements	16
Table 5: Main Parts for Prototyping Stage	22
Table 6: Estimated and Actual Measurements of the Arduino Circuit	33
Table 7: Range Trials for Arduino Circuit	34
Table 8: Cost Comparison Between Implemented Circuits	37

INTRODUCTION

The impetus of this project stemmed from an issue that some people face in their everyday life: waking up to a loud alarm clock. At first this would seem to be a trivial issue, as some need such alarm clocks to wake up, but this is not always the case for people sharing bedrooms. Some people may use an alarm clock to wake up early in the morning, but get up before it goes off, fearing that they will wake those sleeping. Another example is sharing a room in a university dorm. Some college students could be heavy sleepers and need a loud alarm to be woken up. When roommates run on different schedules, early wake-up times can result in a situation where others are woken up earlier than needed or expected. These situations can be extended to roommates in apartments, couples sharing beds, and so on. Based upon personal experience, this issue is important enough that it should be solved.

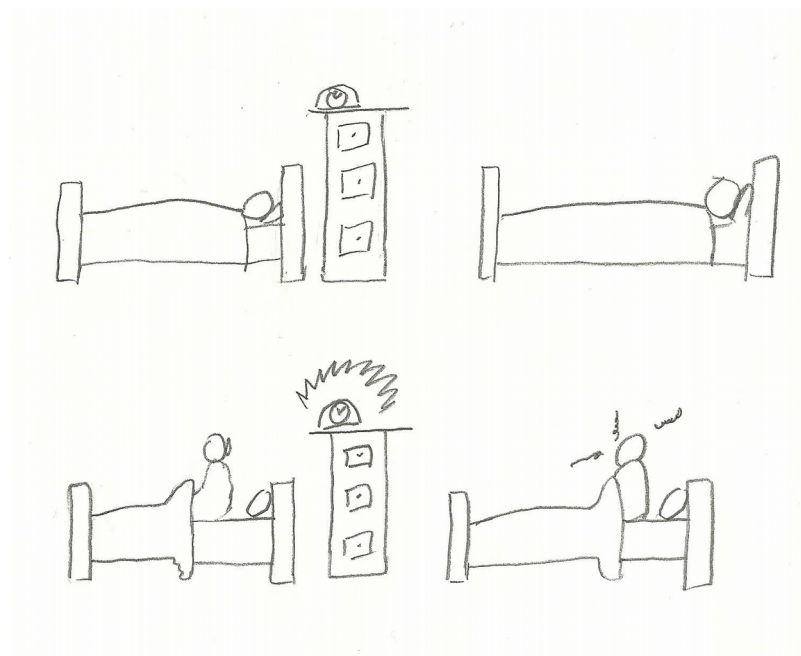


Figure 1: Artist's depiction of problem. Note the frustration in the other person .

To alleviate this problem, the goal of this project is to design a lightweight and portable receiver that will wake users in a quieter manner than before. The intended structure of the

receiver in the final product is based off of hearing-aids, but will incorporate other materials and components to ensure comfort and feasibility of uninterrupted sleep. A separate signaling device – in the case of this project, a smartphone – will signal to the receiver using ubiquitous communication technologies available, while the system works within a typical size bedroom. It will also have an interface that allows the user to see the time, be able to set the alarm, and view any other important information regarding the system process. Basically, the device takes an input, the alarm time, then outputs a signal to the receiver on the user's ear. The receiver will receive a signal from a remote device, activate, and then wake the user.

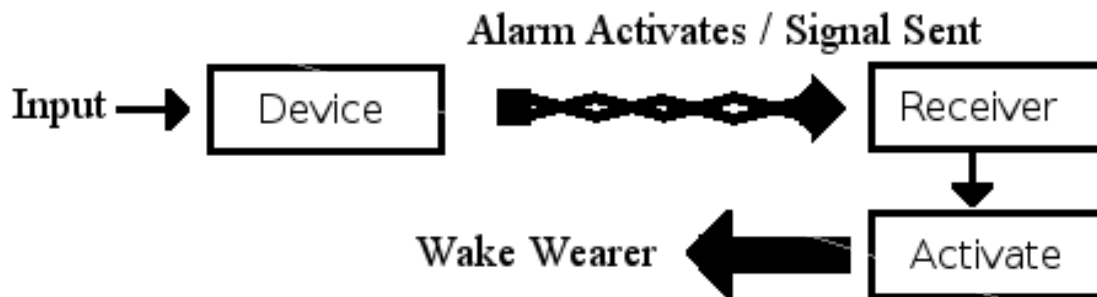


Figure 2: Basic and simple diagram of problem solution

This report is divided into several sections, ranging from Background to the Preliminary Design, that intend to clarify and further define the design of this project. The first section, the Background, will introduce the inventions and devices related to the problem and goal of this project, while also detailing the major issues concerning implementation, especially ethical issues and similar solutions currently in development. This will be followed by the Design Requirements, laying out the specifications and criteria for the project solution, including functions. Design Alternatives considers the possible options for implementing the project functions, supported by research conducted to examine various constraints. All of the requirements, reasoning and research culminated in the Preliminary Proposed Design, which

was further tweaked and implemented as described in the Final Design and Implementation section. The testing stage for the implementation and its results are presented in Performance Estimates and Results, followed by the Production Schedule that describes the timeline of developing and producing the problem solution. Cost Analysis then tabulates how much the implementation would cost. The User's Manual section will then explain how a potential user would operate the system. The report is finally concluded by the Discussion, Conclusion and Recommendations, which will discuss the successes and failures of the project and suggest directions in which to fix and develop the project in the future.

BACKGROUND

The earliest concept of alarm clocks probably stemmed from old large clocks, either in houses or on the public square, that would ring bells or make other noises to indicate a new hour in the day. This eventually evolved into smaller, personal clocks, where users could manually set the alarm time by moving the hands to indicate time. Alarm clocks in the present day are mostly digital, but this technology still relies on a physical clock within the device, as well as an alarm where the user has little control over the sound volume. Common digital alarm clocks have at most one or two buzzer settings, compared to white noise machines which have many more settings. An inherent problem with cheaper digital clocks is that after extensive use while being powered on, they begin to slowly gain time. One example is a digital clock used by the project author, after about 2-3 months the clock has gained about 4 minutes. This issue is easy to fix by resetting the clock, but these alarm clocks should be quiet and consistent. Utilizing smartphones could be the most precise measurement of time, and thus behave as our clock for the project.

Since the advent of cellphones and the Bluetooth standard [1], various "hands-free" devices have been developed and released on the market, functioning as mobile headsets without

the wires. Bluetooth also extends into data distribution, when pushing data from one device onto another. This project involves a wireless receiver using both of these Bluetooth features, although other alternatives are explored, such as RFID. Initial inspiration for the project concept also came from hearing-aid design, which has existed for several decades. The conceptual aspects of both the on-ear hearing-aid and remote hands-free headset combine to form top-level design of the wireless receiver.

Devices similar to the design of this project are currently limited in number, though approximately a year or so before the writing of this report, Kyungmi Moon in South Korea designed something similar currently in development, the “Earlarm”[2]. The intended product is a three-sectional system, where one sets the alarm on a separate device with their smartphone, which then sends a signal to a pair of ear-plugs that will wake user. While similar in design to this project, the focus is more on encapsulating the activation process within just the receiver and smartphone, without functioning as an earplug like the “Earlarm”.

There are three major issues that will need to be addressed when determining the constraints and designing the project: manufacturing, health concerns, and the social impact. One of the goals of this project is compactness, but this may not be readily available during the design and prototyping process. Prototyping with regular and familiar circuit components would definitely ease the process, these components are not small. We also cannot just modify hearing-aids and current wireless headsets since they are rigid structures and would be considered product tampering. However, we can be confident that this project will come to fruition in the future using those products as a conceptual foundation. During the prototyping stage we want to ensure that the concept works, and once we confirm that it does, we can then focus on reducing the overall size.

Another issue is safety regarding the use of the receiver. Since this will be a device that

the user puts on their body and has close contact with the ear, health is a big concern. Using both wireless transmission standards and circuitry leads to concern over electrical shock and radiation from signals. Any problems that could affect the user's ear, such as blocking the ear canal or physical damage, will be a factor in determining whether the receiver needs to be in the ear or outside the ear, and how the device will be powered. We also want the receiver to be small, so it must be low-powered.

This project may have a social impact if it can be manufactured *en masse* and has a market release, but the impact would not be large. A small benefit to the use of the project would be that end users would have easier times waking up, and maybe not affect others' sleep. One detrimental aspect is that the final product might not work for those who need loud noises to wake them up, but this could be field tested during the prototyping process.

In terms of ethical issues, the intent of this project is to replace a tool with a better tool, replacing alarm clocks, only to wake people. When thinking of a problem to solve, another goal was to ensure that the final product would benefit as many lives as possible, and could have other applications beyond the scope of this project. Since it will use Bluetooth or other wireless standards on smartphone, the receiver may not be an entirely secure device and would pose an issue if hacked. It would appear that the effects of this project does not result in any serious ethical issues towards society, a positive aspect. Bluetooth and wireless standards can be used in many negative ways, e.g. stealing of sensitive information as well as disrupting communication between important systems.

All of the major issues concerning the project will be further addressed when we discuss the constraints in the Design Requirements section. Overall, the main idea behind the project arose from old alarm clocks, the noise and time-keeping problems with current ones, and combining the technology with hearing-aids and wireless headsets. There may be problems with

manufacturing the device, as well as health concerns, but we will discuss these further in the next section.

DESIGN REQUIREMENTS

The project's behavior initially was split amongst the signaling device and the receiver as seen in Table 1, but as the design process continued the behavior of communication became a separate entity while some desired behaviors were no longer a goal. Some implementation choices were also considered when determining the final design criteria.

Table 1: Defined Behaviors and Constraints

Part	Behavior	Constraints To Consider
Signal Device	Signaling	Signal Standards, Availability
	Time Tracking	Designing a separate time system, Cost
	Power	Need to design power source, Cost
	Settings	Physical design of device, Circuit design
Receiver	Receiving	Linking with signal device, Health concerns
	Vibration	Circuit design, Power
	Power	Sourcing power, Cost
	Recharge-ability*	Cost, Effectiveness, Longevity
	Power Indication*	Circuit design, physical design
	Physical Design*	Health concerns*, Power, Recharge-ability*

The intended behavior of the system is spread out into two parts, a signaling device that can transmit signals and the receiver that receives these signals. Although both parts have some shared behavioral aspects, the receiver was initially planned to be a rechargeable device that stays on or in the ear while you sleep, while the alarm would be set with the signaling device. When the alarm goes off, a signal would be sent from the signal device to the ear attachment. In this first design, one desired feature was to have the entire receiver itself vibrate, but other ways to use vibration were examined and decided upon later. The alarm should shut off automatically

to preserve power and ease useability. Recharge-ability, another beneficial feature, would have depended on the power requirements for the ear device, with the power dependent on the signaling method. As the focus of this project shifted to ensuring the concept was feasible, recharge-ability was demoted to being a periphery feature and would most likely not be implemented in the prototype.

Potential end users or customers for this system are people who share bedrooms that wish to have their wake-up routine not wake others. End-users in this kind of situation are most likely to be those who share apartments, sleep in the same bed, or college roommates. Other potential users could be those who simply would prefer a quieter, less abrupt way to wake. The end result will ensure that only those who use the device will wake up.

Earlier in the design process, there were a few design approaches examined and some choices researched in order to determine some of the design criteria. Their advantages and disadvantages were compared to choose final criteria, resulting in some design approaches and several choices to be discarded. These are all presented in Table 2, comparing between all possible approaches considered for each behavior. Choices at this stage were then further researched upon as options along with newly considered ones, the process detailed in the Design Alternatives section.

Table 2: Approaches for Signaling Device, Advantages and Disadvantages

Behavior	Design Approach	Choices	Advantages	Disadvantages
Signaling To Alarm Device	Smartphone App	RFID / Other	Direct from phone	Range, Standards
		Phone Signal	Use phone's signal directly	Unsure about protocols, Legality?
	Custom-Made	Design Transmitter	System specifically for receiver	Learn how to build it, complex circuit, standards
Time Tracking	Smartphone App	Phone's Clock	Atomic/Satellite	Access to clock difficult?

		Alarm Clock App	Existing software	Differing OS, Redundant
	Custom-Made	Physical System	-	Learn how to build accurate clocks, power issue
Power	Smartphone App	Phone Power	No separate power	Relies on phone
	Custom-Made	Built-in	Own power supply	Need to construct separate source, time consuming
Settings	Smartphone App	-	Construct software to maintain settings, easy to add new features	Need to function properly
	Custom-Made	-	Simple button based interface	Very limited and fixed design, choices will affect outer shell
Overall	Smartphone App	-	Smartphones nearly ubiquitous	Need to learn programming for OS, beholden to phone power, range, security
	Custom-Made	-	Separate controllable, portable device	Construction, Maintenance, Precision

For the signaling device, two approaches were explored for implementing behavior, as laid out and compared in Table 2. These were to either create a smartphone application, using the smartphone as the power and transmitter, or design and construct a custom-made receiver that would keep track of time.

For communication, there were two choices considered on the smartphone: utilizing RFID or other wireless capabilities; or using the phone signal, such as the GSM/CSMA bands. Instead using the smartphone hardware, a custom-built transmitter would select certain band frequencies, and could work specifically with a custom-built receiver. Considering the overall advantages and disadvantages, a currently existing RFID/Wireless system is the simplest, most cost-effective and easiest choice to implement out of the two, but it could have range issues and

require understanding of specific transmission protocols. Time tracking would also constrain the overall design. With a smartphone, we can use the phone's own clock, which is synchronized to satellites and other atomic clocks. This is the most precise method of keeping track of time, compared to a custom-made clock system that requires understanding construction of digital clocks, which are prone to error to those unskilled. Power for transmission comes from the smartphone itself, while a separate system would require a transformer to draw power from electrical outlets. All the settings would also be controlled by software on a smartphone, thus giving the ability to add more features easily. Putting the settings on the separate, possibly molded transmission device would be affected by the circuit design, as well as limiting addition of future features.

Overall, smartphones have become ubiquitous at the time of this project's development, but creating a software application requires some programming on smartphone operating systems and features are restricted to the phone's specifications. For a custom-made transmitter, a separate device could be built that would then have its own specific capabilities and features, but designing all the specifications and ensuring precision would also be required – a strenuous process.

Table 3: Approaches for Receiver, Constraint Comparison

Behavior	Design Style	Implementation	Constraints
Receiving Signal	Hearing-Aid	Placed on bottom	Size of receiver, Health hazards
	Mesh Earplug	Receiver on end of plug, like a chip	Health hazards
Vibration	Hearing-Aid	Vibrator piece in ear	Comfort, Health
	Mesh Earplug	Whole plug vibrates	Design, Comfort, Health
Power	Hearing-Aid	Button Batteries	Battery types, Cost
	Mesh Earplug	Button Batteries	Battery types, Design, Cost

Recharge-ability (Replacement)*	Hearing-Aid	Compartment	Inserting and removing batteries
	Mesh Earplug	Compartment	Same as above, more problematic
Power Indicator*	Hearing-Aid	LED Light	Design, Manufacturing
	Mesh Earplug	LED Light	Design, Comfort, Manufacturing
Physical Design	Hearing-Aid	Around Ear	Health, Comfort
	Mesh Earplug	In-Ear Mesh	Health, Comfort, Functionality

Two design styles were considered for the receiver: a hearing-aid based design and a mesh earplug. Aside from circuit housing and the location of vibration, most of the implementation choices were the same: power limited to button batteries and indication being relegated to small LED lights. Since the receiver will be an ear device, a lot of decision-making was influenced by health, design, and comfort.

Cost was primarily affected by power constraints. On Amazon (an online market that can sell products from many third-party suppliers) in the spring of 2015, a set of 60 single-use batteries for hearing-aids were about \$15-25, while the lifespan was less than a week. On the other hand, rechargeables were much more expensive, where a two-pack of typical rechargeable hearing-aid size batteries were priced the same as the 60-pack for single-use. The one advantage with the rechargeables was that the lifespan on average is about a year, the average charge cycle typically 12-18 hours, depending on size. These types of batteries require a charger, maybe one using magnetism similar to recharging stations for hearing-aid batteries.

The primary health concern regarding the receiver is where the parts of the device would be placed. Putting the vibrator inside the ear, similar to the in-ear portion of a hearing aid, will cause wax buildup in the ear over time and lead to ear infections. Instead, placing the vibrator on the outside would vibrate the skull. This method also works as the human skull can pick up

vibrations that then travel to the eardrum. Utilizing this physiological feature allows the user to leave their ear canal open, avoiding infections and having to frequently clean the device.

Now that all the constraints are determined and we are close to picking certain design approaches, the project behavior can be rearranged into functions. The functions can be grouped into three main sections: the transmitter, the communication, and the receiver. The transmitter, implemented through a smartphone, will contain the software application and keep track of time, functioning as the alarm settings. Communication through a one-directional system, where either the smartphone sends signals to the receiver. Doing this simplified the behavior of our receiver and eases any possible programming that would be required. The receiver has the most functions: receiving signal, power control (such as a sleep mode), enable (turning the device on and off), and vibration. The standard for receiving the signal will be the same as the communication, while power and enabling are needed to activate the vibration. Overall, this decomposition of functions will work in tandem with our final design criteria.

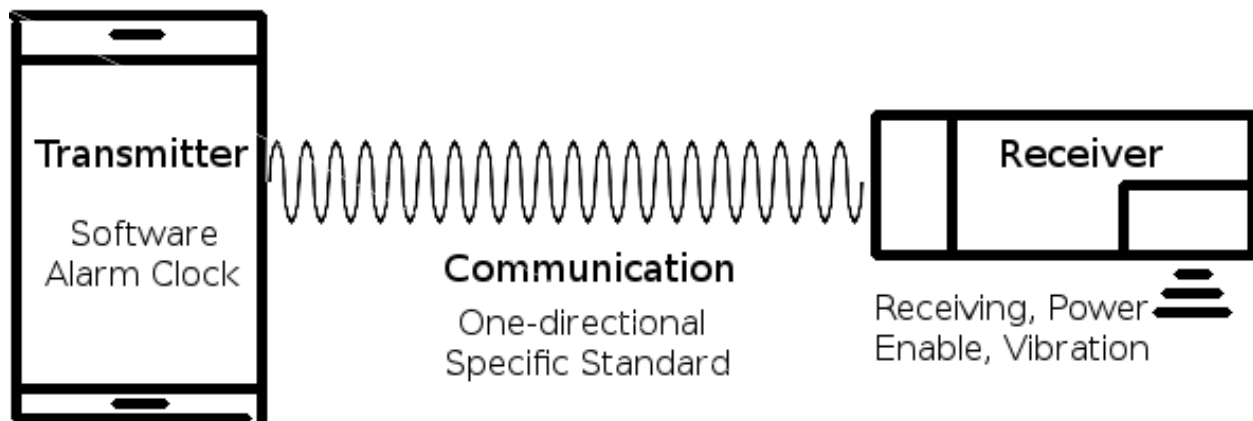


Figure 3: Graphic overview of functional decomposition of project. The smartphone behaves as the transmitter, sending the signal. Receiver gets signal, activates, and vibrates accordingly.

For the final design requirements, the signal device will be any smartphone model, while the software will be programmed for either iOS or Android operating systems (the two major ones). Utilization of wireless capabilities will simplify the process for designing the receiver. The receiver's build will be based on hearing-aids, but not exactly due to health concerns. Instead of

having an in-ear portion of the device, the receiver will go around the ear, with a battery hatch at one end. The exact location of the vibrator is still unknown, but it will be part of the receiver. Lastly, recharge-ability and power indication are periphery features and no longer part of the main goals of the prototyping stage. Together with these goals and constraints, our primary requirements are: this device works in a 20x20 feet room; is quiet enough to avoid waking non-users; is loud enough in vibration to wake the user, and is wireless.

Table 4: Final Decisions on Design Requirements

System Section	Decision
Transmitter	Smartphone: design an application for either iOS and Andriod devices, while using into the built-in communication technology. Will be much easier to make.
Communication	One-directional signals shared between the transmitter and receiver.
Receiver	Hearing-aid style device: Around ear, with battery compartment. Vibrator location and position still unknown. Range no more than 20 feet.

DESIGN ALTERNATIVES

After determining the design specifications, options for implementing the design specifications were researched. These however are not final and were changed during the Final Design and Implementation process. Based on the functions, options were then compared, just like Design Requirements, by their advantages and disadvantages, as well as their specifications and cost. Options were researched for four functions: communication, circuit power, vibration, and software. Only these were chosen because the other requirements were automatically met by selecting the smartphone as the transmitter.

Software and Communication would directly affect each other, so hardware and programming were considered when selecting options. For software, it was initially determined that either Android or iOS operating systems would be used. Applications could be easy to program for both systems using “PhoneGap”, a program that builds versions of applications for

other operating systems, and thus the device would be ready for most of market. The only problem would be hardware compatibility issues. Though most Android-based smartphones and iPhones share similar hardware, their wireless capabilities would be limited by each other's differences. Android OS has open-source coding, community support, and many built-in classes, especially for Bluetooth devices. Furthermore, everything can be programmed in Java, which the project author has past experience. The iPhone operating system, iOS, is already widespread due to high sales and a relatively huge customer base. Even with these benefits though, both operating systems have their disadvantages. The project author would still need to learn how to program for either operating system. Android OS appears to be the better and cost-effective choice, as iOS built-in libraries are proprietary and Apple (the creators and maintainers of iOS) has strict application release policies.

For communications standards and technologies, we looked at four different options for implementation: Smartphone Hardware, Wi-Fi, RFID, and Bluetooth. Using the cellular signal of the phone was immediately rejected in the early stages of the decision process; although an interesting use of hardware, it is very specific to the smartphone models selected and probably would be illegal to tamper with. Wi-Fi could have been a good option because of its ubiquity throughout all communication devices, but since it is a rigid and complex IEEE standard with a lack of small-build, ready-to-use components on the market, this was not further explored. RFID and Bluetooth were both thoroughly researched as potential options, both with great benefits and difficulties. RFID can be used with readers that scan up to 25 meters, satisfying our range requirements, and the technology is already built in to some smartphones. Similarly, Bluetooth is available on practically all current smartphones, while it is easy to buy transceiver modules for circuits. But, the structural and compositional differences between these two options is significant. Bluetooth receivers vary depending on which components you buy and must adhere

to certain “profiles”, or specific device usage, of the standard. RFID comes in either a sizable two or three part system (scanners, transmitters and tags), is not as cheap as Bluetooth, and would be using the technology in a manner that deviates from its original purpose. Some smartphones come with NFC (Near Field Communications) technology that implements RFID, but this is not long-range and would require attaching an RFID scanner and transmitter on to the smartphone. In terms of cost, an entire custom-built RFID system could cost anywhere from \$50-\$100 upward (including all parts), while simple Bluetooth transceivers for circuits can cost \$5-15 dollars at a minimum. Out of all of these options, Bluetooth is the most cost-effective with readily available components and allows for a simpler communication system.

There are not many ways to power the receiver due to its small size, so the primary focus was to use button batteries. Since feasibility is also a priority for prototyping, other battery types were considered and researched in the process. Button batteries, specifically for hearing-aids, are manufactured into four main types: 10, 13, 312 and 675; all of them having similar voltages, but typically near 1.5 volts[3]. These types researched are all zinc air batteries that are disposable, once the pull tab is removed, they activate and only last about week depending on the hearing-aid[3]. The benefits with these batteries is that they are small and compact, but they are low voltage and rechargeable ones are expensive. Another, slightly larger option investigated were coin cell batteries, mainly lithium and silver oxide-based ones. Coin cell batteries can vary depending on the type of manufacturer, but these typically carry a voltage of 3 volts with an amp-hour rating of 200-250mAh, bought in various sizes[4][5]. They are small and inexpensive, similar to button batteries, but they require specific holders, as would any of these batteries. One more type looked at were AA batteries, thought to be slim, common and cheap. Usually 1.2-1.6 volts depending on the material, though lithium-ion ones carry twice as much. Their current ratings are at least three times as much (>600mAh) as the coin cell batteries, but they are too

large for the project and are only useful for the prototyping stage.

The last portion of the system is the vibration, of which only two options were explored: mechanical actuators and piezobuzzers. Mechanical motors were initially sought after, since they create motion and vibration in a compact shell, but ones of decent quality turned out to be expensive and initially not suitable to the project by design. This leaves the piezobuzzers, which create a buzzing noise that changes in volume depending on the voltage. These buzzers come from various manufacturers and can be bought cheaply on online marketplaces (like Amazon), or in mass orders through direct sources (like American Piezo [6]). They also are constructed with leads or pins for attaching to a circuit. To generate the buzzing noise, these buzzers use a square wave that causes a metal disk inside to expand and contract, and thus a wave generator is the only crucial requirement.

Overall, out of the four main functions of which options were explored, possibilities were compared and whittled down to one for prototyping: Android OS for the software; Bluetooth for signaling; coin cell batteries for power; and using piezobuzzers for vibration. Some of the reasoning behind these decisions is further detailed in the next section, but they are not final.

PRELIMINARY PROPOSED DESIGN

The main design proposal is divided in several portions, similar to the functions from the Design Requirements/Alternatives sections earlier in the report, the first and important being the software that will interface with the user. Since the transmitter will be part of a smartphone, one out of several operating systems was selected as a base for the application. Android OS was chosen due to some of its major advantages: open sourcing which allows ease of programming, community support, and available packages. Programs for the operating system are all written in an extended version of Java, which will be easier for the project author to adapt. The operating

system also comes with several Bluetooth APIs (programming interfaces), that will aid with the communication aspect of the project. Since Android comes with various Alarm related classes, used to set alarms for any smartphone, most of the software will use these as a base with some modifications, such as interaction with one of the Bluetooth APIs and periphery settings for the receiver.

All communication will be carried out through the Bluetooth wireless standard, due to its versatility and wide availability on smartphones. When searching for transceiver modules, Bluetooth modules were easy to find, their low cost further swaying the decision. The transceiver that will be used for prototyping is an Arduino-based wireless Bluetooth transceiver. Called the JY-MCU, it is an HC-06 transceiver and is hard-wired to link with other Bluetooth devices. HC-06 type modules are always set to “slave mode” and therefore cannot be programmed to communicate with other devices on their own. Since this transceiver is designed to work with Arduino micro-processing boards, we will use a test circuit to determine its logical behavior. Constructing a simple LED switch circuit, we can connect an Android smartphone to an Arduino UNO board, sending out a “1” or “0” through BlueTerm, a Bluetooth emulation software for the operating system[7]. While the circuit is running, the sending and receiving wires will be connected to the LogicPort, a logic analyzer manufactured by Intronix[8]. This analyzer will allow us to examine the signals being transferred from the transceiver to the circuit board. Once testing is complete, we may choose to design a logic circuit where the transceiver can send signals to the piezobuzzer without the need of the Arduino board. The total cost of this portion of the receiver could be less than \$10 to manufacture.

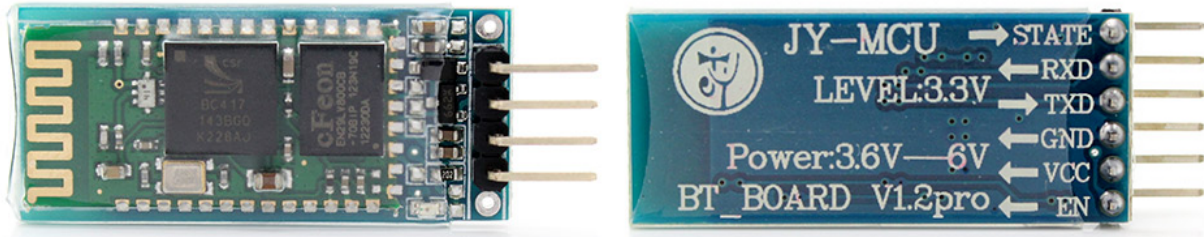


Figure 4: JY-MCU HC-06 Bluetooth transceiver to be used in the project. RXD and TXD are the receiving and sending pins, GND and VCC for power.

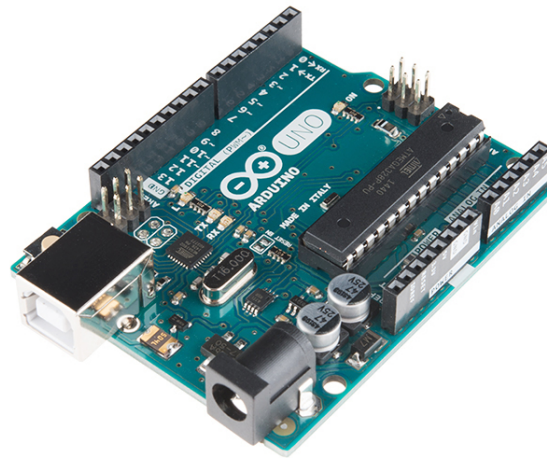


Figure 5: Arduino UNO R3, to be used in the testing phase

Since the Bluetooth transceiver requires 3.6 to 6 volts to operate, the circuit will be powered by two 3-volt coin batteries, particularly the CR2032. These batteries will be wired to the circuit using a holder with leading wires and will be able to power the transceiver, the logic circuit and piezobuzzers. The piezobuzzers that will be used in the circuit can handle anywhere from 3 volts to 24 volts, thus the current power for the prototype is sufficient and the low voltage will create noise that is quiet enough. The buzzer will also be connected to a switch that will enable power, either activated manually or by the logical circuit.

Table 5: Main Parts for Prototyping Stage

Part Name	Function	Reasoning
Sony Xperia Z3	Smartphone / Transmitter	Project author's personal smartphone, Android-based
JY-MCU HC-06 Bluetooth Transceiver for Arduino	Receiver / Communication	Popular transceiver for connecting Android devices
Arduino UNO R3 Microcontroller	Logic / Testing	Used only for the testing phase to determine logic and behavior of transceiver
CR2032 Coin Cell Silver Oxide Battery	Power	Two will provide sufficient voltage to circuit
Generic Piezobuzzers (3-24V)	Vibration	Simple buzzers for the prototyping stage, can handle wide range of voltages

With all the parts together, the receiver for the preliminary design was intended to consist of a Bluetooth transceiver, a logical circuit designed to activate the buzzer, all of which will be powered by two coin cell batteries. For transmission, an application will be designed utilizing various built-in classes with one of the Bluetooth APIs. Users will be able to set the alarm through this application, and once the alarm goes off, the smartphone will send a signal to the receiver. Assuming the software is given out for free with the product, the total cost of the transmitter and receiver should be limited to the receiver, about \$10-\$20.

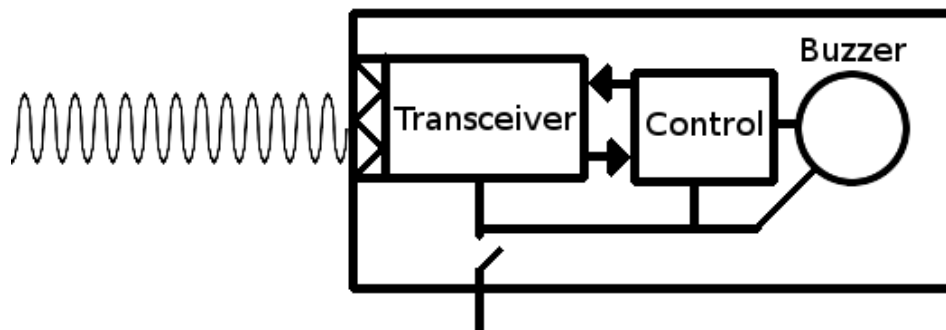


Figure 6: Illustration of basic receiver design: signal is received, the transceiver communicates with the logic circuit, activating the buzzer. All three powered by battery.

FINAL DESIGN AND IMPLEMENTATION

Before implementing the design, quite a few changes were made based upon component behavior, usability of these components, and remaining time in the production schedule. Instead of attempting to analyze the behavior of the Arduino's transmit and receive pins, the focus was shifted to using a smaller microcontroller while also creating a circuit that uses the Arduino UNO. Thus, there would be two versions of the same circuit, one to test the concept of the receiver system, and another acting as a smaller prototype. During testing, the piezobuzzers bought proved to be too loud for our noise requirements, beeping instead of buzzing. These were then replaced with a small vibration motor [9], shown in Figure 7, cheaper than mechanical actuators and functioned as desired, but raised a concern with circuit power that needed to be resolved. All other parts slated for the prototyping stage from the preliminary design were kept, such as the Bluetooth transceiver, while other basic circuit components were added to ensure proper current flow.

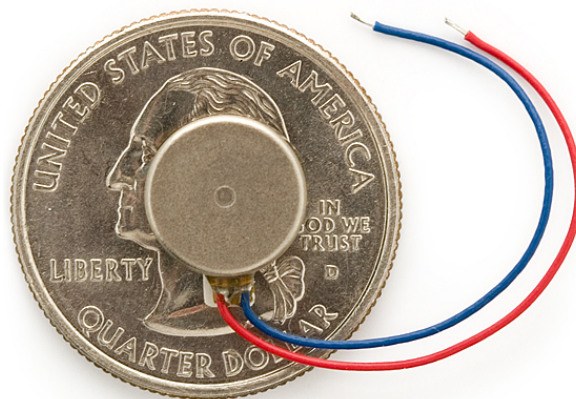


Figure 7: Image of Vibration Motor used in circuits, compared with quarter

The Arduino circuit is a modified version of our LED switch circuit from the testing phase of our preliminary design, incorporating the new vibration motor and batteries. The Bluetooth transceiver is powered by the Arduino UNO, while the Rx and Tx pins are connected to the Arduino's Tx and Rx pins respectively. Since the transceiver operates these pins at lower

logic levels than the Arduino, 3.3V compared to 5V, a voltage divider was added between the Arduino's Tx and the transceiver's Rx pin. The motor had much higher current requirements than what the Arduino could provide, but could operate on the voltage of the selected button batteries. To create a proper current gain, the pin previously used for the LED in our older circuit was connected to the base of a 2N3904 bipolar junction transistor, while the motor was hooked up to the collector with the CR2032 button battery as the voltage source. Nothing needed to be placed in the emitter end and thus the circuit was complete. The final schematic can be seen in Figure 8.

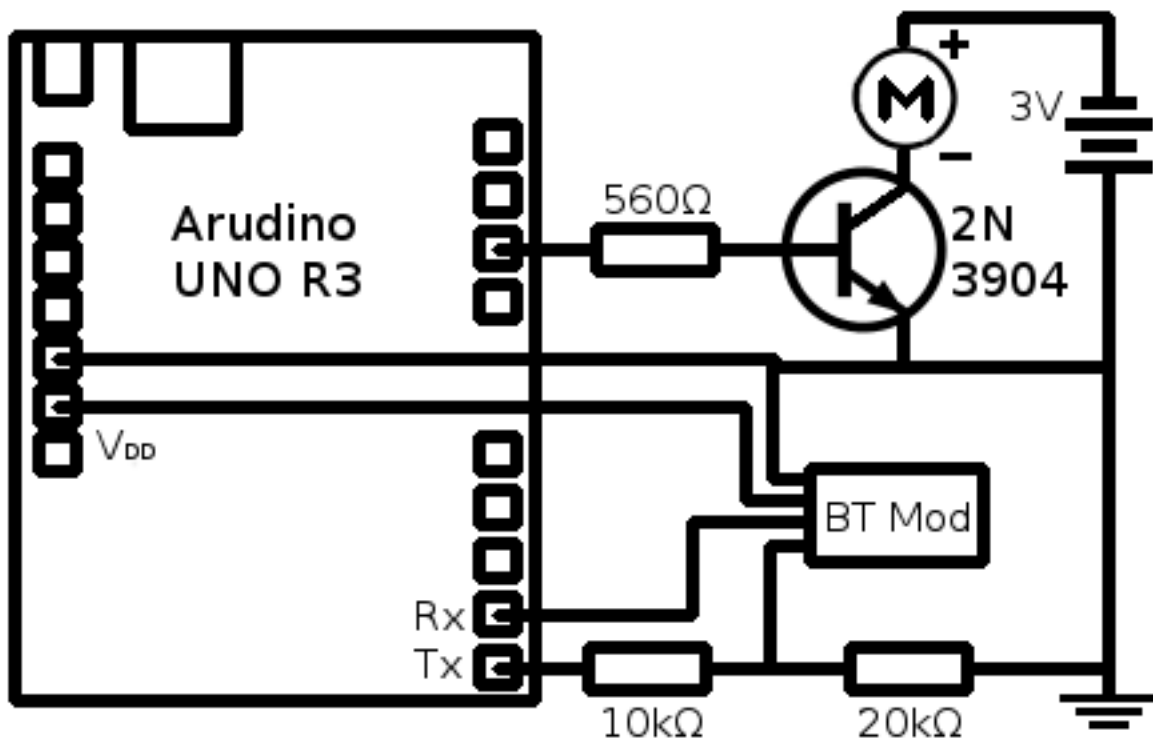


Figure 8: Schematic of implemented Arudino-based circuit, with component values

The internal behavior of the Arduino UNO in this circuit is based on a simple, straightforward program, the code of which is provided in the Appendix. When the Arduino is first turned on, it sets the output pin for the motor while also setting its internal baud rate for communicating with the Bluetooth transceiver, in this case a data rate of 9600. The controller then waits for any transmission to enter its receiver, upon which it checks the value encoded in

the signal. If the value received is a '1', the motor will turn on. There is also a check to see if the return message, to confirm with the transmitter it has received the '1', has already been sent. If not, the return message is sent. The controller will automatically shut off the motor after 30 seconds, if the user has not already done so, in order to conserve energy on the current driver battery. The motor can be activated again at any time. However, due to the behavior of our software application the Arduino circuit needs to be powered off and back on again.

For the smaller microcontroller circuit, some additional research was conducted to decide on a smaller substitute for the Arduino. The chosen substitute were microcontrollers with the PIC architecture, in this case the PIC24FV16KA301 in a PDIP casing [10], seen in Figure 9. PIC microcontrollers come in various customizations for program memory, EEPROM, I/O ports, special functions and so on. Since our Bluetooth transceiver is a serial communication component using the RS-232 standard, our decision for this particular variant of PIC was based upon whether it had a UART (Universal Asynchronous Receiver-Transmitter) port, which can handle the serial standard. The internal UART module for these types of microcontrollers also have their own dedicated registers for handling the transfer of data through the port, making the programming process easier.



Figure 9: Concept image of PIC used in circuit

The circuit construction is similar to the previous Arduino circuit, with some modifications to suit our new controller. For power, the microcontroller will be directly connected to our 6-volt double button battery source from the preliminary design. However, the PIC24FV16KA301 allows only a maximum voltage of 5-volts. To resolve this voltage problem, the voltage source will be connected to a voltage regulator, which will restrict the voltage to 5-volts and keep our controller safe. This is still enough power for all the other components in the circuit, including the Bluetooth transceiver, but there is an issue with the I/O ports on the microcontroller.

The PIC selected for this project behaves normally with low voltage values, cases where the value is a 0, but there is not a large enough range for high voltage values such that we can wire the components the same as the Arduino circuit. Higher voltages in to the microcontroller must be about four-fifths to a whole of the source voltage (about 4 to 5 volts in our setup) in order to be read properly, while high voltage coming out ranges from 4.1 to 4.4 volts. If we base our circuit upon the Arduino circuit, both the Bluetooth transceiver and the motor need to have voltage dividers and proper resistors chosen to ensure that neither component is fried. The Bluetooth transceiver has the voltage at its Rx pin dropped with the base of our current driver dropped for the motor, which our previous transistor can still handle. Since the motor itself is powered by our voltage source from the voltage regulator, we also drop the voltage with a divider for its input voltage. For programming and operating purposes, this variant of the PIC microcontroller also needs to have specific capacitors placed on various pins, mainly the reset clear, voltage source and drain, and the special capacitor pin. Both the schematic for this circuit and the implementation on a performance board (compared to a quarter) can be seen in Figures 10 and 11.

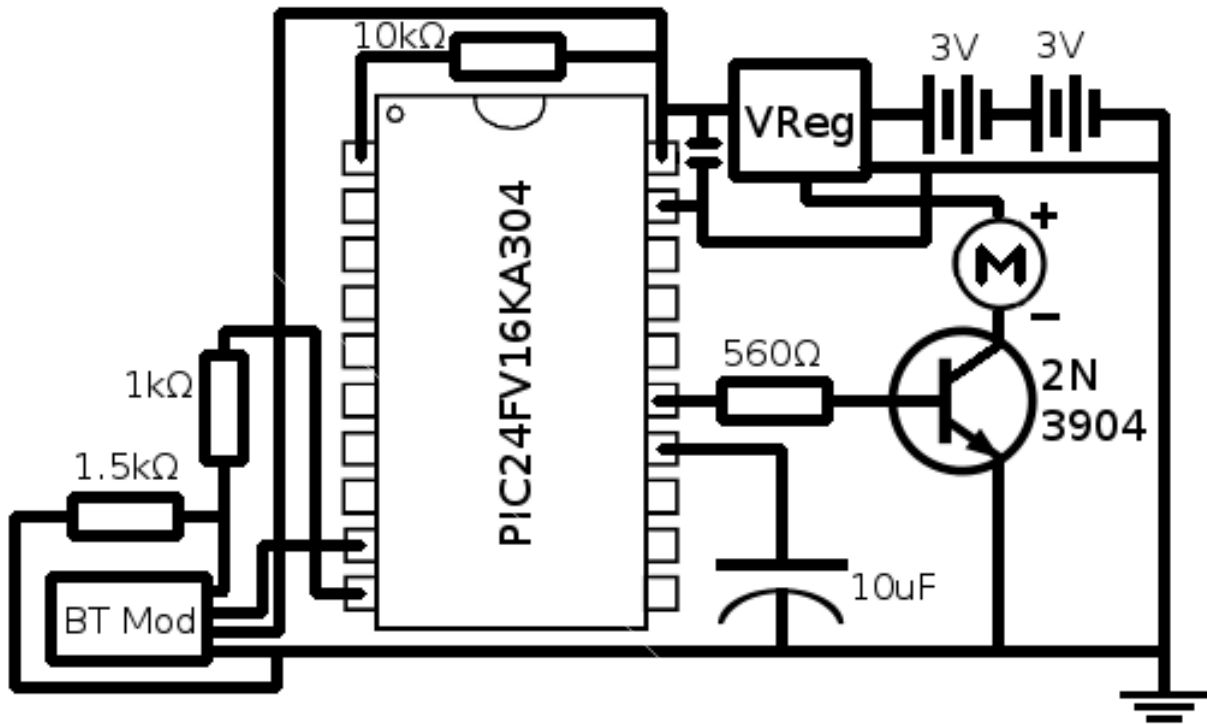


Figure 10: Schematic of implemented PIC-based circuit, with component values

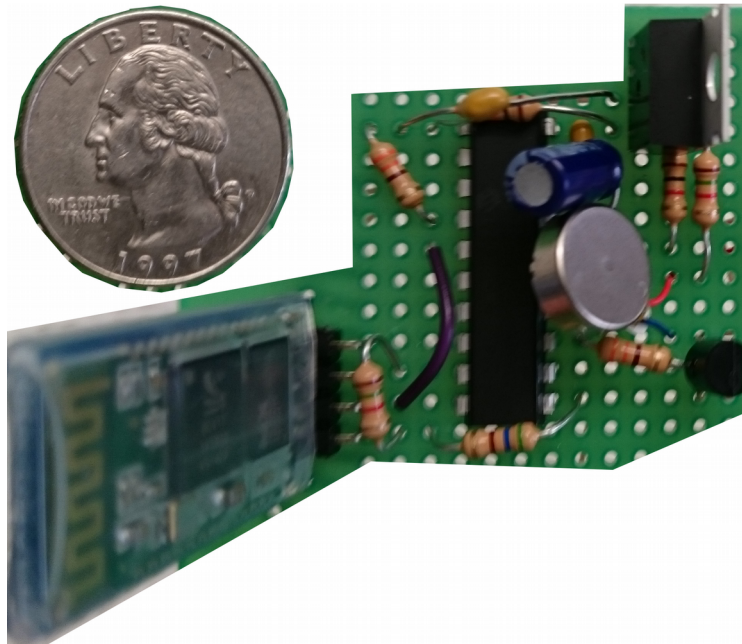


Figure 11: Image of PIC circuit on performance board, compared to quarter

The internal programming of the PIC microcontroller in this circuit was formed from most of the same logic as our Arduino program, just with a longer set up process. Before we can

use the UART on the PIC, we set the parameters of our UART modules behavior, including datasize, baud rate, parity and other settings. Since our PIC contains two UART modules, which changes the pin mapping depending the one selected, we only picked and specified the first module in our program code. The module also needs to know which bit data mode we will be operating in, as well as the parity and stop bits. We want the Bluetooth transceiver to read in one byte at a time, equivalent to the amount of information in our ‘0’ and ‘1’ values from the Arduino, so the module was set to operate in an 8-bit mode, with an even parity check and two stop bits. After setting these parameters, we then enable the module and wait.

With the Bluetooth transceiver handling the connection between the smartphone and the PIC, the microcontroller only waits for a “receive interrupt”, indicating that the Receive Register (UxRXREG) has data in it. Next, it will check if there was any overflow in the register and run a parity check. From this, the microcontroller will know if we are receiving the proper data types. Lastly, the data from the Receiver Register will be read and compared to our signal values. If the byte received is a ‘1’, then the motor will be set to voltage high. Just like the Arduino circuit, the PIC will wait 30 seconds before turning off the motor automatically.

Theoretically, this program will work, but there were some difficulties in programming the PIC. Using the PICKit3 debugging tool[11], attempts were made to download the program to the chip, but the code was wrought with compilation errors that were difficult to fix. Downloading blank template programs to the microcontroller worked, but would not with the planned one. Further scheduling issues resulted in lack of time for field testing the program, so it is unknown whether the code works. Even so, the code has been provided here in the Appendix for your edification, and the problem could be solved by the reader if they choose to do so.

The other half of our implementation consists of the smartphone application. When programming an application for Android OS, there needs to be a few rules as to which versions

of the operating system the program will work on. For this project, the minimum SDK requirement was set to 19, while our target API was 23. This means that our program will run on any smartphone with Android version 4.4 (Kit-Kat[12]) or later, but compiles for Android version 6.0 (Marshmallow[13]), following that version's programming syntax. This will cover about 70% of the current Android market, as of March 2016. The software was also coded using Android Studio[14], an integrated developing environment specifically design for Android and backed by the developers of the operating system.

The behavior of the application was divided into three separate classes. "MainActivity" handles the main menu, the time selection from the user, and setting the system alarm service. "bluetoothSearch" searches for the receiver on the users command, using the smartphone's system Bluetooth service as a transmitter. "alarmReceive" establishes the remote connection with the receiver and manages the communication between the devices. All these class are managed by a "manifest", as well as several layout files that control the flow and appearance of the application. The code for the three main classes, including the string resources for readability, is provided in the Appendix.

Inside of MainActivity, there are several sub-classes and methods that represent certain behaviors of the application. Aside from the onCreate and onActivityResult methods, which are standard for the main launcher activity of an Android program, this class contains implemented versions of TimePickerDialog and WakefulBroadcastReceiver, both abstract classes. The TimePickerDialog contains a time picker widget (Figure 12), which allows to the user to select a time when they press the "SET ALARM" button on the main menu. This information is then sent to the internal setAlarm method within MainActivity, which is used to set a java.Calendar instance that is converted into milliseconds for AlarmManager. The Alarm Manager class allows us to get the system alarm service and set a time for the alarm to activate. Due to differences

between versions of Android, AlarmManager has to be set based upon the user's installed version. Once we know which version, we can then have this alarm set to the system alarm list, which upon passing will go to our WakefulBroadcastReceiver. This sub-class is designed to be activated when the alarm goes off, which then starts the alarmReceive service. If the user wants to cancel the alarm, MainActivity can still use the same AlarmManager and remove the set alarm from the system alarm list directly.

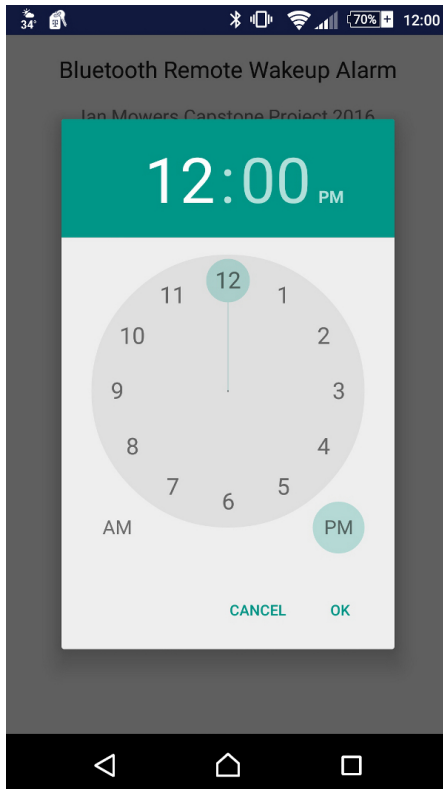


Figure 12: TimePicker on screen



Figure 13: Bluetooth device list

“bluetoothSearch” will start up when the user presses the “FIND ALARM” button. This class contains an implementation of the BroadcastReceiver abstract class, which will collect MAC addresses from Bluetooth devices in the vicinity of the smartphone. These MAC addresses are all that is needed to initiate a Bluetooth connection. On the screen, they will be divided into two lists: Previously Paired Devices and New Devices, as seen in Figure 13. The receiver will appear under the former list if the system has been used before, as the smartphone will remember

past devices. If the receiver is not found on screen, then the user can press the “Scan For Alarm” button, which will start a discovery search process for nearby devices. To obtain the MAC address, the user simply selects the receiver, and the address is parsed out of the list and saved in the MainActivity for use in the alarmReceive class.

As stated before, the alarmReceive class behaves as a service that is activated by the MainActivity’s “WakeReceiver” and is not visible to the user. When activated, the service runs and manages the ConnectedThread sub-class. This sub-class first creates a Bluetooth connection through a BroadcastReceiver and then uses read and write functions to communicate. In our case, this Thread is immediately opened using the MAC address provided and sends the ‘1’ character out to our receiver. Once the data is sent, the service closes the connection immediately, since our system involves only one-way communication between the smartphone and receiver. This also “unregisters” the BroadcastReceiver within the alarmReceive service, meaning that the receiver needs to be powered-off between each use.

Overall, the program runs through a series events, as laid out in Figure 14. The user first opens the application and is prompted to turn on their Bluetooth service for bluetoothSearch. Upon doing so, they then find the receiver and pick it for the application to remember. The MAC address is saved, which then allows the user to “set” the alarm through the TimePickerDialog. If the receiver has not been found, then the alarm cannot be set. Using the TimePicker, the user sets a preferable time, which the application will then set the system alarm service. At this point, the user can actually close down the application. When the alarm time has passed, the alarmReceive service will be activated, establishing a connection with the receiver portion of our system. Lastly, this service will send a signal to the receiver, which will turn that circuit on, left to be shut off by the user.

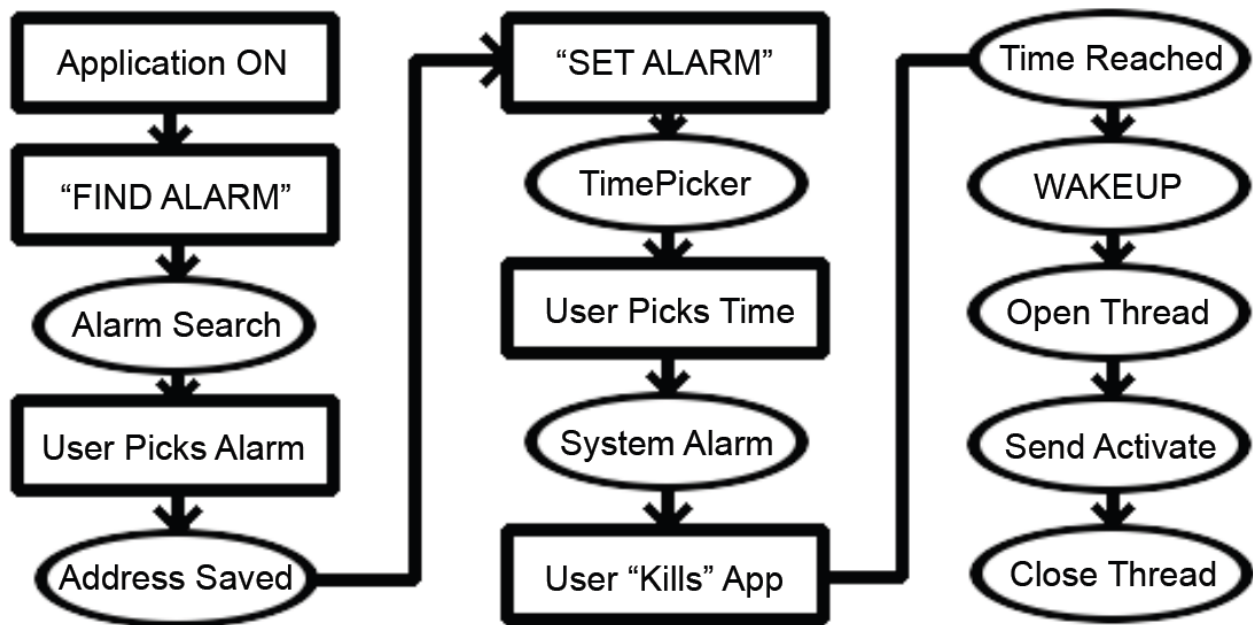


Figure 14: Logic flow of smartphone application operation process

PERFORMANCE ESTIMATES AND RESULTS

Even though both portions of the solution were implemented, tests were performed to check if the system functions as designed. For the application part, the alarm needed to appear on the system alarm list as well as sending the signal out to the receiver. Both of these requirements confirm if the smartphone is remembering to activate the alarmReceive service at the designated time while also communicating with the receiving end. For the receiver circuits, the motor needed to turn on and we needed to check the power requirements of the system compared to calculated estimates and the datasheets. For the overall system, the smartphone needed to wakeup quickly, send a signal to the receiver, establish the Bluetooth connection, activate the motor, and shut the motor off automatically, all within the required range of 20 feet. Before testing the overall functionality of the system, current readings of the Arduino-based circuit were measured first, while the PIC-based circuit was estimated. These measurement were compared to their estimated values, shown in Table 6.

Table 6: Estimated and Actual Measurements of the ArduinoCircuit

Aspect Measured	Estimates / Datasheets	Actual
Bluetooth Transceiver (Not Connected)	30~40 mA	4~40 mA
Bluetooth Transceiver (Connected)	~8 mA	20 → 3.2 mA
Current, Bluetooth Tx Pin to Arduino	~8 mA	5.11 mA
Current, Arduino to Bluetooth Rx Pin	???	.53 mA
Current, Motor from 3V (ON)	85~100 mA	44~48 mA
Current, Motor from 3V (OFF)	0 mA	0 mA

One of the major areas of discrepancy between the estimates and the actual current measurements is the Bluetooth transceiver. Although the datasheet states the transceiver will be between 30 and 40 mA, the current actually fluctuated between 4 and 40 mA when powered by the Arduino UNO. When the transceiver establishes and maintains a connection, the current is supposed to drop to about 8mA, but instead started at 20mA and quickly dropped to 3.2mA. This significant drop in power is beneficial, as less current is being drawn and thus less power needed. However, when it is not connected, i.e. the duration of typical use for the system, there will be a serious energy issue not only in the Arduino-circuit, but more so in the PIC-based circuit.

The motor was another major concern in terms of power, since it requires a larger current draw to start up. Datasheets for the motor stated that it needed at least 2.6 volts along with an 85mA current to start up. When the motor ran on the Arduino circuit, the current ranged from 44 to 48mA, varying depending on the amount of pressure applied to the motor or whether it was anchored down. This was guaranteed by the common-base setup for the transistor switch, but also the 3-volt 210mAh button battery ensured the proper power. In our Arduino circuit, the motor actually runs with less current draw than expected; a great benefit that reduces strain on the microcontroller and the button battery. Due to the estimated lower current requirements of the PIC microcontroller's I/O pins, the motor should not pose a current draw problem but this

aspect still needs to be tested.

Next, the application was tested to see if it would activate the receiver circuit properly. This was done simply by going through the activation process as a future user would. During the early testing stage, the range of the Arduino-based receiver was tested in a radio-noise free environment with five trials, each distinguished by the distance the receiver was tested at. The results of these trials with comments is presented in Table 7.

Table 7: Range Trials for Arduino Circuit

Trial #	Parameters			Comments
	Time (min)	Distance (m or ft.)	Additional	
1a-b	5	.3 m / ~1 ft.	a – phone Bluetooth off b – phone Bluetooth on	a failed, b success
2a-c	6	1.4 m / 4.62 ft.	None	a-b failed, code fixed c success
2d-f	1~2.5		(code fix) Bluetooth automatically enables	d-e failed, code fixed f success
3	4	~2.7 m / ~8.9 ft.	No line-of-sight, Arduino replugged in	Success
4	5	~4.5 m / 14.85 ft.	Same as #3	Success
5	4	~6m / 19.8 ft.	Same as #3	Success

The first two trials both failed on their first round, due to errors or quick fixes in the application code. Even after the code was corrected during Trial #2, it was discovered that the Arduino circuit needed to be plugged in again after one round of use, since the application unregisters the BroadcastReceiver for connecting and communicating with the circuit. The subsequent trails had more consistent parameters: removing line-of-sight by placing human head in front of transceiver, ensuring that Bluetooth was turned off on the phone, and that the receiver circuit had been previously unplugged before setting the alarm. In all of these trails, the circuit worked at each distance, even in the final trial of approximately 20 feet, meeting our desired range requirement on the Arduino circuit.

Now with these measurements, we can make changes to the design to increase feasibility of a final product and improve the system behavior. Due to programming problems, the PIC microcontroller was not tested and current measurements were not taken for lack of faith in the setup presented in the schematic. The type of PIC microcontroller used also contained more features and functionality than needed, and could be reduced to a smaller version that meets the bare minimum design requirements we presented earlier. The next time this system is implemented, we will ensure that this circuit is properly built and runs as intended. As for working parts of the Arduino-based circuit, the Bluetooth transceiver turned out to be a major power drain on the system, which could be disastrous for the PIC-based circuit, where in that case the power would only last for one typical night of sleep (9-10 hours). This was not as low-powered as expected, so either a Bluetooth transceiver with a lower current rating needs to be used, or the system should run on the Low Energy Bluetooth standard[15]. The power consumption of the Arduino microcontroller itself was not considered during the testing process, but by design the unit requires way too much power for our future circuit, and thus must be removed entirely and replaced with the PIC or a smaller microcontroller.

PRODUCTION SCHEDULE

Designing the solution took several steps, involving some planning and decision-making along the way. The first step in this process was to finalize the functional decomposition before moving on to researching parts. Various types of parts for each aspect of the preliminary design, including some that would be part of the final design, were compiled into a spreadsheet for comparison on cost and specifications. These parts were then ordered over the next few weeks as decisions were made about which specific types would be used. Once most of the parts were procured, testing was started on the Bluetooth transceiver and interaction with the Arduino board and smartphone, using the LED switch circuit discussed in the Preliminary Design.

At this stage, the design still needed to be finalized, but implementation had also begun. The piezobuzzers were first tested using a modified version of the LED switch circuit, while basic training on programming for Android OS was completed. Next, various PIC microcontrollers were researched, compared and picked for the final design and implementation, to which samples were ordered along with a physical debugging tool. Before receiving these parts, logic for the smartphone application was planned out with initial programming practice on some key behavior of Android programs. Once the microcontroller samples were received, the schematics for both the Arduino and PIC circuits were finalized.

At the same time, some more research on built-in Android functions was completed and programming the actual application had begun. There were initial problems with the code in its early stages, but in a short amount of time the application began functioning with the Arduino circuit. Afterwards, this software was repeatedly tested with the Arduino, while the code was cleaned up and finished. Focus then shifted toward programming the PIC circuit, which proved to be a difficulty. With the PIC microcontroller set up in a circuit for programming, as required by the user guides, we were able to download a blank sample program to the device, but the actual program developed for this project would neither compile or download. Within the remaining time left to work on this project, this was the one issue that remained unresolved.

Some improvements could have been made in scheduling to ensure that all aspects of the implementation stage were actually finished. Learning how to use and program the PIC microcontroller should have begun earlier in the design stage, with the PIC itself considered as a design alternative in the preliminary design. Also, the workload could be spread out with timing planned better, delegating proper amounts to each part of the implementation, especially researching, circuit design, and programming. However, we would need a better sense of the time requirements for completing certain parts of the project, which could not necessarily have

been known in advance.

COST ANALYSIS

To ensure that either system could be sold as a product, we need to check to see if the cost is feasible for market. If it is not at a reasonable price, the product may not sell and thus cannot be produced to reach consumers. The pricing for components of each circuit is shown in Table 8 for comparison.

Table 8: Cost Comparison Between Implemented Circuits

Arduino Circuit		PIC Circuit	
Parts	Cost	Parts	Cost
Arduino Board	\$24.95	PIC24FV16KA301-I/P	\$2.90 max
Vibration Motor	\$4.95	Vibration Motor	\$4.95
CR2032 Battery	\$1.95	2xCR2032 Battery	\$3.90
Bluetooth Transceiver	\$4.95~9.95	Bluetooth Transceiver	\$4.95~9.95
Resistors/Transistors	\$1 max	Resistors/Transistors	\$2 max
		Voltage Regulator	~\$1
TOTAL	\$37.80~42.80	TOTAL	\$19.70~24.70

Comparing these two, the PIC circuit is certainly the cheaper of the two, even though it is a more condensed system, with a total price almost half as much as the Arduino circuit. The only concern about the PIC circuit, as talked about in the Results section, is the significant power drain on the battery. Unless lower-powered components are implemented, the batteries will become an additional cost to the user in the future. The Arduino circuit poses a problem with cost on top of its bulkiness; the exorbitant price further cementing our reasoning that this circuit is not feasible as a product. For future versions of the system, we will see if we can cut costs further, because although the PIC circuit is the cheaper option, it still seems a bit expensive for a simple Bluetooth-based receiver.

USER'S MANUAL

Although both the Arduino-based and PIC-based circuit behave similarly, there are a few differences concerning power and usage. The Arduino-based circuit can either be plugged into a wall outlet with an AC adapter, giving sufficient power for as long as needed, or powered by a 9-volt battery hooked up with wires and lead pins attached to the V_IN and GND pins on the unit. The 3-volt button battery for the motor will also need to be replaced with the proper type, CR2032 Lithium-Ion, if the previous one fails. This circuit also turns on automatically once the Arduino is powered, but needs to be unplugged again after each use to clear memory and ensure Bluetooth connects each time.

For the PIC circuit, one needs to simply flip the switch on the battery case and the circuit has power. If the circuit does not work during use, i.e. the motor does not vibrate, most likely either one or both of the 3-volt button batteries is dead, which need to be replaced with the proper type just like the Arduino-based circuit. If that does not solve the problem, then any one of the components within the circuit is dead, most likely the Bluetooth transceiver or PIC microcontroller. In this situation, the circuit no longer works and those components or the whole product need to be replaced.

For the smartphone application, there is a specific order that the user needs to do for the system to work. Future versions of the software will correct the awkward flow. First off, the smartphone needs to have Bluetooth capabilities and must be running at least Android OS version 4.4. "Kit-Kat"[12]. When the user opens the application, he will first be asked to turn on Bluetooth. Once doing so, they need to "find" the alarm, by turning the receiver on and pressing the "FIND ALARM" button on the main screen (Figure 15). If the system has been previously used before, it should appear on the first list, "Previously Paired Devices". If not, the user can press "Scan for Alarm" and the application will search for Bluetooth devices in the vicinity. If

the alarm name appears, then the user taps it and the alarm will be ready for use.

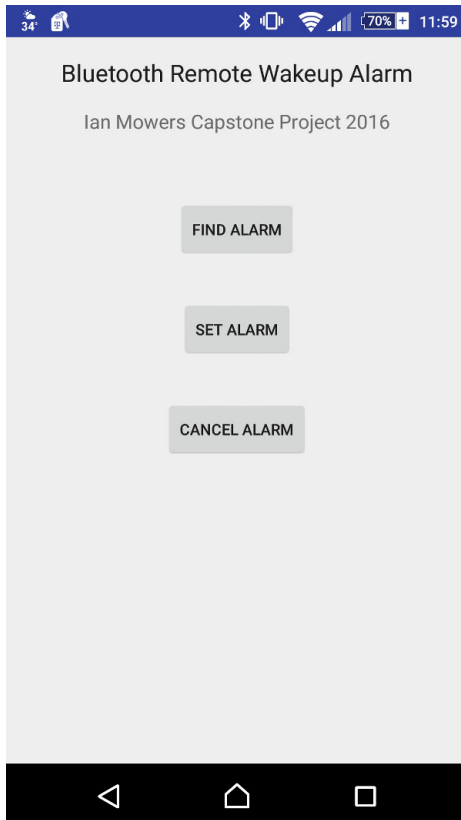


Figure 15: Main menu for smartphone application

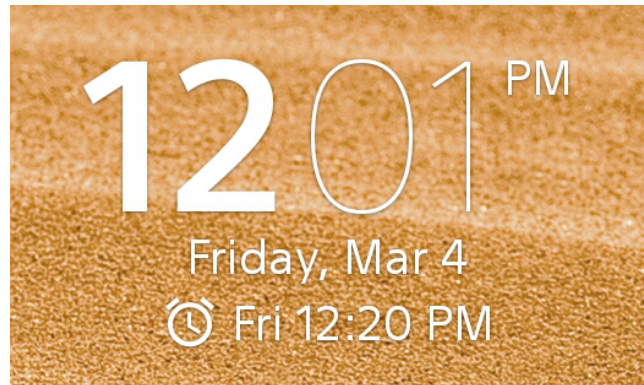


Figure 16: Screen image of alarm time with system clock

Next, the user can press “SET ALARM”, which will display the time picker. The default time presented will be the current time when the button is pressed, so the user should set a time that is in the future, pressing and dragging the hours and minutes to the desired time. If the user sets the wrong time, they should pressed “CANCEL ALARM” and try again before closing the application. At this point, the application can be closed and Bluetooth turned off, since the time is internally saved in the smartphone and Bluetooth will automatically turn on at the alarm time. The phone may be screen-locked as well, but should not be powered-off, and the user can check the alarm time by looking at the time next to the little alarm clock symbol found anywhere the time appears on screen (Figure 16). At most, there will be a ten second delay between the smartphone activating the alarm service and the receiver activating. When the receiver motor

begins to vibrate, the user may shut off power to the circuit after waking up or the motor will shut off automatically after thirty seconds.

DISCUSSION, CONCLUSIONS, AND RECOMMENDATIONS

The main point of this project was to create a quieter alarm device, that users could wear and would allow them to wake up without waking others in a room. The final implementation needed to be compact, but able to handle wireless transmission such as to communicate with a smartphone, and had to be powerful enough to work in a bedroom of typical size.

We first defined how our design to the solution would function and researched potential approaches of implementing the circuit. Alternatives to each portion of the initial design were decided by the conclusion of the preliminary design stage. Upon procuring initial parts, the design was finalized based upon further research and some initial testing of the solution. With this information, an alternative to the Arduino-based circuit was sought through the PIC microcontroller; the Android application being developed in tandem. The final PIC-based circuit was not developed past the design stage, as this version of the implementation was plagued with programming issues. Focus was then shifted on to test if the software application would work with the already functioning Arduino circuit. In the end, the two-part system worked as expected within the required range and thus we can conclude that the concept works, but implementation needs tweaking and refining for the future.

Both circuit designs had current supply issues, as the Bluetooth transceiver used in both circuits required too much current in order to not be continuously connected with the smartphone. Even though the Arduino circuit was feasible, it was neither compact nor low powered to operate as an on-ear device. The PIC circuit's power requirements remained partially unknown and untested, but will be done in future work.

For future implementations, smaller microcontrollers will be needed along with a much

smaller Bluetooth transceiver. The PIC microcontroller chosen for this project possessed more abilities than needed and contributed to the programming issues. Finding simpler PIC microcontrollers, which do exist compared to the one used this time, that meet the bare minimum requirements is a must. It is unknown whether smaller Bluetooth transceivers can be found, but this is critical for the solution's success, with lower power ensuring that the device can last more than one night of use. If not, then the logical behavior of either the application or the receiver needs to change to account for this problem.

Although the software application worked exactly as planned, there are some aspects of it that need tweaking. Differences in the behavior of the AlarmManager class between the target Android APIs, or versions of the operating system, affected how the alarm was internally set in the phone. This is beyond our control, as the operating system strives to save battery life, but the options for using the AlarmManager beyond version 5.0, which is slowly becoming the bulk of the Android market, are few. Alternative programming structures and methods will be researched to improve this.

How the user runs the application and system needs to be streamlined, since some current behaviors are repetitive or needless. When the user searches for the alarm, if he has previously used the receiver system before, the receiver MAC address will already appear in the "Previously Paired Devices" list and the user can simply select it. The smartphone only needs this address and the application does not communicate with the receiver until the alarm activates, so it would make more sense to only turn on Bluetooth if the user really needs to search for the receiver. Eliminating the need to turn on Bluetooth until alarm time, as well as every time the application is opened, is more convenient to the user. In addition, the overall layout and formatting of the application screens could be cleaned up, along with adding features suggested by peers, e.g. a "snooze" function.

Overall, we learned from this project the capabilities of a system that combines both integrated circuits and software technology. It is possible to consolidate a separate receiver system that will activate at the whim of a smartphone, another one of the many possibilities of this wonderful device. The design and implementation procedure was lengthy, riddled with sudden decision-making and difficulties that affected the final product, but from the failures of the current design, we can learn to better manage the project timeline and do more thorough research up front. Now, this process, designing solutions that contribute to the benefit of humanity, has become a part of my toolbox of future problem solving.

REFERENCES

- [1] – Bluetooth Core Specification <https://www.bluetooth.com/specifications/bluetooth-core-specification>
- [2] – Daily Mail.com, Sarah Griffiths, An alarm clock that DOESN'T disturb your partner – <http://www.dailymail.co.uk/sciencetech/article-2541186/> (January 17, 2014)
- [3] – BatteryMart.com, Hearing-Aid Batteries FAQ – <http://www.batterymart.com/p-2010-04-29-hearing-aid-batteries-faq.html>
- [4] – Information found through a search for Coin cell batteries (mainly Energizer manufactured) on Amazon.com, during the month of October 2015.
- [5] – SparkFun Electronics Coin Cell Battery: <https://www.sparkfun.com/products/338>
- [6] – Specs and Information page for American Piezo Company buzzers: <https://www.americanpiezo.com/standard-products/buzzers.html>
- [7] – Blueterm Application development site: <http://www.pymasde.es/blueterm/>
- [8] – Intronix Company Website: <http://www.pctestinstruments.com/>
- [9] – SparkFun Electronics Vibration Motor: <https://www.sparkfun.com/products/8449>
- [10] – Product Page for PIC24F16KA301 family: <http://www.microchip.com/wwwproducts/en/PIC24F16KA301>
- [11] – Product Page for PICkit3 debugger tool: <http://www.microchip.com/Developmenttools/ProductDetails.aspx?PartNO=PG164130>
- [12] – Development Information page for “Kit-Kat”: <https://developer.android.com/about/versions/android-4.4.html>
- [13] – Development Information page for “Marshmallow”: <https://developer.android.com/about/versions/marshmallow/android-6.0.html>
- [14] – Home page for Android Studio: <https://developer.android.com/sdk/index.html>

[15] – Informational page about the Bluetooth Low-Energy standard: <https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy>

APPENDIX

Arduino Circuit Code:

```
const int buzzpin = 13;          // Set Arduino Pin for output
const int a_second = 1000;      // Amount of ms in a second
const int buzz_time = 30;      // Amount of seconds to buzz
const int baud_rate = 9600;    // Set baud rate for Bluetooth
int turn_on = 0;                // Determines if motor needs to turn on
int was_sent = 0;               // Make sure verification only happens once

void setup() {
    pinMode(buzzpin, OUTPUT);    // Activate buzzpin
    digitalWrite(buzzpin, LOW); // Make sure the motor is turned off
    Serial.begin(baud_rate);    // Turn Serial on with set baud rate
}

void loop() {
    //If the transceiver receives any data, set state
    if(Serial.available() > 0){
        turn_on = Serial.read();
        was_sent = 1;
    }
    if (turn_on == '1') {        // If 1 received, turn on motor
        digitalWrite(buzzpin, HIGH);
        if(was_sent == 1){      // Check if return message sent
            Serial.println("THANKS"); // Send a thank you back
            was_sent = 0;
        }
        for (int count = buzz_time; count > 0; count--){
            delay(a_second);    // Wait the amount of seconds
        }
        digitalWrite(buzzpin, LOW); // Turn off motor
        turn_on = 0;           // Must change to 0
                                // Otherwise will continue looping
    }
    else {                       // Else, Nothing was received
        if(was_sent == 1){      // Check if return message sent
            Serial.println(""); // Send nothing back
            was_sent = 0;
        }
    }
}
```

PIC Circuit Code (Not Working):

```
// Set Oscillator to desired settings
void config_osc(){
    OSCCONbits.COSC = 0b010;
}
```

```

// Initialize everything needed
void setup(){
    U1MODEbits.UARTEN = 0b1; // Enable the first UART module
    U1MODEbits.UEN = 0b11; // Enable pins, here U1RX and U1TX enable
    U1MODEbits.PDSEL = 0b01; // Set Parity-Data Selection, 8-bit even par
    U1MODEbits.STSEL = 0b1; // Set # Stop Bits, in this case 2
    U1MODEbits.BRGH = 0b0; // Set Baud Rate to Standard Mode
    U1MODEbits.USIDL = 0b0; // Allows operation in Idle Mode
    U1MODEbits.WAKE = 0b1; // U1RX pin will be sampled during Sleep Mode
    U1STAbits.UTXEN = 0b0; // Disable transmit bit, since unused
    U1STAbits.URXISEL = 0b01; // Interrupt is set
    TRISAbits.TRISA2 = 0; // Activate RA2 pin for motor, set as output
}

int main(void) {
    config_osc();
    setup();
    // Continuous loop
    while (1){

        // Check if receiver is active
        while(!U1STAbits.RIDLE == 0b0){

            // Check if we overflowed or had a parity error
            if(U1STAbits.PERR == 0b0 && U1STAbits.OERR != 0b1){

                // If buffer data is available
                if(U1STAbits.URXDA == 0b1){

                    unsigned char rec_word = U1RXREGbits.U1RXREG;
                    // Compare data and turn on motor
                    if (rec_word == '1'){
                        LATAbits.LATA2 = '1';
                        wait_ms(30000);
                    }
                }
            }
            // Clear overflow bit and shut off motor
            U1STAbits.OERR = 0b0;
            LATAbits.LATA2 = '0';
        }
    }
    return (EXIT_SUCCESS); // Shuts off when power pulled
}

```

Application Code:

MainActivity.java –

```

package edu.union.imowers.capstonealarmfinal;

import android.annotation.TargetApi;

```

```

import android.app.AlarmManager;
import android.app.Dialog;
import android.app.PendingIntent;
import android.content.Context;
import android.os.Build;
import android.support.annotation.NonNull;
import android.support.v4.app.DialogFragment;
import android.app.TimePickerDialog;
import android.bluetooth.*;
import android.os.Bundle;
import android.support.v4.content.WakefulBroadcastReceiver;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.widget.Button;
import android.content.Intent;
import android.widget.TimePicker;
import android.widget.Toast;

import java.util.Calendar;

public class MainActivity extends AppCompatActivity {

    //Request Codes for Intents
    private static final int BLUETOOTH_SYNC = 50;
    private static final int REQUEST_BT_ENABLE = 100;

    //Layout Materials
    Button find_button;
    Button alarm_button;
    Button cancel_alarm;

    //Toast Messages
    private Toast blue_check;
    private Toast alarmset_mess;

    //Global Variables Particular To This Class
    private BluetoothAdapter deviceadp;
    private boolean alarm_found = false;
    private String alarm_mac;
    private AlarmManager alarm_setter;
    private PendingIntent alarm_intent;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        deviceadp = BluetoothAdapter.getDefaultAdapter();

        //Create sync button
        find_button = (Button) findViewById(R.id.find_button);
        find_button.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                if (deviceadp == null) {
                    blue_check = Toast.makeText(getApplicationContext(),
R.string.bluetooth_no_support, Toast.LENGTH_SHORT);

```

```

        blue_check.show();
    }
    else {
        blue_check = Toast.makeText(getApplicationContext(),
R.string.bluetooth_avail, Toast.LENGTH_SHORT);

        blue_check.show();
        Intent go_blue = new Intent(getApplicationContext(),
bluetoothSearch.class);

        startActivityForResult(go_blue, BLUETOOTH_SYNC);
    }
}
});

alarm_button = (Button) findViewById(R.id.alarm_button);
alarm_button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) { showTimePickerDialog(v); } });

cancel_alarm = (Button) findViewById(R.id.cancel_alarm);
cancel_alarm.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        // If clicked, alarm will be canceled
        if (alarm_setter != null) {
            alarm_setter.cancel(alarm_intent);
        }
        // Let user know
        alarmset_mess = Toast.makeText(getApplicationContext(),
R.string.alarm_cancel, Toast.LENGTH_SHORT);

        alarmset_mess.show();
    } });

//Bluetooth needs to be already turned on
checkBluetoothOn();
}

@Override
public void onActivityResult(int requestCode, int resultCode, Intent data) {

// If we asked user for a Bluetooth enable request
if(requestCode == REQUEST_BT_ENABLE) {
    // If Bluetooth was not enabled, ask user again
    if(resultCode != RESULT_OK){ checkBluetoothOn(); }
}

// If we are returning from bluetoothSearch after finding a MAC address
if (requestCode == BLUETOOTH_SYNC) {
    if (resultCode == RESULT_OK) {
        // Get Intent to retrieve MAC Address from bluetoothSearch
        alarm_mac = data.getStringExtra("MAC_address");

        //Should print MAC address to screen for testing
        Toast sync_mess = Toast.makeText(getApplicationContext(),
R.string.sync_done, Toast.LENGTH_SHORT);

```

```

        sync_mess.show();

        // Need to check if MAC address received is not empty
        if(!alarm_mac.isEmpty()) { alarm_found = true; }
    }
}

// Helper method, checks if Bluetooth is on using an Intent request
private void checkBluetoothOn(){

    if (!deviceadp.isEnabled()){
        Intent activateBlue = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
        startActivityForResult(activateBlue, REQUEST_BT_ENABLE);
    }
}

// Activates TimePicker
public void showTimePickerDialog(View v) {

    // Check if MAC Address is obtained, if not do not show TimePicker yet
    if(alarm_found) {
        DialogFragment newFragment = new AlarmSetter();
        newFragment.show(getSupportFragmentManager(), "alarmSetter");
    }
    else {
        Toast.makeText(getApplicationContext(), R.string.not_yet,
        Toast.LENGTH_SHORT).show();
    }
}

// Need to define sub-class to display TimePicker within MainActivity
// Non-static because we cannot call setAlarm without it
private class AlarmSetter extends DialogFragment
    implements TimePickerDialog.OnTimeSetListener {

    @Override
    @NonNull
    public Dialog onCreateDialog(Bundle savedInstanceState){

        // Use current time as default value
        final Calendar current_time = Calendar.getInstance();
        int current_hour = current_time.get(Calendar.HOUR_OF_DAY);
        int current_min = current_time.get(Calendar.MINUTE);

        //Returns instance of TimePicker with current time, 12 hour view with AM/PM
        return new TimePickerDialog(getActivity(), this, current_hour, current_min,
        false);
    }

    // Once user has said OK on the time, go to set the alarm

```

```

public void onTimeSet(TimePicker view, int hour, int minute){
    setAlarm(hour, minute);
}
}

private void setAlarm(int hour, int minute){

    // Get system service for alarms and set it to the AlarmManager
    alarm_setter = (AlarmManager)
    getApplicationContext().getSystemService(Context.ALARM_SERVICE);

    // Get a new instance of Calendar
    Calendar clock_set = Calendar.getInstance();

    // Make sure the calendar is lenient so we do not have to worry about
    changing day into the next month or week
    if (!clock_set.isLenient()){ clock_set.setLenient(true); }

    clock_set.setTimeInMillis(System.currentTimeMillis());

    // Need to check if day is the next day
    // Otherwise system will think the alarm has already passed
    if(hour < clock_set.get(Calendar.HOUR_OF_DAY)){
    int temp_day = clock_set.get(Calendar.DAY_OF_MONTH);
    temp_day++;
    clock_set.set(Calendar.DAY_OF_MONTH, temp_day);
    }

    clock_set.set(Calendar.HOUR_OF_DAY, hour);
    clock_set.set(Calendar.MINUTE, minute);
    clock_set.set(Calendar.SECOND, 0);

    // Create intent and set AlarmManager with new time
    // Intent will go to WakeReceiver, carrying along the MAC Address

    Intent temp_intent = new Intent(getApplicationContext(),
    MainActivity.WakeReceiver.class);

    temp_intent.putExtra("MAC_address", alarm_mac);
    alarm_intent = PendingIntent.getBroadcast(getApplicationContext(), 0,
    temp_intent, PendingIntent.FLAG_UPDATE_CURRENT);

    // If alarm was previously set with this application, cancel it
    alarm_setter.cancel(alarm_intent);

    // Go to separate alarm setting methods based upon current SDK
    if(Build.VERSION.SDK_INT >= 21){ setManagerNew(clock_set); } // 5.0 and later
    else{ setManagerOld(clock_set); } // 4.4 Kit-Kat

    // Notify user that alarm has been set
    alarmset_mess = Toast.makeText(getApplicationContext(), R.string.alarm_set,
    Toast.LENGTH_SHORT);
    alarmset_mess.show();
}
}

```



```

// Following two functions set the AlarmManager based on API
@TargetApi(21)
private void setManagerNew(Calendar time){
// If users want to fix, they can do so by pushing SET ALARM button
AlarmManager.AlarmClockInfo alarm = new
AlarmManager.AlarmClockInfo(time.getTimeInMillis(), null);

alarm_setter.setAlarmClock(alarm, alarm_intent);
}

private void setManagerOld(Calendar time){
alarm_setter.setExact(AlarmManager.RTC_WAKEUP, time.getTimeInMillis(),
alarm_intent);
}

// WakefulBroadcastReceiver sub-class that will activate at the right time
// Sends MAC Address to alarmReceive and starts the alarm activation process
public static class WakeReceiver extends WakefulBroadcastReceiver {

@Override
public void onReceive(Context context, Intent intent) {

BluetoothAdapter deviceadp = BluetoothAdapter.getDefaultAdapter();

// Check if Bluetooth is enabled. If not, force it on
while(!deviceadp.isEnabled()){ deviceadp.enable(); }

Intent start_wakeup = new Intent(context, alarmReceive.class);
String temp = intent.getStringExtra("MAC_address");
start_wakeup.putExtra("MAC_address", temp);
startWakefulService(context, start_wakeup);
}
}
}

```

bluetoothSearch.java –

```

package edu.union.imowers.capstonealarmfinal;
import android.app.Activity;
import android.bluetooth.*;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.os.Bundle;
import android.view.View;
import android.view.Window;

```

```

import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.Button;
import android.widget.ListView;
import android.widget.TextView;
import java.util.Set;

public class bluetoothSearch extends Activity {

private BluetoothAdapter deviceadp;
private ArrayAdapter<String> btdevicelist_curr;
private ArrayAdapter<String> btdevicelist_disc;

// BroadcastReceiver for accumulating MAC Addresses
private final BroadcastReceiver btReceiver = new BroadcastReceiver() {

public void onReceive(Context context, Intent intent) {
String action = intent.getAction();
if (BluetoothDevice.ACTION_FOUND.equals(action)) {
// Get the BluetoothDevice object from the Intent
BluetoothDevice device =
intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);

if (device.getBondState() != BluetoothDevice.BOND_BONDED) {
// Add the name and address to an array adapter to show later
btdevicelist_disc.add(device.getName() + "\n" + device.getAddress());
}
}

// When discovery is finished, check if any devices found
else if (BluetoothAdapter.ACTION_DISCOVERY_FINISHED.equals(action)) {
if (btdevicelist_disc.getCount() == 0) {
String noDevices = getResources().getText(R.string.nofound).toString();
btdevicelist_disc.add(noDevices);
}
}
}
};

// The click listener for all devices in the device list
private AdapterView.OnItemClickListener deviceselect = new
AdapterView.OnItemClickListener() {
public void onItemClick(AdapterView<?> av, View v, int arg2, long arg3) {
String noPaired = getResources().getText(R.string.nopair).toString();
String noFound = getResources().getText(R.string.nofound).toString();

// Cancel discovery since device has been selected
deviceadp.cancelDiscovery();
String info = ((TextView) v).getText().toString();

// If nothing found, DO NOT continue forward with splicing MAC Address
// If any devices have been found, then proceed
if ((info != noPaired) && (info != noFound)){
//Take the selected MAC Address and save it to connect

```

```

        // Send this information back to MainActivity
        if (info.length() >= 17) {
            String address = info.substring(info.length() - 17);
            //Sending MAC Address back to main activity
            Intent return_intent = getIntent();
            return_intent.putExtra("MAC_address", address);
            setResult(RESULT_OK, return_intent);
        }
        else {
            // MAC Address splicing did not work, sure-fire check because our
            // strings noPaired and noFound are automatically smaller
            setResult(RESULT_CANCELED, getIntent());
        }
        finish();
    }
}
};

```

```

// Method that runs when activity started
@Override
protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);

    // Put Device List in a Window
    requestWindowFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
    setContentView(R.layout.activity_bluetooth_search);

    //Get default adapter since we cannot pass it through an Intent
    deviceadp = BluetoothAdapter.getDefaultAdapter();

    //If user backs out at any point before discovery starts
    setResult(RESULT_CANCELED);

    // Register the BroadcastReceiver for Action Found and Discovery Finished
    IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
    registerReceiver(btReceiver, filter);
    filter = new IntentFilter(BluetoothAdapter.ACTION_DISCOVERY_FINISHED);
    registerReceiver(btReceiver, filter);

    // Set up Array Adapters for displaying current and discovered items
    btdevicelist_curr = new ArrayAdapter<>(this, R.layout.device_name);
    btdevicelist_disc = new ArrayAdapter<>(this, R.layout.device_name);

    //Set up the clickable ListViews so devices can be added during discovery
    ListView pairedListView = (ListView) findViewById(R.id.paired_list);
    pairedListView.setAdapter(btdevicelist_curr);
    pairedListView.setOnItemClickListener(deviceselect);
    ListView newDevicesListView = (ListView) findViewById(R.id.new_list);
    newDevicesListView.setAdapter(btdevicelist_disc);
    newDevicesListView.setOnItemClickListener(deviceselect);

    // Go get current devices
    getcurrentDevices();
}

```

```

// Initialize button to perform device discovery
Button scanButton = (Button) findViewById(R.id.begin_scan);
scanButton.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        //Do device discovery, which will automatically add to ListView
        discoverDevices();
        v.setVisibility(View.GONE);
    }
});
}

//Check to see if the device has already been previously paired
private void getcurrentDevices(){
Set<BluetoothDevice> pairedDevices = deviceadp.getBondedDevices();
if(pairedDevices.size() > 0) {
    findViewById(R.id.title_paired_list).setVisibility(View.VISIBLE);
    for (BluetoothDevice device : pairedDevices) {
        btdevicelist_curr.add(device.getName() + "\n" +
device.getAddress());
    }
}
else{
//No devices that have been paired usable thus just put no paired string
    btdevicelist_curr.add(getText(R.string.nopair).toString());
}
}

// Initiate device discovery
private void discoverDevices(){
findViewById(R.id.title_new_list).setVisibility(View.VISIBLE);
deviceadp.startDiscovery();
}

// What to do when user closes the app
@Override
public void onDestroy(){
    super.onDestroy();
    //Ensure discovery is off
    if(deviceadp != null){ deviceadp.cancelDiscovery(); }

    // UnregisterReceiver to ensure that device stops searching
    // Why the alarm receiver must be restarted/turned off before reuse
    unregisterReceiver(btReceiver);
}
}

```

alarmReceive.java –

```

package edu.union.imowers.capstonealarmfinal;
import android.app.IntentService;

```

```

import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.content.Intent;
import android.support.v4.content.WakefulBroadcastReceiver;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.UUID;
public class alarmReceive extends IntentService {

    // Constants, in this case the UUID for Bluetooth
    private static final UUID alarmReceiveClass_UUID = UUID.fromString("00001101-
0000-1000-8000-00805F9B34FB");

    // Globals
    private BluetoothAdapter deviceadp;
    ConnectThread alarm_connect;
    ConnectedThread alarm_comm;

    //Instantiate message to send to alarm
    byte[] send_mess = {(byte) '1'};

    public alarmReceive() {
        super("alarmReceive");
    }

    @Override
    protected void onHandleIntent(Intent alarm_intent) {

        // Get local bluetooth adapter
        deviceadp = BluetoothAdapter.getDefaultAdapter();

        // Get MAC Address from Intent, needed to start socket
        String MAC_Address = alarm_intent.getStringExtra("MAC_address");

        // Start connection
        alarm_connect = new ConnectThread(deviceadp.getRemoteDevice(MAC_Address));

        // Start connected thread to alarm receiver
        alarm_connect.run();

        // Finish
        WakefulBroadcastReceiver.completeWakefulIntent(alarm_intent);
    }

    //Class to initiate connection with Bluetooth console
    private class ConnectThread extends Thread{

        private final BluetoothSocket alarm_socket;
        private final BluetoothDevice alarm;

        public ConnectThread(BluetoothDevice device){

```

```

BluetoothSocket temp = null;
alarm = device;
// Need to check if we can create a RfCommSocket, since socket is final
try{
temp = alarm.createRfcommSocketToServiceRecord(alarmReceiveClass_UUID);
} catch(IOException e){}
alarm_socket = temp;
}

public void run(){

// Turn off discovery first, but it should be off anyway
deviceadp.cancelDiscovery();

// Attempt to connect to device
try { alarm_socket.connect();}
catch (IOException connectException){
try { alarm_socket.close(); }
catch (IOException closeException){}
return;
}
// Method to start the alarm and manage the connection
startAlarm(alarm_socket);
}
}

```

```

//Class to maintain connection with Bluetooth console
private class ConnectedThread extends Thread {

private final BluetoothSocket alarm_socket;
private final InputStream in_stream;
private final OutputStream out_stream;

// Constructor to make thread
public ConnectedThread(BluetoothSocket socket){
alarm_socket = socket;
InputStream temp_in = null;
OutputStream temp_out = null;
// Must check if we can obtain IO Streams, final variables
try {
temp_in = alarm_socket.getInputStream();
temp_out = alarm_socket.getOutputStream();
}catch (IOException e){} // If problem, don't do anything
in_stream = temp_in;
out_stream = temp_out;
}

public void run(){
byte[] buffer = new byte[1024]; // Set buffer for IO streams
int read_stream; // Set of bytes read from the read() call
while (true){
try{
read_stream = in_stream.read();

```

```

        // This does nothing, just standard practice
    }catch (IOException e){
        break; // IO stream has been interrupted
    }
}

// Write to output stream, sending data to alarm
// Implemented because we can choose what to send out
public void write(byte[] bytes){
    try{
        out_stream.write(bytes);
        out_stream.flush(); // Clean up stream when finished
    }catch(IOException e){} // If problem, no nothing and try again
}

// Close socket when done to clean up internal resources
public void cancel() {
    try { alarm_socket.close(); }
    catch (IOException e) { }
}
}

public void startAlarm(BluetoothSocket alarm_socket){
    // Create socket
    alarm_comm = new ConnectedThread(alarm_socket);

    // Send preset message to alarm
    alarm_comm.write(send_mess);

    // Do not need to check message received from alarm
    // Alarm will automatically turn off
    alarm_comm.cancel();
}
}
}

```

strings.xml – (Resource list of strings referred to in main code)

```

<resources>
<string name="app_name">Ian Mowers Capstone Project</string>
<string name="app_title">Bluetooth Remote Wakeup Alarm</string>
<string name="app_subtitle">Ian Mowers Capstone Project 2016</string>
<string name="title_activity_bluetooth_search">bluetoothSearch</string>
<string name="title_activity_bluetooth">bluetooth</string>
<string name="bluetooth_avail">Bluetooth Supported, Searching for Alarm
Device</string>
<string name="bluetooth_no_support">Bluetooth not supported on this
device</string>
<string name="blue_en">Bluetooth Enabled</string>
<string name="not_yet">Alarm Not Found Yet, Please SYNC first</string>
<string name="nopair">Alarm not paired</string>

```

```
<string name="nofound">Alarm not found</string>
<string name="sync_done">Alarm Found and Saved</string>
<string name="alarm_set">Alarm Time Set</string>
<string name="alarm_cancel">Alarm Setting Canceled</string>
<string name="title_activity_alarm_receive">alarmReceive</string>
<string name="MAC">MAC_address</string>
<string name="find_button">FIND ALARM</string>
<string name="alarm_button">SET ALARM</string>
<string name="cancel_alarm">CANCEL ALARM</string>
<string name="paired_list">Previously Paired Devices:</string>
<string name="new_list">New Devices:</string>
<string name="scan_button">Scan For Alarm</string>
</resources>
```