# LOW-POWER SYSTEM-ON-CHIP (SOC) IMPLEMENTATION FOR SELF-SUFFICIENT WIRELESS RESPIRATORY DATA AND ENERGY HARVESTING SYSTEM

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAIʻI AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

DECEMBER 2011

By

Roxanne K. K. F. Yee

Thesis Committee:

Olga Boric-Lubecke, Chairperson
Tep Dobry
Galen Sasaki
Xiangrong Zhou

# Dedication

For my parents and grandparents…

*"Brothers and sisters: Have no anxiety at all, but in everything, by prayer and petition, with thanksgiving, make your requests known to God. Then the peace of God that surpasses all understanding will guard your hearts and minds in Christ Jesus. Finally, brothers and sisters, whatever is true, whatever is honorable, whatever is just, whatever is pure, whatever is lovely, whatever is gracious, if there is any excellence and if there is anything worthy of praise, think about these things. Keep on doing what you have learned and received and heard and seen in me. Then the God of peace will be with you."*

*~ Philippians 4: 6 – 9*

# Acknowledgements

I would like to extend a great amount of gratitude to Pronk Technologies Inc. and all of its employees for the use of equipment and for their understanding and intellectual insight, especially Karl Ruiter and Mike Rosenman. Another influential group of people that provided input and help to me were my advisor and thesis chairperson, Dr. Boric-Lubecke, and all of the students whom she advises, particularly Wansuree Massagram, Bryson Padasdao, and Chenyan Song. I am also much obliged to Noah Hafner and Jeremy Chan for their technical guidance. Furthermore, my thesis committee members, Dr. Tep Dobry, Dr. Galen Sasaki, and Dr. Xiangrong Zhou, deserve recognition for their caring comments and suggestions. I'd also like to give credit to my high school freshman English teacher, Ms. Stephanie Darrow, for convincing me to keep my *Writer's Inc.* Lastly, but most importantly, I wish to say a special "Thank you!" and convey my love to all of my family and friends, especially Sky Kiyabu, Jeremy Porter, and Kristina Wong, for their love, support, and encouragement. I couldn't have done it without all of you! Thank you!

# Abstract

Recent studies and systems have made it possible to measure respiratory rate comfortably and non-invasively; however, these methods do not particularly focus on their convenience. Specifically, either patient mobility is decreased because they are confined within the respiratory sensor's effective range or the wireless apparatus attached to them necessitates battery replacements every few days. A solution to this problem is to utilize the attachment solution, but remove the battery component. The overall project of this thesis proposes that this can be done by using the respiratory effort itself to power the apparatus. However, with current portable harvesting technology, respiratory effort does not yield more than a couple milliwatts of power at best, thus the apparatus must be optimized for minimum power consumption. Beyond low-power hardware selection, lies the need for low-power software implementation of the respiratory signal's digital processing. This thesis investigates a handful of algorithms and their suitability for this power-restricted environment, and ultimately demonstrates the feasibility of measuring respiratory rate with less than 100 μW of average power by utilizing a low-power system-on-chip. Rather than utilizing the chip's analog-to-digital converter, as is normal convention, a simple comparator approach was adopted. Although rather simplistic and atypical, the minimalistic attitude towards signal processing presented in this thesis is necessary for the development of systems tailored to incredibly low-power environments.

# Table of Contents

# List of Tables

# List of Figures

# 1. Introduction

Strong evidence suggests that an abnormal respiratory rate is a precursor to serious events, such as the immediate admission to an intensive care unit (ICU), and fatal health conditions, such as cardiac arrest. For the former event, respiratory rate has even been considered a more reliable indicator than blood pressure and heart rate, and for the latter, it has been recognized as being able to predict the event up to 72 hours or more in advance. [**1**,**2**]

However, until recently and despite its stature as one of humanity's four major vital signs, respiratory rate has not been given the same amount of attention by medical personnel as the other three vital signs of blood pressure, heart rate, and temperature [**3**]. Such neglect could be due to the fact that there was not an easy and quick way to non-invasively measure it in the past. Whatever the reason may be, obtaining an accurate, non-invasive respiratory rate measurement has become a popular engineering query within recent years. Regardless of many advances, several of these non-invasive methods require that the patient sacrifice mobility or convenience; for instance, they are obligated to reside within range of a large immovable sensor; or, if the patient is able to move, they must attach a unit to themselves that might require its battery to be refreshed every couple of days.

Remaining within a certain area or changing or recharging a battery every so often is neither convenient nor comfortable. The guiding research project behind this thesis's study has proposed a solution to this problem by suggesting the utilization of human energy harvesting to power a wearable wireless unit. Normally, this might be associated with a person jogging around or participating in another event that requires active physical exertion to produce power; however, the overall project encompassing this study proposes the passive harvesting of energy, where the respiratory effort itself will be used to power the system used to measure it.

Such a self-sustaining system requires that the combined components of the system use less energy than what can be produced by the subject being harvested. Therefore, when designing a system powered by a low-energy source, such as respiratory

effort, each component must be analyzed and designed for its maximum power efficiency. One of these components is the digital processing unit, which is expected to calculate the respiratory rate from the respiratory signal and transmit it wirelessly to a base station. Power optimization for this unit is highly dependent on the algorithm it uses to process data as well as the implementation of that algorithm; it is this algorithmic element that is the topic of this thesis.

## 1.1. Thesis Organization

This rest of this thesis is organized generally in chronological order, where the remainder of Chapter 1 provides background information about the medical, energy harvesting, and overall project aspects of this study and also details the specific goals of this thesis. In Chapter 2, a handful of algorithms are considered for implementation, and two were selected to become the center of this study. Details of the implementations and their limitations are covered in Chapter 3, and the power consumption measurements and calculations are presented in Chapter 4. The conclusion and directions of future work reside in Chapter 5, and following them are the appendices, which contain code, figures, calculations, encountered problems and issues, and terminology clarifications.

## 1.2. Background

Many different fields of interest are united by this study, each with their own importance and contribution. This subchapter provides background material on these fields, which include the medical and energy harvesting aspects of this study and also the overall project that this study is a part of.

### 1.2.1. Medical Background

The end product to which this thesis's results will apply is ultimately a device that will be used in the medical field to measure respiratory rate. The following two subchapters discuss the importance of this, sometimes overlooked, vital sign and explain the current methods and technologies utilized to measure it.

#### *1.2.1.1. Respiratory Rate*

Often a glazed-over vital sign, respiratory rate, abbreviated as RR, is the rate at which a person breathes. It is normally measured in breaths per minute, notated as br/min in this study. Accurate monitoring of RR can indicate respiratory dysfunction, disease, general ailments, or the increasing potential for a life-threatening event, such as cardiac arrest; however, despite its significance to a patient's health, RR is not generally appreciated in the medical community as much as other vitals signs, such as heart rate [**3**,**4**]. This disenchantment with RR might be caused by the great variability in the breathing action, whether it is a variation from person-to-person or a variation from breath-to-breath. To acquire an idea of the vastness of these variations, Table 1.1 provides descriptions and illustrations of several types of breathing patterns.

**Table 1.1: Breathing Patterns [5,6].**

| Breathing Pattern | Description | Illustration |
|---|---|---|
| Normal Breathing | Breathing within the following ranges (br/min):<br>• Newborn: 30 – 60<br>• Early Childhood: 20 – 40<br>• Late Childhood: 15 – 25<br>• Adult: 14 – 20 |  |
| Apnea | A period of 10 seconds or more without breathing. |  |
| Tachypnea | Rapid shallow breathing with the following ranges (br/min):<br>• Infants 0 – 2 months: > 60<br>• Infants 2 – 12 months: > 50<br>• Children > 12 months: > 40 |  |
| Hyperpnea (Hyperventilation) | Rapid deep breathing. |  |
| Bradypnea | Slow breathing. |  |
| Cheyne-Stokes Breathing | Deep breathing interspersed with apnea. |  |
| Ataxic Breathing (Biot's Breathing) | Unpredictable irregular breathing, which can be deep, shallow, and/or interspersed with apnea. |  |
| Sighing Respiration | Breathing punctuated by sighs. |  |
| Obstructive Breathing | Breathing caused by blockage or tightening of airways. |  |

### *1.2.1.2. Modern Technology*

Current technology used for measuring respiratory rate can be divided into two categories, direct and indirect, depending on the type of sensor utilized to detect a breath. The former indicates that a breath will be sensed by placing the sensor within the patient's airway, whereas, the latter is inclusive for all other devices which do not come into contact with the patient's airflow [**4**]. Table 1.2 contains a short list of some of these current sensor technologies, provides a brief description of how each functions, and specifies whether the sensor is direct or indirect.

**Table 1.2: Modern Sensor Technology Used To Measure Respiratory Rate.**

| **Sensor Technology** | **Description** | **Direct/ Indirect** |
|---|---|---|
| Pneumotachometer | When certain-shaped objects are placed into the airway, they can create known differences in pressure related to the airflow through and/or around them. In the case of a pneumotachometer, a pressure sensor, positioned in the airway, measures pressure differences caused by the flow pattern and translates them into signals ready for electrical processing. The relationship between the flow and pressure difference may be linear or nonlinear, depending on the exact sensor used [**4**]. | Direct |
| Temperature Sensor | Airflow into the body is usually cooler than that heading outward. With a temperature sensor, this inhale versus exhale temperature difference is used to electrically distinguish each breath [**4**]. | Direct |
| Humidity Sensor | When a patient breathes, air exhaled has a higher humidity than air inhaled. In recent research, the intensity of this humidity was monitored based on the water condensation, absorption, and evaporation around the humidity sensor [**7**]. | Direct |
| Carbon Dioxide ($CO_2$) Monitor | Under normal circumstances, the amount of $CO_2$ expelled from the body is less than the amount admitted. Measuring $CO_2$ concentration levels can assist in monitoring a patient's breathing patterns by converting the amount of $CO_2$ detected into electrical signals for processing [**4**]. | Direct |
| Displacement Sensor | While breathing, a patient's chest and abdomen expands and contracts. The displacement between the fully expanded and fully contracted states of these body parts can be transformed by a displacement sensor signal into a signal convenient for electrical processing. This category of sensor is very broad and encompasses strain gages, piezoelectric sensors, linear variable differential transformers, and variable reluctance magnetic sensors [**4**,**8**]. | Indirect |
| Inductance Plethysmograph | Similar to displacement sensors, inductance plethysmographs depend on the action of the patient's chest or abdomen expanding or contracting when breathing. However, instead of measuring the actual displacement, the plethysmograph electronically measures the change of inductance of a wire looped around the chest or abdomen. The difference between an inductance plethsymograph and a general displacement sensor is that the former requires a *loop of wire* for proper functionality [**4**]. | Indirect |

**Table 1.2 (Continued): Modern Sensor Technology Used To Measure Respiratory Rate.**

| Sensor Technology | Description | Direct/ Indirect |
|---|---|---|
| Transthoracic Electrical Impedance Sensor | Every breath a patient takes alters the volume of air within their lungs. This alteration changes the electrical impedance of the lungs, which can be measured electronically, usually through the same electrodes used for an electrocardiogram (ECG) measurement [**4**,**9**]. | Indirect |
| Acoustic Sensor | As a patient breathes, the flow of air causes low-frequency sound waves. With acoustic sensors, these "breathing sounds" can be characterized into an electrical signal ready for analysis [**4**,**10**]. | Indirect |
| Contact Motion Sensor | Movement of the chest or abdomen while breathing can be detected through various types of sensors provided they are in contact with the patient. Some examples are a pressure-sensing pad [**4**], which is placed under the patient, and an accelerometer [**11**], which is mounted to the patient's torso. These motions are interpreted for breathing patterns through electrical processing. | Indirect |
| Non-Contact Motion Sensor | Unlike contact motion sensors, non-contact motion sensors do not rely on any physical connection to the patient. In recent years, there have been several studies into using electromagnetic waves, such as microwaves and radio waves, to capture the breathing motion. Some of these methods rely on characterizing the motion using the Doppler Effect [**12**,**13**,**14**,**15**,**16**], whereas another very recent method encircles the patient with a network of off-the-shelf transceivers and measures how chest and abdominal motion impedes the radio signals [**17**,**18**,**19**]. | Indirect |

Technological development of respiratory monitors has tended toward utilizing indirect sensors because they are less invasive, less obtrusive, less obstructive, and, thus, more comfortable to the patient. This effort for a more pleasing hospital stay has also prompted devices that do not require the patient to be wired to large pieces of medical equipment. Current implementations of these wire-free technologies include systems that either require the patient to have a small local unit attached to them that transmits data to a larger apparatus, or require the patient to remain within the effective area of one of the non-contact motion sensors mentioned in Table 1.2.

"Wire-free" technology for any medical device does not necessarily imply "wireless"; for instance, some devices may act as data loggers by locally storing data. After collection, this data must be uploaded to a more complex monitoring apparatus through a physical connection for proper access. Implementation of wire-free devices can be separated into two categories: non-portable and portable. In general, wire-free devices, as any other medical device, have four basic data handling and processing stages: sensing, signal conversion, interpretation, and display.

For non-portable devices, the sequence of the stages is straightforward; the non-contact sensor translates a physical trait into an electrical signal, the signal is converted from the analog to the digital domain, the data is interpreted for specific features, and the pertinent information is displayed for human readability. All of these stages are executed by a single device in real-time without much deviation, as illustrated in Figure 1.1, and there remains a great amount of freedom as to what signal features the device interprets. For the specific case of respiration, the system may desire to assess all or any combination of respiration features, such as breathing pattern, tidal volume (i.e., amplitude), and respiratory rate.

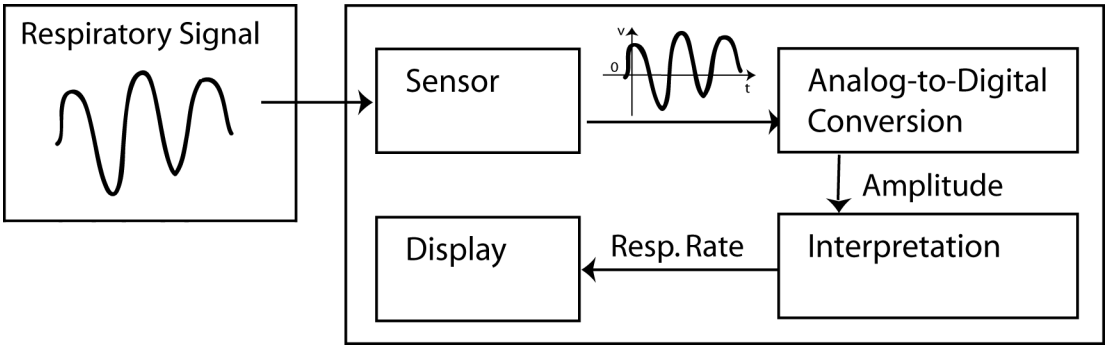


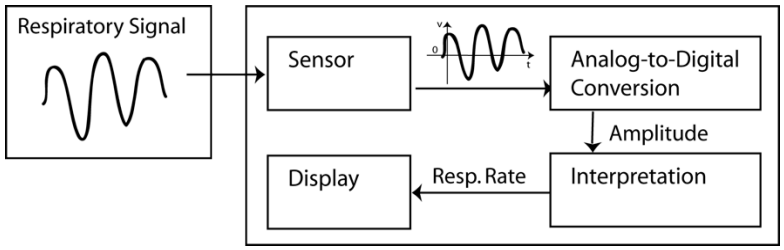
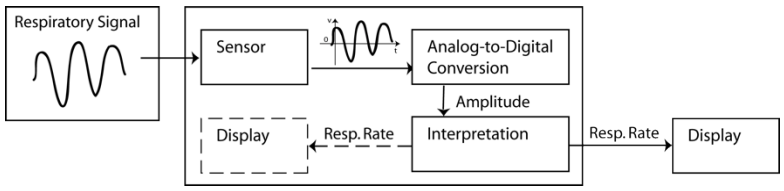
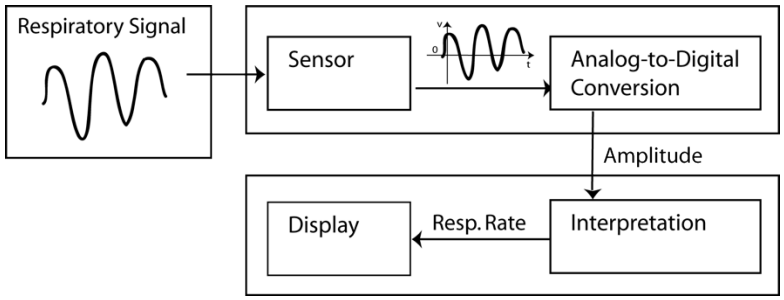
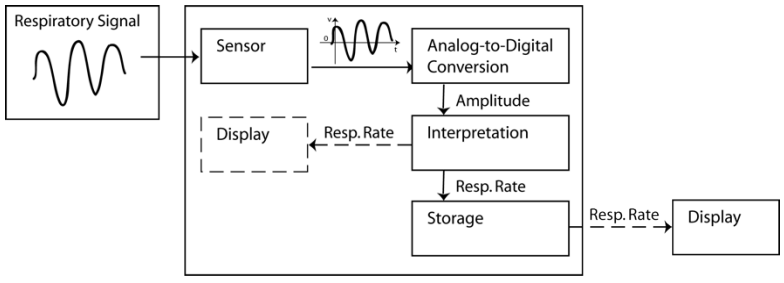

**Figure 1.1: Non-portable "Wire-free" Device Data Handling and Processing Stage Diagram**

On the contrary, portable devices handle information in several different implementation flavors due to their unique circumstance of limited power availability. Most portable devices generally function as a sensor node and are partnered with a non-portable device that handles the more complex operations. Although the data handling and processing stages remain the same as non-portable devices, portable data handling and processing stages are sometimes executed out of their typical order (ex., the physician looks at a display and does the interpretation himself or herself) or have their complexity sacrificed based on power-saving techniques and the device's specific functionality requirements (ex., only the respiratory rate is obtained versus the breathing pattern).

More specifically, portable device implementations differ with regard to the freshness of data, the fashion in which data is displayed, and at what point in the information flow the data is interpreted and/or stored. With regards to respiratory rate as

the data of interest, some devices will either display the rate value in real-time, display it after-the-fact, or utilize a combination of the two. The display itself can either be on a local unit, on a separate unit linked by physical connection or wireless technology, or on both. The rate value itself can either be extracted from the waveform and interpreted locally, sent to another unit for processing, or partially handled locally and partially handled remotely. There are several combinations that can be made, and Table 1.3 contains some depictions of common ones; Dotted lines notate optional or temporary connections. Particularly, this thesis's overall project plans to realize the second implementation presented in the table.

**Table 1.3: Portable Device Stage Data Handling and Processing Variations.**

| Implementation | Diagram |
|---|---|
| Real-time analog information is locally converted to digital information and displayed locally. |  |
| Real-time analog information is locally interpreted and transmitted to a separate display. |  |
| Real-time analog information is transmitted to a separate unit for interpretation and display. |  |
| Analog information is locally interpreted and stored for later viewing on a separate display. |  |

Despite the advantage of comfort, being wire-free comes at a slight cost: patient mobility and convenience. The issue with current iterations of wire-free technology is that the patient either needs to be bed-ridden or have their local unit's battery replenished every couple of days [**9**]. Constant battery replacement or recharging is quite inconvenient in particular situations, especially if the device is used in an at-home monitoring application. A relatively new solution that is under research and addresses the

problem of decreased mobility and convenience is the objective of this thesis's overall project. Details of this project are discussed in Subchapter 1.2.3.1.

### 1.2.2. Energy Harvesting Background

Energy harvesting can be defined as utilizing ambient energy provided by natural occurrences. Some well-known examples include solar energy, wind energy, and hydropower [**20**]. Recently, mankind has enhanced its determination to utilize these kinds of energy resources as opposed to finite fuels, which cannot be replenished within a lifetime. Part of this effort is mankind's desire to reduce portable-device battery waste by utilizing what Texas Instruments Incorporated (TI) calls "micro-harvesting". Unlike solar energy and other resources that are "macro-harvested", micro-harvested energy is produced from relatively feint activities, such as vibrations and body heat [**21**]. This sort of harvesting is *required* for wireless devices that do not have the option of being connected to a large generator.

### 1.2.3. Study Background

The scope of this study is limited to a subsystem of a more complex overall project. The relationship and goals of both the overall project and the subsystem are described in the following subchapters.

#### *1.2.3.1. Overall System Overview*

The goal of the overall project presiding over this study is to develop a wireless, battery-free monitoring unit powered by the physiological signals that it monitors [**22**]. In particular, the project concentrates on harvesting a person's respiratory effort and electrocardiogram potentials. The monitoring "on-patient" unit will also have a sister "off-patient" unit to which it will communicate data utilizing radio frequency (RF) technology. This method of "zero-net energy" monitoring allows for the patient to move around without wires and removes the need for frequent battery replacement or recharging. It, thus, avoids the issues with modern respiratory monitoring technology mentioned in Subchapter 1.2.1.2.

For the "on-patient" unit, a passive displacement sensor and electrode-array will be used to convert the respiratory effort and biopotentials, respectively, into two analog

voltage waveforms. These waveforms will then be directed to a power module, which will rectify the signal into direct current (DC), making it usable by the DC components of the system. Simultaneously, the waveforms are also conditioned and fed to a data processing module, most likely a digital signal processor (DSP) or a system-on-chip (SoC), which prepares the analog signal for digital processing, digitally processes the signal, and sends the pertinent data to an RF radio that handles the wireless transmission to the "off-patient" unit [**23**].

The end product should be a system in which the monitored patient is comfortable and does not feel inconvenienced by the monitoring unit; it is planned that the "on-patient" unit will be sewn into clothing that will generate the power required to monitor the wearer [**24**,**25**,**26**,**27**]. Also, the "off-patient" unit will most likely be integrated into a larger health monitoring system for convenience and to enhance mobility [**28**,**27**]. Figure 1.2 depicts a block diagram of the overall system with respect to the respiratory rate monitoring portion. This particular aspect is the premise of this thesis, and the greyed SoC is the digital processing unit, which will be the focus of this study. Details concerning the SoC are discussed in the next subchapter.

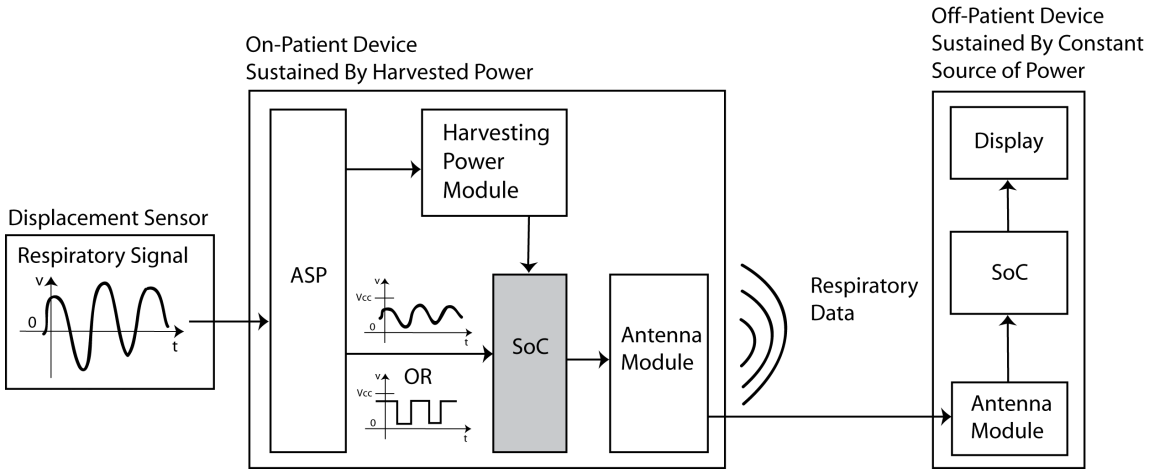



**Figure 1.2: Overall Project System Block Diagram With Respect To The Respiratory Monitoring Portion.**

### *1.2.3.2. Subsystem Overview*

As mentioned in the previous subchapter, this study pertains to the digital data processing unit of the respiratory rate monitoring subsystem of an overall project. More

specifically, the module of interest is the SoC utilized to digitally extract the respiratory data from the output of the signal-conditioning ASP (analog signal processor).

Like every other "on-patient" component of the overall system, the SoC must function while drawing the least amount of power in order to preserve and guarantee functionality if the patient's breathing pattern begins to yield less energy. Typically, power optimization for wireless biological monitoring systems is achieved through the use and/or development of low power hardware components. These systems include architectures that involve unconventional wireless circuitry and/or protocol [**29**] or an application specific integrated circuit (ASIC) [**30**,**31**,**32**].

However, unlike the other "on-patient" components, the SoC must not only be optimal in hardware efficiency; it must also have optimal software efficiency. Even if a top-of-the-line-energy-efficient SoC is utilized, its value is harshly degraded if the software logic is not coded with power conservation in mind. Published wireless medical software techniques acquire low power consumption in at least one of two ways: (1) avoidance of wireless transmissions, and (2) adaptive sampling and power allocation.

With current technology, the action of wireless transmission has communally been deemed the most power inefficient component of wireless applications. Thus, to avoid this action, systems tend to implement either a data compression technique and/or depend on infrequent transmissions. Data compression relies particularly on extracting and transmitting only pertinent information from the biological signal [**33**,**34**,**35**,**36**], whereas infrequent transmission may imply an adaptive transmission rate [**37**,**38**], transmission on-demand [**39**], or simply transmission at a slower rate.

Other inventive methods of minimizing power consumption include monitoring the system's and/or input signal's current state in order to execute dynamic sampling and power allocation. Specific features such as signal curvature [**40**] or the surpassing of a threshold can be used as indicators to alter the way the system functions. Some systems implement dynamic voltage scaling (DVS), in which clock frequency and supply voltage are allocated based on computational load [**39**]. An interesting wireless non-medical low-power application developed a magnetic disturbance wake-up [**41**], which could also be applicable to other applications.

Since this thesis focuses on the efficiency of an SoC, it concentrates on the varying aspect of software versus the concrete nature hardware. Therefore, the major goals of this study are: (1) to explore the power efficiency of simple respiratory rate monitoring algorithms and evaluate which ones are efficient enough to warrant implementation on the "on-patient" unit of the overall project, and (2) to implement those suitable algorithms in the most energy efficient manner.

The SoC chosen for the overall project was Texas Instrument's (TI) CC430 chip, which combines the company's low-power MSP430 microcontroller with its low-power CC1101 sub-1-GHz RF transceiver [**42**]. In order to become acquainted with the CC340's capabilities regarding low-power processing and wireless communication, an evaluation kit was ordered. The kit came with two EM430F6137RF900 evaluation boards and two 868/915 MHz antennas; however, an additional purchase of the MSP-FET430UIF was required for programming the chip via the JTAG interface [**43**]. Informational materials, such as the CC430's datasheet and user's guide, were found on the CC430's product page [**44**]. Also available on the product page were code examples and schematic symbols and footprints. Additionally, the EM430F6137RF900 webpage [**45**] contained a schematic of the evaluation board as well as CC430 RF examples and design guides.

A simplified block diagram of the test subsystem is depicted in Figure 1.3. Since patient participation was unnecessary for this study, the EM430F6137RF900 substituting as the "on-patient" unit was deemed the "transmitter", and the EM430F6137RF900 substituting as the "off-patient" unit was deemed the "receiver". These designations were based on each board's assigned unidirectional wireless function in the subsystem. The reasons for this unidirectional communication assumption are discussed in Subchapter 3.1.

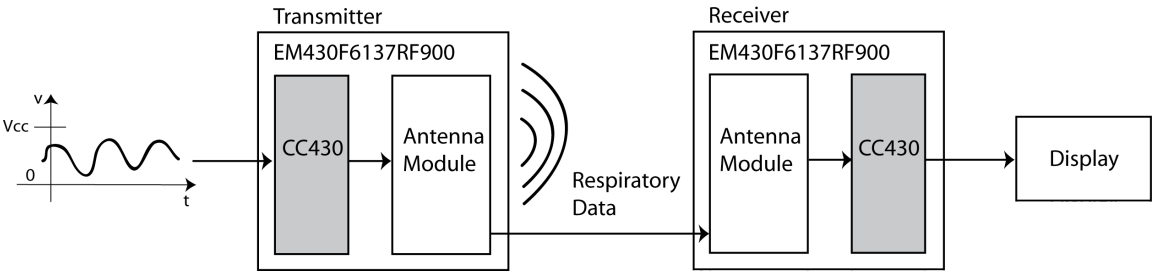



**Figure 1.3: Study Subsystem Block Diagram.**

The general flow of the subsystem begins with the transmitter SoC extracting pertinent data from the conditioned respiratory signal it receives from the overall system ASP. After digital extraction, the data is then sent to the antenna module for transmission via RF waves. Following the reception of the data through its corresponding antenna module, the receiver SoC prepares the data for human readability and forwards the information to a display.

As previously mentioned, this study's primary interest lies in evaluating the power efficiency of a few simple respiratory rate monitoring algorithms and their implementations. In particular, these transmitter SoC algorithms are composed of two parts: the digital extraction logic and the RF settings and packet handling. Unfortunately, the RF portion of the algorithm depends heavily on what antenna hardware will be used, what range of RF communication is required, and what wireless protocol must be implemented; all these parameters have yet to be solidified in the overall project. Therefore, this study only concentrates on optimizing the digital extraction portion of the algorithm. Fortunately, the RF portion and the digital extraction portion of the algorithm can be isolated from each other by proper modularization of the code. This isolation is explained further in Chapter 3, but it essentially means that the transmission related lines of the code were commented out when measuring the energy characteristics of the digital extraction portions and left in when the code needed to be functionally validated.

Although this thesis focuses on minimizing the power consumption of the digital extraction portion, power measurements for both the digital extraction portion and the RF portion were conducted. Measuring the power consumption of the RF portion is unnecessary in relation to optimizing the digital extraction portion; however, it was conducted in order to perform two things: (1) illustrate that the RF portion is the

dominant energy hoarder, and (2) provide a baseline overall picture of the order of magnitude of power required for full SoC functionality. Thus, the goal of checking the feasibility of using the CC430 is achieved by acquiring these measurements in Section 4.

In addition to illustrating the CC430's suitability for the overall project, this thesis proposes the utilization of the SoC's comparator module rather than its ADC (analog-to-digital converter) module as its analog interface to the respiratory signal. This selection is atypical of modern low-power portable wire-free respiratory monitoring systems since much information can be obtained with an ADC analog interface. In fact, it is also uncommon for the more commonly measured vital sign of heart rate [**46**,**47**,**48**].

However, in some cases, the decision to use an ADC may not be left to the developer; many of the current microcontroller-based systems are fundamentally *bound* to utilizing the ADC, since the particular sensor and/or technique that they use produces noisy or asymmetric data that is difficult to condense into a sensible digital representation by any other readily available means. Some examples of sensors that do not easily loan themselves to utilizing a simple comparator are acoustic sensors [**49**,**50**,**51**], motion sensors, temperature sensors [**52**], and humidity sensors [**7**].

Other studies that used sensors with less noise such as inductance plethysmographs and displacement sensors did not specifically discuss their reasoning for utilizing the ADC in their respiratory signal processing [**53**,**54**]; one cause may be that it is assumed obvious that their application environment is too prone to noise and requires intensive algorithmic filtering for accurate measurement, while another cause may be that there is no other analog interface within their digital processing chip.

In addition to the uncontrollable aspects, which may govern the selection of the ADC, there also exist the functional requirements of the system; for example, the main purpose of the system may not be to capture the rate of a signal, but it may be to capture the actual waveform [**55**]. Although the reasons for utilizing the ADC may be vast and/or unclear, its prominence in mixed signal processing is unquestionable.

This thesis's overall system has a low power budget and is currently only concerned with the rate of respiration and not the tidal volume or respiratory waveform shape; therefore, instead of requiring an accurate capture of respiratory amplitude provided by an ADC, the distinction between one breath and another is all that is needed.

In this sense, there are only two states of the respiratory signal that must be tracked: (1) a breath has been detected, and (2) a breath has not been detected. This binary trait is what prompts the proposed use of the comparator versus the ADC.

The general idea is that since the comparator only does a single comparison with little computational effort, it will utilize less power, making it the more ideal option for analog-to-digital conversion in this thesis's particular situation. It essentially operates as 1-bit ADC. Successful use of a comparator as a rate detector for a biological signal has been previously done for both heart rate and respiratory rate [**8**,**56**]. However, as opposed to this thesis, both studies did not elaborate on how the rate is extracted from the digitized output of the comparator, did not explain power dependency on the algorithm, and did not utilize a comparator that is embedded into a microcontroller. There is some literature that seems to imply that it used the on-chip comparator to measure the RR-interval (i.e., time between two R-waves) of a heart signal in a similar fashion to one of the algorithms in this thesis [**37**]; however, other literary sources of implementations using the comparator to monitor specifically respiratory rate are scarce.

In addition to analyzing the connection between power consumption and algorithmic complexity, this thesis wishes to illustrate utilization of the comparator already embedded in the SoC's microcontroller to decrease the number of components in the respiratory monitoring system. Figure 1.4 illustrates how this thesis's implementation will differ from previous published work; the first diagram depicts the ADC implementation that most modern technology uses, the second diagram depicts the previous implementation with the comparator, and the third diagrams depicts the implementation that will be pursued in this study.

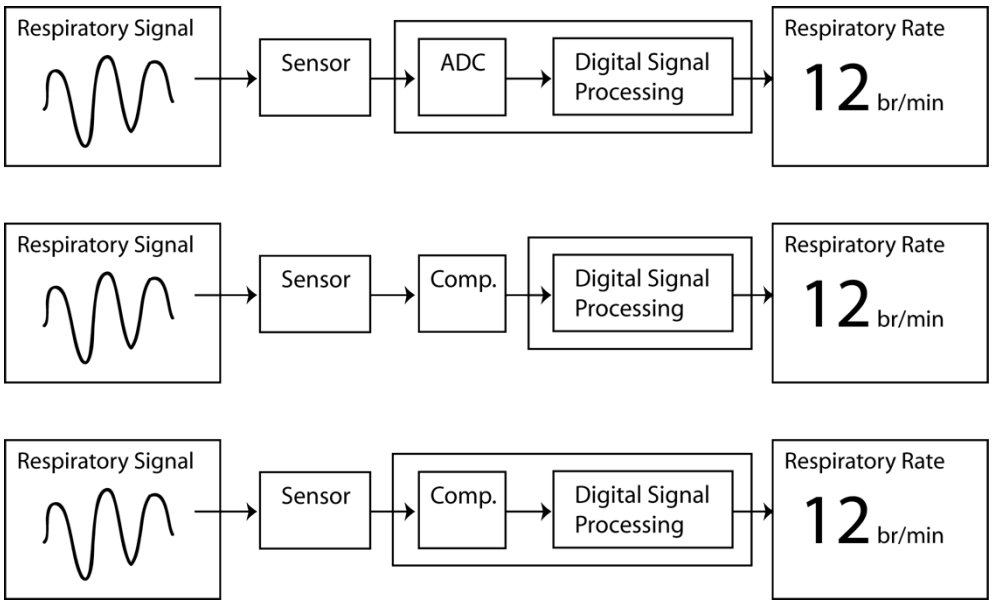



**Figure 1.4: Analog Interface Implementation Variations.**

The comparator claim of this thesis is not made to prove that the particular SoC comparator used in this study is the most power efficient analog-to-digital conversion method for all biomedical applications. The main focus of this thesis is to illustrate that, for environments where it is crucial that minimal energy be wasted, there is a careful balance required between accuracy, complexity, and power consumption specific to the application environment itself.

The decision for utilizing the comparator was a decision based on this balance and sacrifices complexity for power-savings. This decision is relative to the overall system and is not based on absolute hardware abilities. For instance, there exists an 8-bit ADC circuitry separate from the CC430 that uses as little as 3.1 μW for a 100 kHz sampling frequency at 1-V supply voltage [**57**,**58**]. The power consumption for this ADC is inherently less than that of the CC430. However, there will always be smaller, faster, smarter, and less-power-consuming technology; what is more important are the relative decisions made for the given situation.

The comparator used in this study may not utilize less power than the ultra-low-power 8-bit ADC mentioned, but it is considered much more power-efficient relative to the ADC that is available to this thesis. Also, although it is possible to utilize a separate low-power ADC, there becomes an issue of how that ADC interfaces with the SoC's

microcontroller. For mixed signal processing, the balance of power-consumption must be harmonized between the analog and digital portions.

Although an off-chip ADC may require lower power, how does it affect the logic of the microcontroller that must obtain the value and calculate a respiratory rate from it [**59**]? Instead of utilizing only one analog pin of the SoC, more digital pins must be utilized, and the SoC and the ADC must be synchronized for proper sampling. These are only two contingencies out of many that may arise and cause unexpected power results. It is for this reason that this thesis proposes the use of the SoC's own comparator over its ADC; integration between the analog and digital realms will be both seamless and less complex, resulting in less wasted energy for the algorithms implemented.

In summary, this thesis proposes an untraditional SoC on-chip comparator approach to achieve the most energy efficient implementation of qualified respiratory rate extraction and transmission algorithms. The criterion of success with respect to the overall project is that the algorithms implemented utilize less power than is available from the respiratory effort.

# 2. Data Extraction Algorithms

Given an ideal sinusoidal respiratory signal, there exist multitudes of methods with which the transmitter SoC can obtain the rate of that signal; however, it is nearly impossible to produce an exhaustive list of this plethora of possibilities, and therefore, this thesis only visits a few simple ones that the author believes might offer the reader intrigue to explore the topic of algorithm power optimization. The simplicity of these algorithms also makes it possible to draw a conclusion of whether or not it is feasible to measure respiratory rate at all given the power constraints of the overall system.

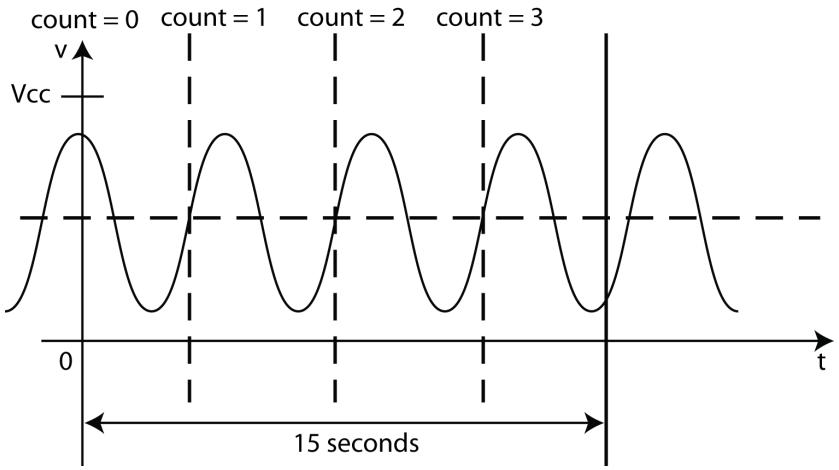
If the algorithms are too complex and the power measured is beyond what is available in the system, then no knowledge has been gained as to whether it is even possible for the overall system to function with a low-power SoC. However, if the algorithms are simple, several different conclusions can be made depending on the outcome. If the power consumption is too great, then it can be hypothesized that it is not possible to gather the respiratory rate under the system constraints with the technology proposed. On the other hand, if the power consumption is within the specifications that are desired, then the possibility of system integration has been proved and the developer can systematically increase the complexity of the algorithm where needed until the power budget is at its limit.

In general, the algorithms discussed are separated into two categories with respect to the RF transmission operation: pre-RF calculation and post-RF calculation. Pre-RF calculation algorithms gather data from the respiratory signal and process this data "on-chip" before transmitting the end respiratory rate result to the receiving module, whereas post-RF calculation algorithms gather the data and transfer it "raw" to the receiving module in expectation that the receiving module will handle the data processing of the rate. Making this RF-transmission-based distinction is actually very important with regards to analyzing the power consumption of the algorithms.

RF transmissions are known to be the energy bottleneck of most SoC systems [**60**] and keeping this trait in mind allowed argumentative filtering of which algorithms were worth implementation and further study. In total, this thesis proposes and discusses the advantages and disadvantages of two pre-RF calculation algorithms: breath counting and breath interval timing; and three post-RF calculation algorithms: real-time data transfer,

delayed data set transfer, and dummy data transfer. However, in the end, only the two pre-RF calculation algorithms were implemented based on reasons discussed here.

The first pre-RF calculation algorithm, breath counting, is a simple technique used to measure respiratory rate and is normally what is performed by the nurse or doctor during a regular physical exam. The general idea is to count the number of breaths that occur within a certain time interval; if the time interval used is one minute, then the number can be taken straight as the respiratory rate in breaths per minute (br/min). Sometimes the medical practitioner will choose a smaller time interval, like 30 seconds, and will convert it to the proper units by multiplying by some factor, which is 2 in the 30-second case. This multiplication is referred to as "rate projection" in this study and an example is shown in Figure 2.1. The basic block diagram of this algorithm is pictured in Figure 2.2.




$$= \frac{3\ br}{15s\ unit}\left(\frac{4\ (15s\ unit)}{1\ min}\right) = 12\ br/min$$

**Figure 2.1: Breath Counting Algorithm Rate Projection Example.**

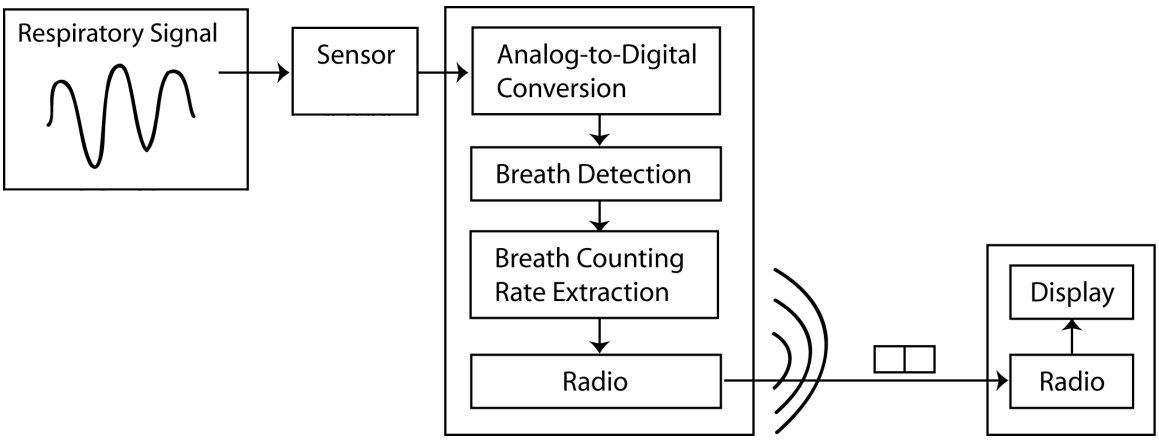



**Figure 2.2: Breath Counting Algorithm Block Diagram.**

There are three advantages to breath counting: (1) the algorithm is mathematically simple, (2) the technique has an inherent averaging effect over the time interval selected, and (3) the rate of RF transmission is generally controllable. From a code implementation perspective, the first advantage, the simplicity of the math, translates to saved CPU (central processing unit) processing time and space. The maximum-sized data required to be stored is the respiratory rate value, which, as will be explained in Chapter 3, has a range of 2 to 90 br/min. This means that the data only requires one byte for storage and also means that only one byte of data needs to be sent via costly RF transmission. The actual calculations to acquire the rate are very easy for the CPU since, at the very least, it only involves an addition of 1 every time a breath is detected. If rate projection is used, the math becomes a little more complicated; however, if the multiplication factor is a power of 2, then a the simpler operation of a left bit-shift may be used. Another advantage is that all of these calculations require only integer math.

The second advantage is the algorithm's ability for inherent averaging. Since the time from one breath to another can vary drastically depending on the breathing pattern of the individual patient, counting the number of breaths over a certain interval allows immunity from this detail. Figure 2.3 illustrates two remarkably different breathing patterns that would yield the same respiratory rate with the breath counting algorithm.

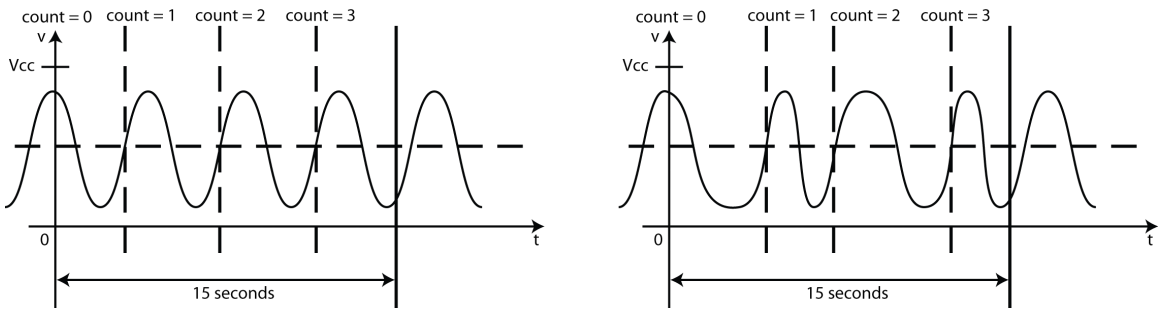



**Figure 2.3: Inherent Averaging in Breath Counting Algorithm.**

The last implementation advantage is that the RF transmission can occur at any time at or beyond the projection time interval selected. This means that a transmission occurrence is predictable and controllable, and, therefore, the expensive RF action can be postponed until transmission is absolutely required. For instance, in the case of counting for 30 seconds, RF transmission can either occur every 30 seconds or be withheld until a later time; a delay in transmission does not affect the accuracy of the calculated rate result.

On the other hand, the aforementioned advantages also have the following opposing disadvantages: (1) improved accuracy depends on the rate of respiratory signal and its phase shift with respect to CPU timing, and (2) the rate accuracy is directly proportional to the length of the time interval chosen. The first disadvantage is illustrated by Figure 2.4, which depicts what respiratory rates would be calculated for a rate of 6 br/min using a few different rate projection schemes and assuming that each peak denotes when a breath should be counted.

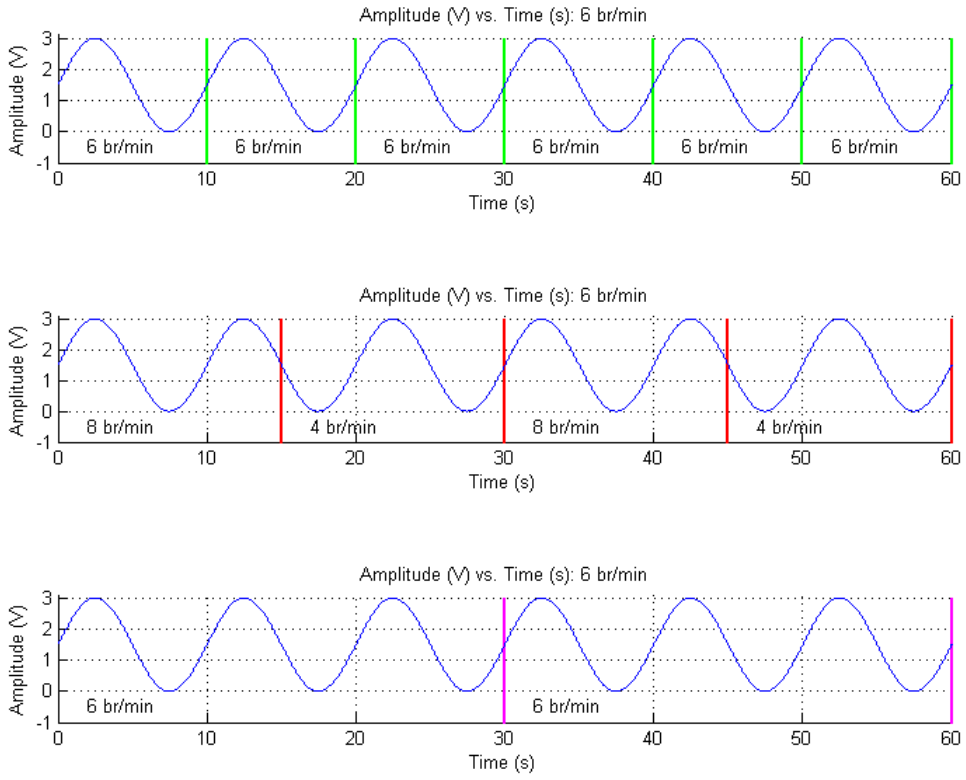



**Figure 2.4: Breath Counting Inaccuracy with Respect To Respiratory Input.**

If the respiratory rate is a multiple of the multiplication factor used for rate projection, or, in other words, the respiratory signal and the time interval frequency are relatively in-phase, then the signal is perfectly accurate with respect to one minute. This can be seen in the first and third plots in Figure 2.4, which illustrate a 10-second interval and a 30-second interval, respectively. Since 6 br/min is both a multiple of 6 and a multiple of 2, the rate calculations for each interval are the equal to each other and correct. In contrast, 6 br/min is not a multiple of 4, and therefore, there is a discrepancy in calculation in the second plot illustrating a 15-second interval. Where the discrepancy lies with respect to the one-minute window depends on its phase shift; for instance, if the wave were "advanced" by a few seconds, the first calculation would be 4 br/min and the second would be 8 br/min. This phase shift is inconsequential in relation to a long expanse of time, but the inaccuracy based on the "in-phase"-ness of respiratory signal and

the time interval frequency is not, since the input to the system is uncontrollable and unpredictable.

This “in-phase” inaccuracy is also the root of the second breath counting disadvantage, where the rate accuracy is directly proportional to the time interval chosen. This behaviour can also be stated in the following fashion: The advantage of the averaging effect lessens as the time to average over decreases. This means that the greater the multiplication we do on an already inaccurate measurement, the greater the magnification of error. Figures 2.5 and 2.6 illustrate this concept and depict rates of 7 br/min and 13 br/min, respectively. Neither of these rates are a multiple of the multiplication factor of any of the three projection schemes utilized.

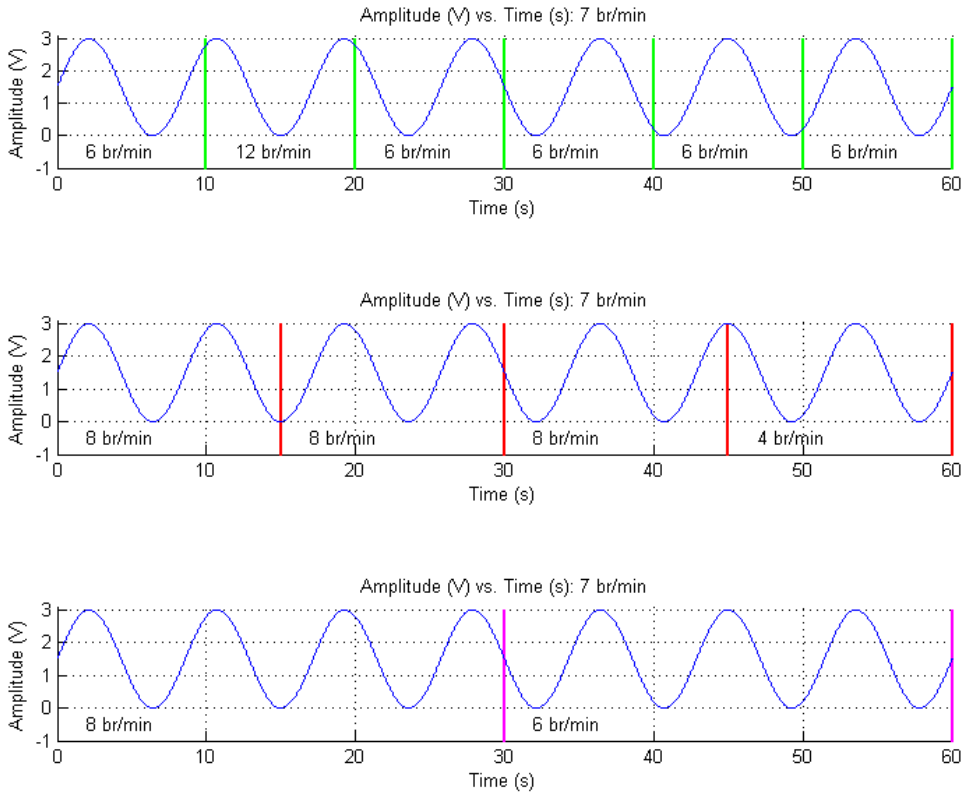



**Figure 2.5: Breath Counting Consecutive Calculation Relative Inaccuracy with Respect To Time Interval Chosen.**

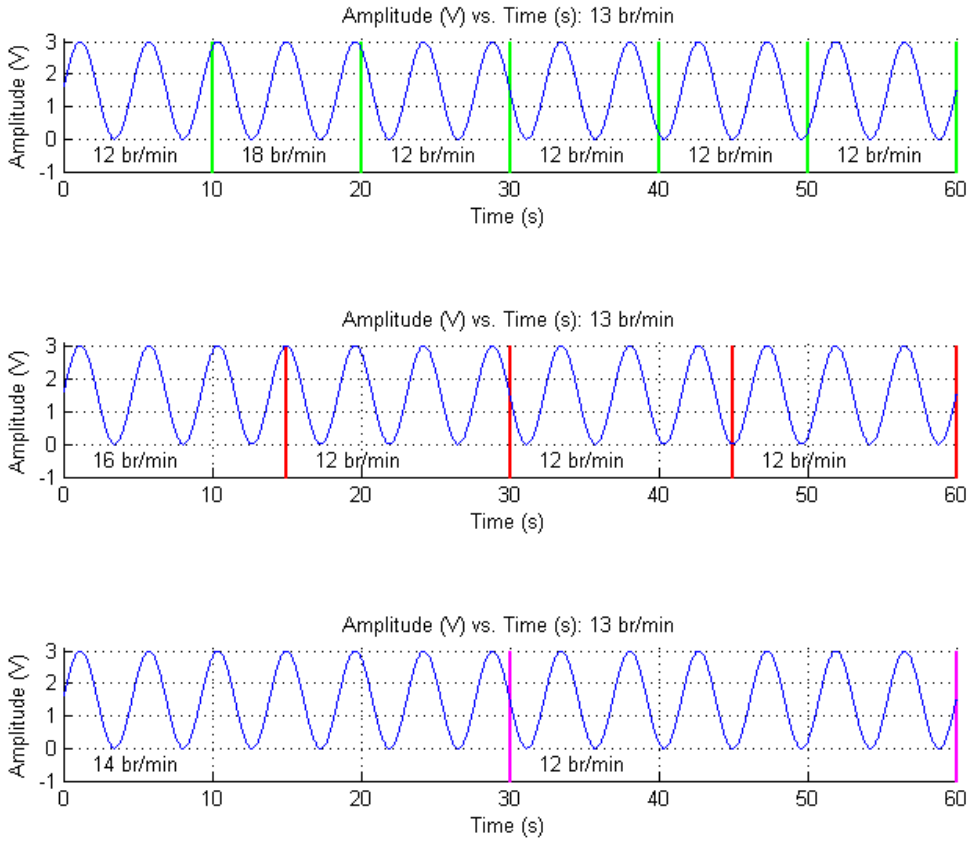



**Figure 2.6: Breath Counting Maximum Inaccuracy with Respect To Time Interval Chosen.**

Notice that both figures illustrate that the *difference* between successive calculations can be as large as the multiplication factor of the projection scheme chosen. For each of them, the estimates for the 10-s (i.e., first) and 30-s (i.e., third) projection interval plots are 6 br/min and 2 br/min, respectively. Figure 2.7, below, also illustrates that input signals with respiratory rates lower than the multiplication factor of the projection cause calculation results to fluctuate between a rate of zero and a rate of the multiplication factor.

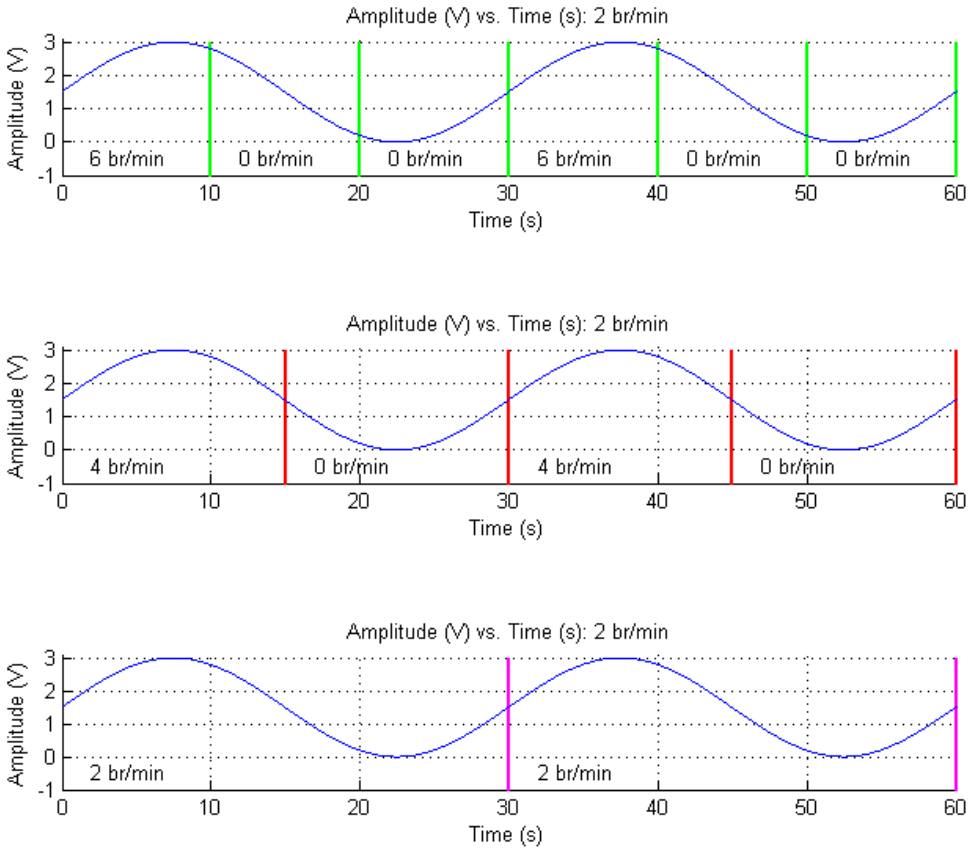
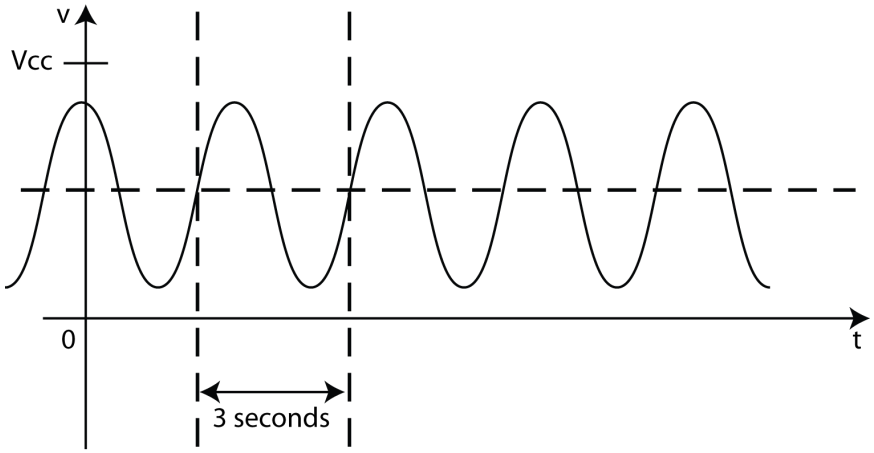



**Figure 2.7: Breath Counting Fluctuations To 0 br/min.**

Although it is possible to average out these errors over a full minute, such averaging mitigates the ability to transmit respiratory data at intervals less than one minute. This means that amount of tolerable error of the system determines the shortest acceptable interval. Also, taking into account the randomness of nature, a patient is highly unlikely to breath in an exact sinusoidal fashion to yield a breathing rate that is an integer value; they most likely will have some averaged out rate that has a fractional part, like 15.5 br/min. With respect to the SoC code, averaging creates more complex math and introduces the possibility of truncation and rounding errors. However, complex math may not necessarily be too much of a disadvantage, as is the case with the second pre-RF calculation algorithm of this thesis, breath interval timing.

Utilizing the inverse operation of breath counting, breath interval timing uses the breath itself as the basis versus a set interval of time. In this approach, the interval between two consecutive breaths is timed and then inverted by dividing the single breath by the interval measured. Figure 2.8 provides an example of this process, and Figure 2.9 presents a block diagram of the breath interval timing algorithm.




$$= \frac{1\ br}{3\ s}\left(\frac{60\ s}{1\ min}\right) = 20\ br/min$$

**Figure 2.8: Breath Interval Timing Algorithm Example.**

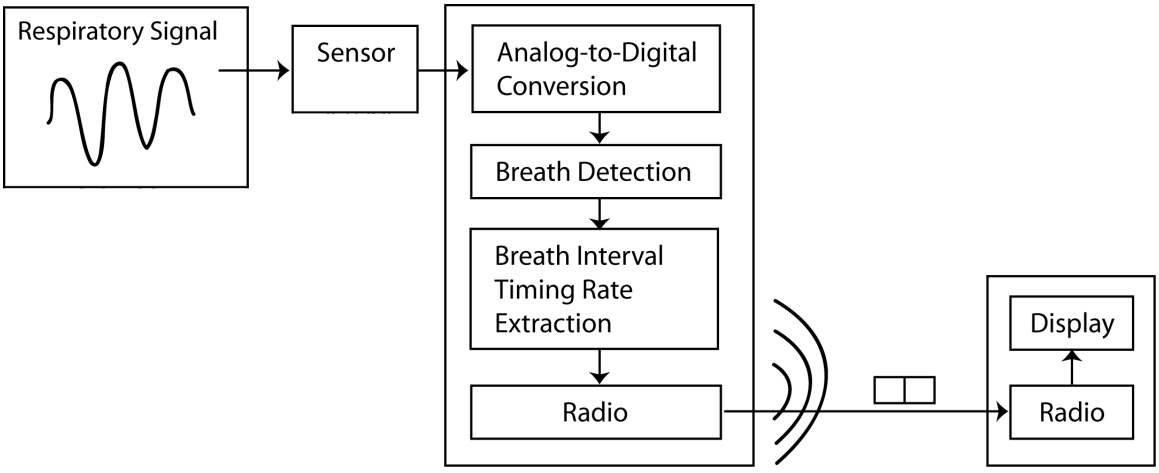



**Figure 2.9: Breath Interval Timing Algorithm Block Diagram.**

The advantages of breath interval timing are: (1) better accuracy in rate projection as compared to breath counting for normal breathing, and (2) the potential for faster

results than in breath counting. For the first advantage, “rate projection”, in this case, does not imply a multiplication towards the estimated respiratory rate, as was the case with breath counting. Instead, rate projection is more of a rate assumption where it is assumed that every breath occurs in a generally uniform periodic pattern and the time variations between each breath are miniscule and, thus, negligible in the overall picture. This assumption means that the rate between any two consecutive breaths is fairly close to the overall rate; in effect, the technique inherently projects the breath-to-breath rate to become the overall rate. Given normal respiratory function and no multiplication of inaccuracy, breath interval timing has a fairly good chance for accuracy. The actual rate of the respiratory signal does not directly affect the accuracy of the rate calculation as with breath counting.

The second advantage of breath interval timing is that it does not need to wait a set amount of time before its result is transmitted; data availability occurs after a single breath. Thus, transmission can occur after any breath or it can be set to occur at a specific frequency, resulting in a wider range of control over when the data is transferred via RF. For example, if the patient breathes at 40 br/min, which equates to 1.5 seconds between each breath, RF transmission can occur every 1.5 seconds or be withheld until transmission is required. As opposed to breath counting, the accuracy of the rate calculation is not degraded as the rate of RF transmission increases.

Unfortunately, the advantages of breath interval timing are also its disadvantages; (1) the inherent rate prediction makes the algorithm susceptible to the larger breath-to-breath variations that breath counting was immune to, and (2) the math that increases the accuracy of the result is more complex than a simple addition. The first drawback is illustrated in Figure 2.10, in which the interval between breaths 1 and 2 is quite different than the interval between breaths 2 and 3. Therefore, two very different results can occur depending on which breaths are chosen for the calculation. However, in the overall picture, this may still be an advantage over breath counting, because there is a higher likelihood of informing the medical practitioner of abnormal breathing patterns.

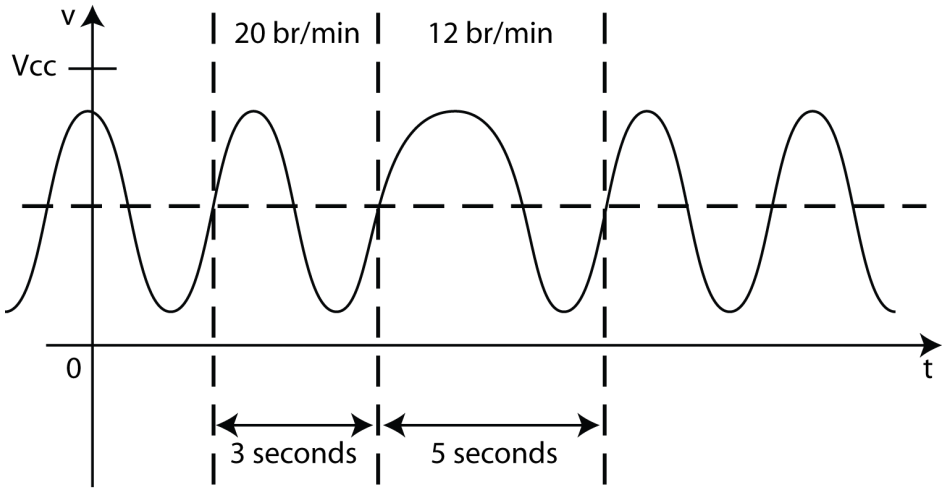



**Figure 2.10: Variance in Interval Timing Algorithm Results Caused by Breath-to-Breath Variance.**

As for the second disadvantage, the implementation of the math required is a bit more complex, leading to slightly higher CPU time and space requirements. Depending on the granularity of units for timing the breath interval, the largest value that needs to be stored is not the respiratory rate, but is the time between two breaths. This means that, instead of a simple byte, possibly a 16-bit or higher variable is required. Luckily, the data actually transmitted via RF is still only a byte as with the breath counting algorithm.

Unfortunately, unlike the breath counting algorithm, unless there is a reset after a certain time limit, a long case of apnea can cause the timing module of the SoC to infinitely run and constantly overflow. If, or when, apnea ceases, the result will be corrupted and useless. In addition to changes in the space and run-time requirements, the division required is inherently a float operation that cannot be simplified to a right bit-shift. In this light, if integer math is used in an attempt to lessen mathematical complexity, not only is there a truncation error, but there is also an error based on the granularity of the timing scale chosen. Figure 2.11 depicts a respiratory signal with a rate of 31 br/min; the first plot illustrates when the time scale is too large and two breaths occur before the "time" is updated; the second and third plots show that as the granularity of the time scale improves, so does the RR estimation.

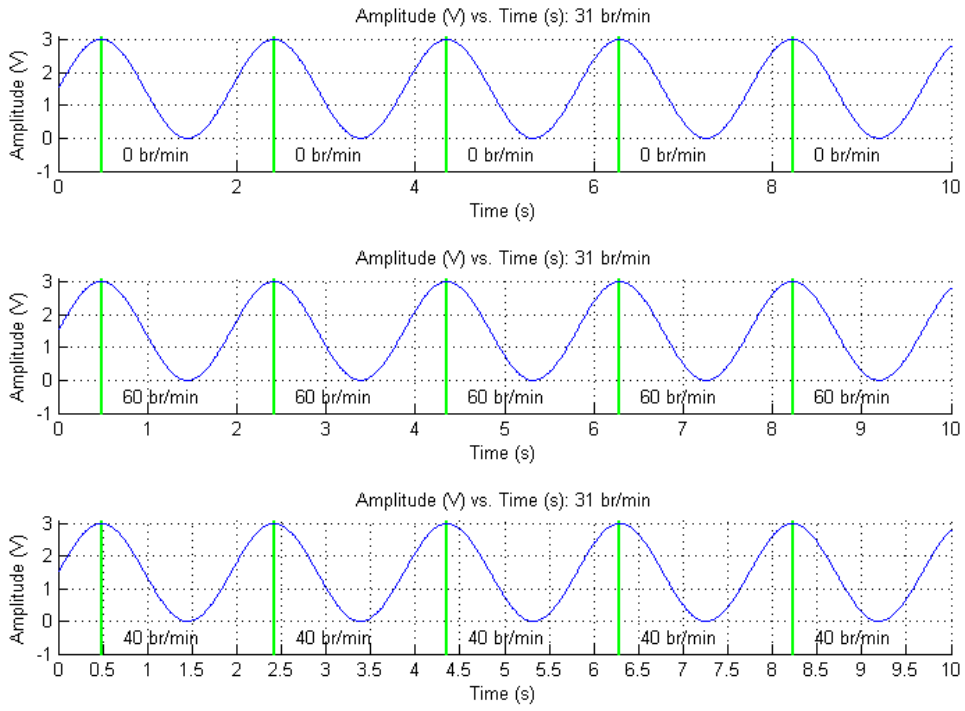



**Figure 2.11: Timing Scale Error.**

Explicit averaging may help with breath-to-breath variations; however there remains the question of how many samples are considered "enough". Also, the time it takes to acquire the samples decreases the speed at which data can be transmitted. If breathing is slow, the time between breaths can become very long, and transmission time will be hindered. To rid the transmitter of complex math, post-RF calculation algorithms, as opposed to pre-RF calculation algorithms, leave the heavy calculations as a task for the receiver. The first post RF-algorithm under consideration is the real-time data transfer algorithm.

Real-time data transfer is simply a "pass-through" algorithm; the voltage of the respiratory signal is read at an acceptable sampling rate and then transmitted via RF directly after this reading is finished. However, it is important to note that this algorithm may have some hidden pre-processing in order to prepare the voltage for transmission depending on how the algorithm is implemented. The transmitted voltage is then processed by the receiving module, which can check for "zero-crossings" or another

periodic feature indicating that a full or half of a breath has elapsed. Figure 2.12 provides an example of how this algorithm works, and a block diagram of it is available in Figure 2.13.

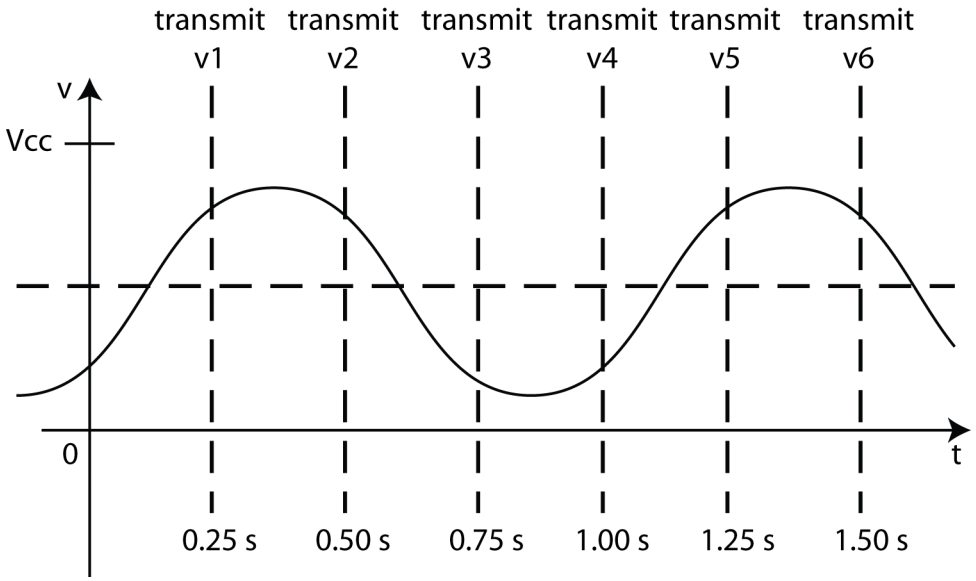



**Figure 2.12: Real-Time Data Transfer Example with 4-Hz Sampling Frequency.**

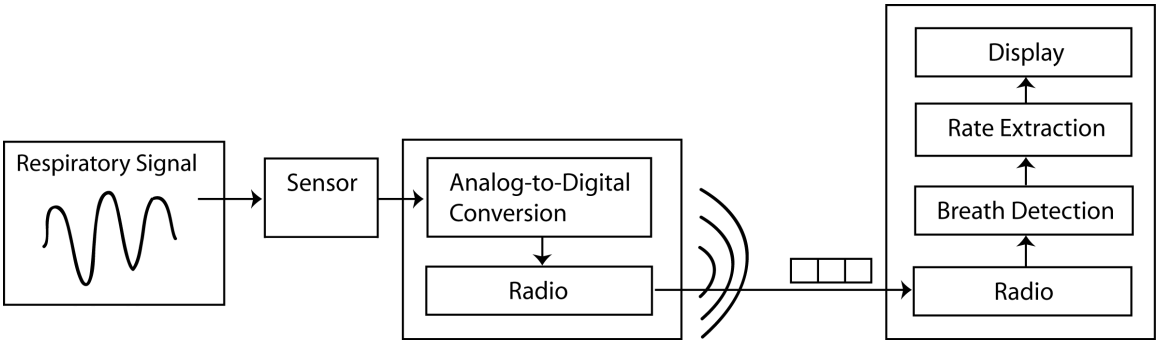



**Figure 2.13: Real-Time Data Transfer Algorithm Block Diagram.**

The advantage of this algorithm is that accuracy of sampled data is guaranteed since there is nothing done to it other than preparation for RF transmission. However, the disadvantage of this algorithm is that constant sampling and transmission of a possibly larger than 8-bit value must occur at a relatively rapid pace. The sampling rate must be set fairly high in order to cover the whole RR spectrum and capture the waveform accurately under the Nyquist Theorem. Given that the highest respiratory rate is 90

br/min, the sample rate must be at least at 2 * (90 cycles/60 s) = 3 Hz, which equates to about 0.33 seconds between each RF transmission. Since transmission must occur at a rate mandated by the nature of the algorithm, it is not controllable like in the aforementioned pre-RF calculation algorithms. Also, unless the "current-time" is sent with the data (acquired either through a timer or the real-time clock (RTC)), there will be a slight delay from true "real-time" since the message must pass through space.

A slight variant of transferring data in "real-time" is transferring data in "near-real-time" in order to cut down on the number of transmissions; the delayed data set transfer algorithm reads the data, stores it, and then transmits it in small bursts as indicated in Figure 2.14 and pictured in Figure 2.15. The advantages of this algorithm are that there are slightly less transmissions than real-time transfer and that there is a possibility for application of some small calculations to compact the overall data prior to transmission. However, the delay in data must be balanced with this transfer advantage or else the transferred data might be too stale when the receiver procures it. Although, it is unknown whether a burst will take less power than a sequence of smaller transmissions, power savings is expected based on the assumption that less packet wrapping information is used per payload byte. In comparison to real-time data transfer, this algorithm also has the same issue with the delay through space and, unfortunately, requires storage space.

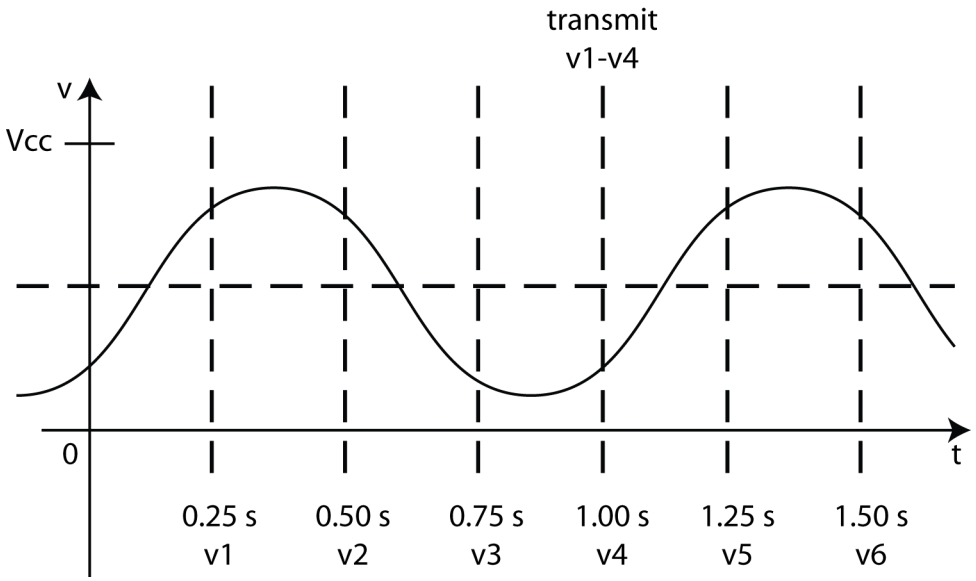



**Figure 2.14: Delayed Data Set Transfer Example with 4-Hz Sampling Frequency and 1-Hz Transmission Frequency.**

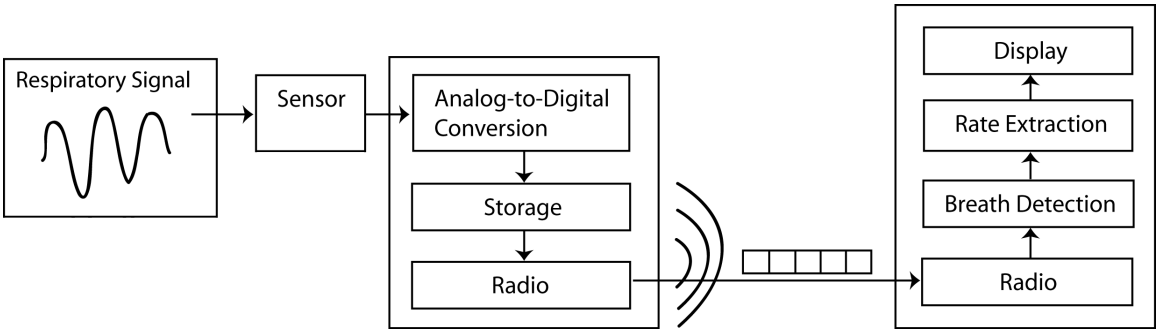



**Figure 2.15: Delayed Data Set Transfer Algorithm Block Diagram.**

The last post-RF algorithm considered completely removes all calculations on the transmitter's part; instead of transmitting an actual number with pertinent information, dummy data transfer sends a dummy packet each time a new breath is detected as exemplified in Figure 2.16 and illustrated in Figure 2.17. The receiving module is then expected to process the data in either a manner similar to the pre-RF calculation algorithms or utilize more advanced algorithms that might account for artifact movements and such. Essentially, the SoC acts as an analog-to-digital-RF converter.

The advantage of this algorithm is that there are no requirements of storage or mathematical computations for the transmitter; not even any hidden preparations as in

real-time data transfer. However, this means that more power is required when the respiratory rate is higher, since faster breathing would initiate more frequent RF transmissions. This may be taken as an advantage or disadvantage; a faster respiratory rate may have a shallower breath, which might, in turn, decrease the amount of power available to run the SoC.

Another disadvantage that is important to note is the delay caused by transmitting through an unknown space. The general idea of the algorithm is to create a "digital-like" RF signal from which the receiver calculates the rate; however, since there are many variables in the environment that can interfere with the reception of the data, such as the distance between the transmitter and receiver, the amount obstacles in the room, and RF interference from other wireless devices, integrity of timing is questionable. This problem might be again solved by sending the "current-time" with the dummy packet; however, utilizing a timer or the RTC slightly complicates the algorithm and counteracts the general advantage of "no computations".

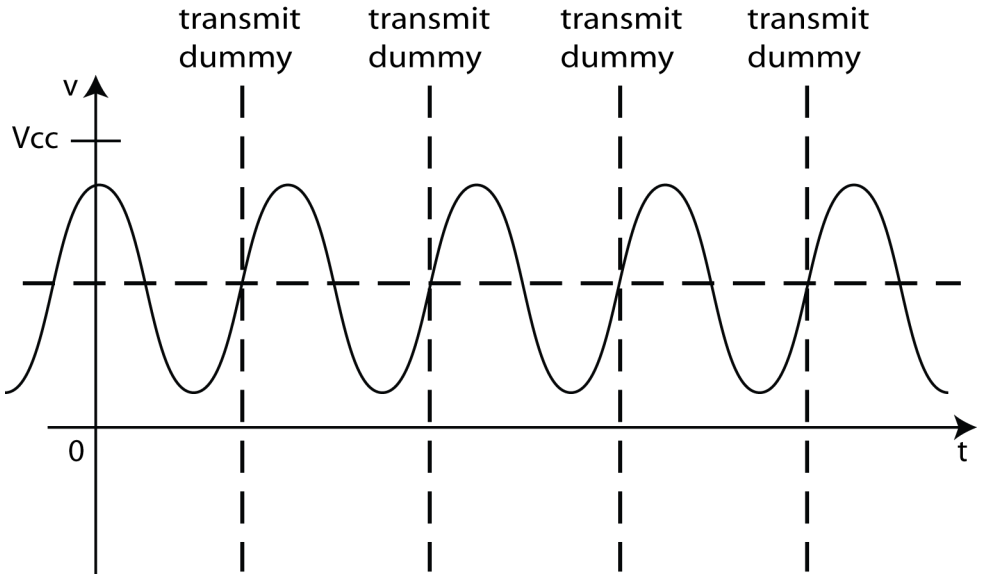



**Figure 2.16: Dummy Data Transfer Example.**

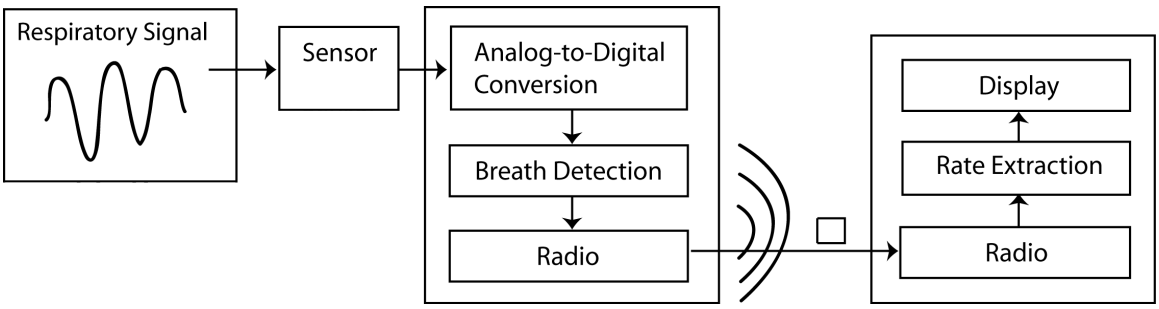



**Figure 2.17: Dummy Data Transfer Algorithm Block Diagram.**

Table 2.1 provides a summary of the pros and cons of each of the five algorithms discussed. As mentioned in Chapter 1, there are two parts to each algorithm: the digital extraction portion and the RF portion. The table allows a comparison of the fives algorithms with respect to these parts. The strength of the two pre-RF calculation algorithms lies in optimization of the RF portion while sacrificing simplicity of the digital extraction portion. In contrast, the three post-RF calculations algorithms concentrate on refraining from complex computations while sacrificing reserved use of RF components. Conceptually, pre-RF and post-RF calculation algorithms are equally matched; however, empirically, there is a generally accepted predilection towards pre-RF calculation algorithms.

**Table 2.1: Algorithm Pros and Cons (* indicates algorithms implemented)**

| Algorithm | Pros | Cons |
|---|---|---|
| Breath Counting* | • Simple math<br>  o Integer addition<br>  o Faster computations<br>  o Less CPU time<br>• Data guaranteed to fit in a one byte variable<br>• Rare use of wireless transmission module with small packet | • Long and fixed sample duration<br>• Freshness of data transmitted is inversely proportional to preciseness and accuracy of the data itself<br>• Slightly more complex math if averaging or projection is used |
| Breath Interval Timing* | • Potential for a variable rate of wireless transmission proportional to rate of signal to be measured<br>  o Possibly shorter sample durations<br>  o Variable use of wireless transmission module with small packet | • Complex math<br>  o Integer division or multiplication of inherently floating point values<br>  o Time capture scaling math<br>  o Slower computations<br>  o More CPU time |
| Real-Time Data Transfer | • More precise and accurate data<br>• Straightforward, pass-through algorithm<br>• No data storage | • Heavy use of wireless transmission module with medium packets<br>• Possible distortion of breathing rate due to physical transmission delay from the local atmosphere and spatial obstacles, depending on implementation |
| Delayed Data Set Transfer | • More precise and accurate data<br>• Straightforward, pass-through algorithm<br>• Potential for some algorithmic summarization of data | • Mild use of wireless transmission module with large packets<br>• Delayed data<br>• Data storage required<br>• Possible distortion of breathing rate due to physical transmission delay from the local atmosphere and spatial obstacles, depending on implementation |
| Dummy Data Transfer | • Extremely simple, zero-computation, pass-through algorithm<br>• No data storage<br>• Rate of wireless transmission is directly proportional to the rate of breathing; this may be considered a con depending on future studies | • Distortion of breathing rate due to physical transmission delay from the local atmosphere and spatial obstacles, unless a certain implementation is applied |

In Chapter 1, it was stated that there are currently two ways in which software is optimized specifically for low-power wireless medical applications: (1) the avoidance of transmissions, and (2) adaptive sampling and power allocation. Of the five algorithms presented, only the two pre-RF calculation algorithms follow these principles. More specifically, both breath counting and breath interval timing practice data compression and infrequent transmissions to save energy.

Although it would be interesting to evaluate the power consumption of all five algorithms, current low-power wireless technology already flags the RF transmission as the overwhelming power bottleneck of wireless systems [**60**]. Thus, it was decided that all post-RF calculation algorithms be left unimplemented in the interest of time. If RF technology improves in the future, the feasibility of post-RF calculation algorithms needs to be revisited; however, the sole implementation of the pre-RF calculation algorithms was considered sufficient to accomplish the aim of wirelessly acquiring the respiratory rate with an SoC while utilizing the least amount of power.

# 3. Implementation

There are two facets to implementation; the first is the context of the implementation, and the second is the implementation itself. The first subchapter below explains the context of this thesis's implementations and its requirements, constraints, and assumptions, while the second subchapter elucidates the actual code implementation for the CC430 in detail.

## 3.1. Requirements, Constraints, and Assumptions

There are several limiting factors to be considered when designing any system. For this thesis, these are the functional requirements of the system, the constraints introduced by available hardware, and the assumptions made. Some of these limiting factors are cyclic in nature; for instance, an assumption might be made to create further constraints on the system, or certain requirements may be reduced based on a presented assumption. This subchapter will first explain the functional requirements and hardware constraints that must be accounted for, and then it will describe the assumptions made; however, because of their intermingled dependencies, this general order might be broken in certain instances to provide better clarity and coherency.

In order to determine the functional requirements of this thesis's subsystem, it is important to understand the overall project's target environment. In general, when designing a device to be used in a medical setting, there are certain unique challenges and parameters that account for the safety, security, and privacy of patients, medical personnel, and all others involved in the medical scene.

For instance, the speed and accuracy of data is very important in order to determine the medical condition of the patient; if data is unacceptably inaccurate or too slow, it could compromise the patient's safety. Also, considering the wireless attribute of this thesis's particular situation, proper RF emission standards and guidelines must be followed to avoid causing interference with other wireless devices, especially other wireless medical devices. RF emissions must also be regulated with regards to how much exposure is allowed within the target environment and around the individual patient. As to security and privacy, encryption of wireless data might need to be considered depending on the application and range of the signal. Such precautions are very important

when the data transferred can personally identify the patient or other involved parties; however, it may not be too critical in certain situations given that, with current hospital proceedings and bed-side monitors, a patient's current health status is visible to almost anyone in the room. This visibility to immediate parties is most likely allowed to promote swiftness during emergencies; however, if the patient is at home and is monitored via their home wireless network, unscrupulous parties outside and near the home could tamper with or tap into private information and medical data [**61**,**62**,**63**,**64**].

One of the main tasks of this study is to measure respiratory rate; therefore it is a requirement that the system is functional for most of, if not the whole, respiratory spectrum. Normal ranges of RR were mentioned in Table 1.1; however, these rates do not take into account the infrequent, but still possible, extremes. After perusing several sources, it was determined that the slowest rate expected is 2 br/min [**65**], while the fastest rate expected is 90 br/min [**5**]. Converting this range into hertz (i.e., one cycle = one breath), the range requirement becomes about 0.033 – 1.5 Hz. As for the speed at which the rate displayed must be updated, there does not seem to be a standard; therefore, this thesis assumes that, at best, data should be updated every 10 seconds based on the definition of apnea.

Although very pertinent to the target environment, the medical RF wireless requirements of the system cannot be implemented to the fullest. This is because of the constraints imposed by the EM430F6137RF900 boards and the 868/915 MHz antennas provided with them. These components limit this thesis's RF communication to the 868/915 MHz range, and TI provides a disclaimer that the boards have not undergone emissions testing. Although this thesis carried out its testing under these constraints; the end product *must* be sure to follow all regulatory standards and procedures. Two organizations that should be referenced when considering wireless medical devices are the United States Food and Drug Administration (FDA) and the Federal Communications Committee (FCC). The first organization provides guidance for the medical aspects [**66**] of wireless devices, and the second provides safe RF emission policies [**67**] and explains which frequency bands unlicensed devices are allowed to use [**68**]. More specifically, the FCC defines the industrial, scientific, and medical (ISM) RF band.

Privacy and security are also important to the end device; however, as mentioned before, the extent to which these traits require implementation depends on the specific environment of the system. For simplicity, this study assumes that the patient is relatively near the base station and that other people are unable to be in the vicinity of the wireless signal without being detected. Therefore, privacy and security were not addressed in this study. Moreover, implementing encryption to add privacy and security produces added complexity to the algorithm that might become an undue power burden if it is decided that it is not needed in the end product.

In addition to requirements from the medical environment, there is the functional requirement that is the focus of the overall project: the power consumption must be less than what is produced by respiratory effort. After referencing sources [**22**,**23**], which describe the energy harvesting aspect of the overall project, and discussing constraints with the author of [**23**], it was estimated that the power available to the system is roughly 100 μW – 3.3 mW. Therefore, using the most stringent constraint, this study's SoC must not exceed an average power consumption of 100 μW to be considered successful enough for implementation. This criterion is heavily based on the crude assumption that the other components of the system take little to no power for proper operation. If it is discovered that this is not the case and that the power consumption measured at the end of this experiment is not acceptable, this study must be revisited.

The second limiting factor to consider is the constraint from available hardware and hardware features. The general implementation of the digital processing unit is an open-ended question; however, this study has chosen to investigate the CC430 as a viable candidate for this role. Choosing a specific SoC not only limits what modules are offered, but also creates power constraints based on what modules are actually used. The CC430 has several different components that must all be evaluated for necessity first before consideration as a limiting component of this thesis.

In addition to volatile (random-access memory, RAM) and non-volatile (flash) memory, the CPU, and several digital and analog I/O pins, the CC430F1637 on the evaluation board contains a unified clock system (UCS), a power-management module (PMM), a direct memory access (DMA) controller, a port-mapping controller, a cyclic-redundancy check (CRC) module, an AES 128-bit encryption accelerator, a watchdog

timer (WDT), two general 16-bit timer modules, a real-time clock (RTC), a 32-bit hardware multiplier, a reference voltage generator, a 12-bit analog-to-digital converter (ADC), a comparator, two universal serial communication interfaces (USCI) that, combined, support UART/IrDA/SPI/$I^2C$ protocols, a liquid crystal display (LCD) controller, a sub-1-GHz radio, and an embedded emulation module. Not all of these modules are pertinent to this thesis, because of the assumptions named earlier or the fact that their functionality is not required; the modules of particular interest are the RAM, flash memory, CPU, analog I/O pins, UCS, 16-bit timers, 12-bit ADC, comparator, and sub-1-GHz radio. Each of these components has specific power requirements, which impose particular constraints on this thesis's subsystem. After thoroughly examining the datasheet for recommended operating conditions and considering what parts came with the evaluation kit, Table 3.1 was generated as a summary of important points. Also incorporated into the table are special features of the CC430 that cater to low-power environments; these include four low-power modes (LPM) and the ability for interrupts.

**Table 3.1: CC430 Features and Constraints.**

<table>
<tr><th>Module/Feature</th><th>Constraints</th></tr>
<tr><td>System</td><td>
<table>
<tr><td>PMMCOREVx</td><td>Vcc</td><td>$f_{SYSTEM}$</td></tr>
<tr><td>0 (no radio)</td><td>1.8 to 3.6 V</td><td>0 to 8 MHz</td></tr>
<tr><td>1 (no radio)</td><td>2.0 to 3.6 V</td><td>0 to 12 MHz</td></tr>
<tr><td>2 (radio, default PMM)</td><td>2.2 to 3.6 V</td><td>0 to 16 MHz</td></tr>
<tr><td>3 (radio)</td><td>2.4 to 3.6 V</td><td>0 to 20 MHz</td></tr>
</table>
</td></tr>
<tr><td>Temperature</td><td>- 40 to 85 °C</td></tr>
<tr><td>LPM</td><td>
<table>
<tr><td>LPM</td><td>Vcc</td></tr>
<tr><td>0</td><td>2.2 V (PMMCOREV0)<br>3.0 V (PMMCOREV3)</td></tr>
<tr><td>1</td><td>unknown</td></tr>
<tr><td>2</td><td>2.2 V (PMMCOREV0)<br>3.0 V (PMMCOREV3)</td></tr>
<tr><td>3</td><td>3.0 V</td></tr>
<tr><td>4</td><td>3.0 V</td></tr>
</table>
</td></tr>
<tr><td>16-bit Timer</td><td>$f_{TIMER}$ = 0 – 25 MHz (capture pulse 20 ns min)</td></tr>
<tr><td>Interval Reference Voltage Generator</td><td>Vcc = 1.8 to 3.0 V</td></tr>
<tr><td>ADC</td><td>Vcc = 2.2 to 3.6 V<br>Analog Input = 0 to Vcc (also capped by reference voltage generator)<br>$f_{ADC12CLK}$ = 0.45 to 5.4 MHz<br>$f_{ADC12OSC}$ = 4.2 to 5.4 MHz</td></tr>
<tr><td>Comparator</td><td>Vcc = 1.8 to 3.6 V</td></tr>
<tr><td>Sub-1-GHz Radio</td><td>Vcc = 2.0 to 3.6 V<br>PMMCOREVx = 2,3<br>Frequency Bands: 300 – 348 MHz, 392 – 464 MHz, 779 – 982 MHz<br>Data Rates: 0.6 – 500 kBaud (2FSK), 0.6 – 250 MHz (2GFSK, OOK, ASK), 26 – 500 MHz (MSK)<br>$f_{RFCRYSTAL}$ = 26 to 27 MHz</td></tr>
</table>

The general supply voltage, Vcc, range is about 1.8 to 3.6 V; however, in order to utilize the radio, only the PMMCOREV2 and PMMCOREV3 settings are allowed. These settings require a voltage of 2.2 to 3.6 V. Again referencing the datasheet, the most power efficient LPM setting is LPM4, which means that a voltage of 3.0 V is recommended for proper operation instead of 2.2 V. Therefore, the Vcc that this study utilized as the lowest allowable was voltage was 3.0 V. As for the radio hardware required, such as the crystal, the circuitry was already provided on the EM430F6137RF900, and, as mentioned earlier, the frequency was limited to 868/915MHz by the antennas provided.

Table 3.1 also illuminates the constraint that the conditioned respiratory signal must be in the range of 0 to Vcc volts in order to be properly connected to an analog pin. Therefore, the test environment simulated a patient's breathing by utilizing a function generator to produce a sinusoidal waveform between 0 and 3.0 V. A sinusoidal shape was chosen based on the periodicity of the recorded respiratory waveforms in [**23**,**22**]. Although the piezoelectric and servomotor outputs are not ideally sinusoidal, they still retain certain important characteristics, such as maxima, minima, rising edges, and falling edges, that can be used by the algorithms mentioned Chapter 2 to track the signal's rate. Another proponent for the 0 – 3.0 V sinusoidal assumption was the circumstance that the ASP portion of the overall system was not fully developed at the time of this thesis; having a known respiratory input allows for better checking of development correctness since results are predictable, are reproducible, and allow for systematic debugging.

Other assumptions were based on the indefinite nature of the overall project and end environment. For instance, the noise tolerance from extraneous movement, the distance between transmitter and receiver, and amount of data assurance required are unknown; therefore, these quantities were not considered during the implementation of this thesis. Also, since the general goal is only to "monitor" the patient, communication between the transmitter and the receiver was assumed unidirectional, as stated before. This assumption also cuts down on the energy required for acknowledging packets and such. If wireless integrity is an issue in the future, then the receiving device is expected to be able to handle most of the processing by ensuring it receives the expected information in a timely manner, otherwise sounding an alarm. This is most likely the better option versus relying on the transmitter's functionality, since the receiving device should be plugged into a more stable power source. Table 3.2 summarizes these assumptions and the other limiting factors deduced from the discussion in this subchapter.

**Table 3.2: Deduced Study Limiting Factors.**

| Limiting Factor | Summarized Points |
|---|---|
| Requirements | • Respiratory rate: 2 – 90 br/min (0.033 – 1.5 Hz)<br>• Transmitter power consumption: ≤ 100 μW |
| Constraints | • CC430 Vcc: 3.0 – 3.6 V<br>• CPU clock frequency: 0 – 20 MHz<br>• Respiratory signal: 0 – Vcc V<br>• RF communication frequency: 868/915 MHz<br>• RF crystal frequency: 26 MHz |
| Assumptions | • Apnea alarm: none<br>• Communication: "on-patient" → "off-patient"<br>• Data assurance (CRC): none<br>• Encryption (AES-128): none<br>• Noise tolerance: none specified<br>• Receiver power consumption: none specified<br>• RF signal strength/sensitivity: none specified<br>• Shortest time b/w rate display updates: 10 s |

## 3.2. CC430 Code Details

As with any SoC, the CC430 combines many digital functions into a single package and is designed to be versatile enough to fit whatever suits the project's need; implementation of anything is not a trivial task, especially when power is of major concern. Each line of code must be scrutinized, and each choice must be made without arbitrariness. Implementation of the two data acquisition algorithms selected in Chapter 2 will be presented in two parts; firstly, this thesis will explain the low-power and general elements common to both algorithms, and then, secondly, it will describe and highlight particular aspects exclusive to each algorithm's implementation. Although the following implementations focus on minimizing power consumption, there are also several places, which allow deviation to spawn further studies. Therefore, whenever possible, the author will illuminate areas for variation and possible avenues for improvement.

Before discussing the transmitter implementations, the author would like to briefly remind the reader that the receiver implementation is presumed relatively inconsequential since it is assumed that it has an "infinite" source of power. Its purpose in this study is to merely validate the functionality of the transmitter algorithm; thus, its own algorithm will not be discussed further, and, if the reader desires to view the code in detail, they may refer to Appendix A. This code utilizes an RS232 serial interface in order to display the received RR on a PC using a program called HyperTerminal. Communication between the receiver and the transmitter employs a low-power

proprietary star network RF protocol stack developed by TI called SimpliciTI™ [**69**]. Figure 3.1 illustrates an updated block diagram of the subsystem considering this feature.

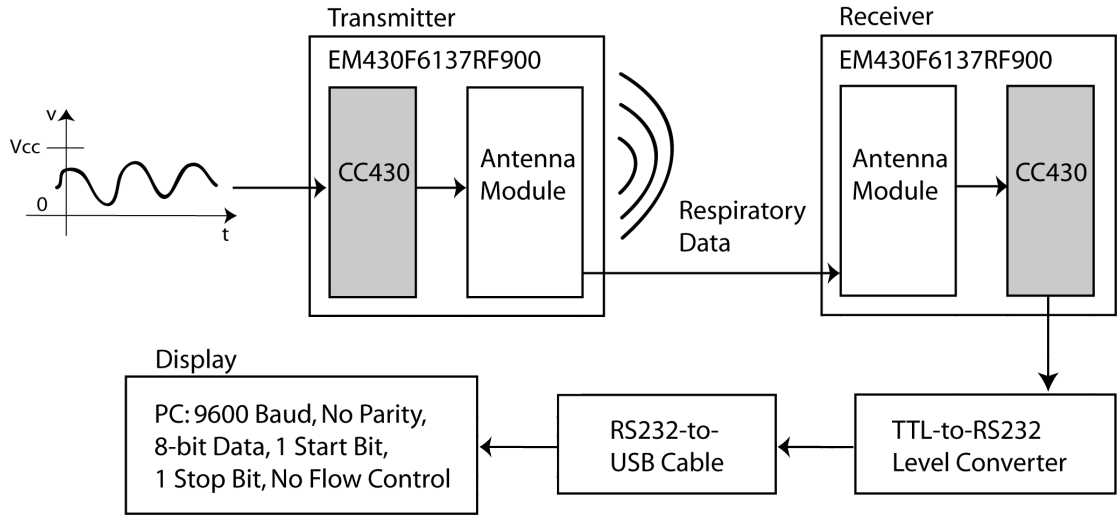



**Figure 3.1: Updated Subsystem Block Diagram.**

For both the breath counting algorithm and breath interval timing algorithm, several general low-power practices were employed. For instance, interrupts were utilized to maintain low-power mode (LPM) and avoid polling. In addition, all unneeded default functionalities were disabled; more specifically, the supply voltage supervisor (SVS) and the supply voltage monitor (SVM) were completely disabled, and all unused pins were set as digital outputs outputting a logical 0 to avoid parasitic current. Also, macros were used to allow switching between options that should ultimately be determined by the specific application environment. For instance, there are four different time intervals at which transmission can occur: every 10 seconds, 15 seconds, 30 seconds, or 1 minute. This option requires that the certain registers be set to certain values. Although it is tempting to let the microprocessor calculate the proper values, it poses unnecessary computation requirements on the processor. If possible, it is always preferred that hand-calculations and macros be used to avoid unwelcome power consumption.

As per computational accuracy, averaging was considered for both algorithms; however, as mentioned in Chapter 2, it poses extra computational requirements from the microcontroller, leading to extraneous power consumption if there exists no need for improved accuracy. It also requires a costly division and an additional variable to store

the sum of the samples. In addition, the time it takes for the data to be ready for RF transmission would be longer. In any event, if desired, averaging can be done on the receiver end, and therefore, was left unimplemented in this study.

For CPU and timer clocking sources, the CC430 has a special clock module called the Unified System Clock module, or USC; it allows for several clock sources to be linked and divided various ways to provide timing for the CPU and peripheral modules. This versatility leads to several possible choices of which it is difficult to definitively say which possesses the best balance of timing accuracy and low power consumption. Therefore, the author chose to use the internal, trimmed, low-frequency oscillator (named "REFO"), which has a typical frequency of 32.768 kHz, as the single clock source for the system. The reason for this choice was because it required the least amount of tampering with the EM430F6137RF900 and was cited as requiring only 3 μA. Future studies may want to look into the other possible clock sources for improved power efficiency.

The last commonality between the two algorithms is the analog interface. There are two interfaces that can be used to interact with the respiratory signal; they are the 12-bit ADC and the comparator. As mentioned in Chapter 1, the comparator functions like a 1-bit ADC. The fact that both the breath counting algorithm and breath interval timing algorithm only require an *indication* of when a breath occurs rather than the amplitude allows for a choice between an ADC breath detection interface and a comparator breath detection interface. This study utilizes the comparator since it has the conceptual potential to utilize less power than the ADC. Citing the datasheet, the can draw as little as 0.5 μA with certain settings as opposed to the ADC, which can draw 220 μA. Figure 3.2 illustrates the general concept of the subsystem.

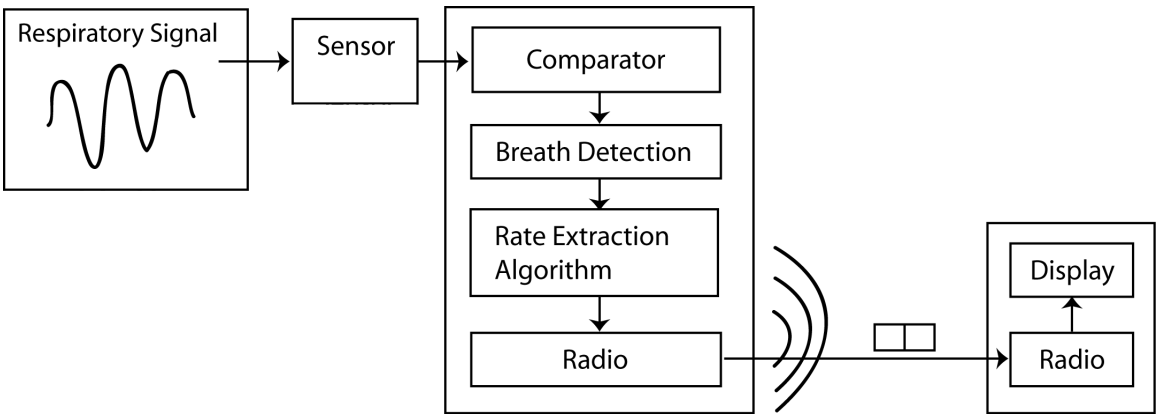



**Figure 3.2: Conceptual Subsystem Block Diagram.**

Depending on the reaction of the comparator to the respiratory signal, a new breath will either be considered as having just started or as in-progress. It is then possible to extract the RR from these occurrences using the breath counting or breath interval timing algorithms. After extraction, the rate is packaged and sent to the receiver via RF technology. The basic premise of how the comparator scheme will work with the pre-RF calculation algorithms is that, on the rising edge of the comparator output, an interrupt occurs to indicate that the breath has been detected. Figure 3.3 depicts how the CPU reacts to the comparator input. Also, the diagram indicates the usage of a single timer for both the transmission timing and the time point captures.

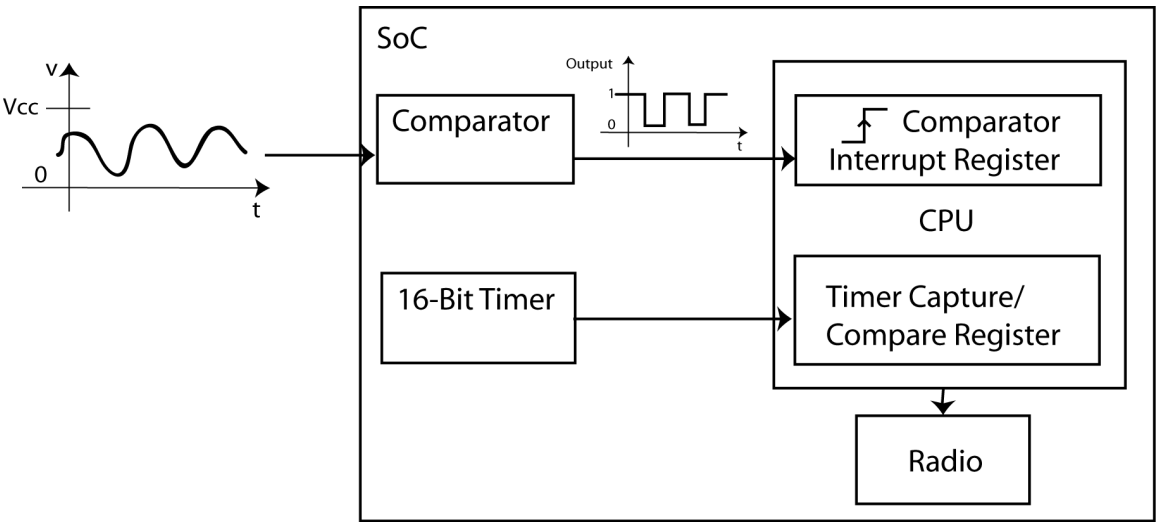



**Figure 3.3: Operational Transmitter SoC Block Diagram.**

Regrettably, there are some drawbacks to the comparator selection. For instance, if a single reference voltage is used, such as 1.5 V, a low-frequency sinusoidal input creates a ringing effect at the output. An example is shown in Figure 3.4: Since the amplitude of the input hovers near 1.5 V, uncertain digital comparisons tend to initiate unwanted interrupts despite SoC filter settings for maximum delay.

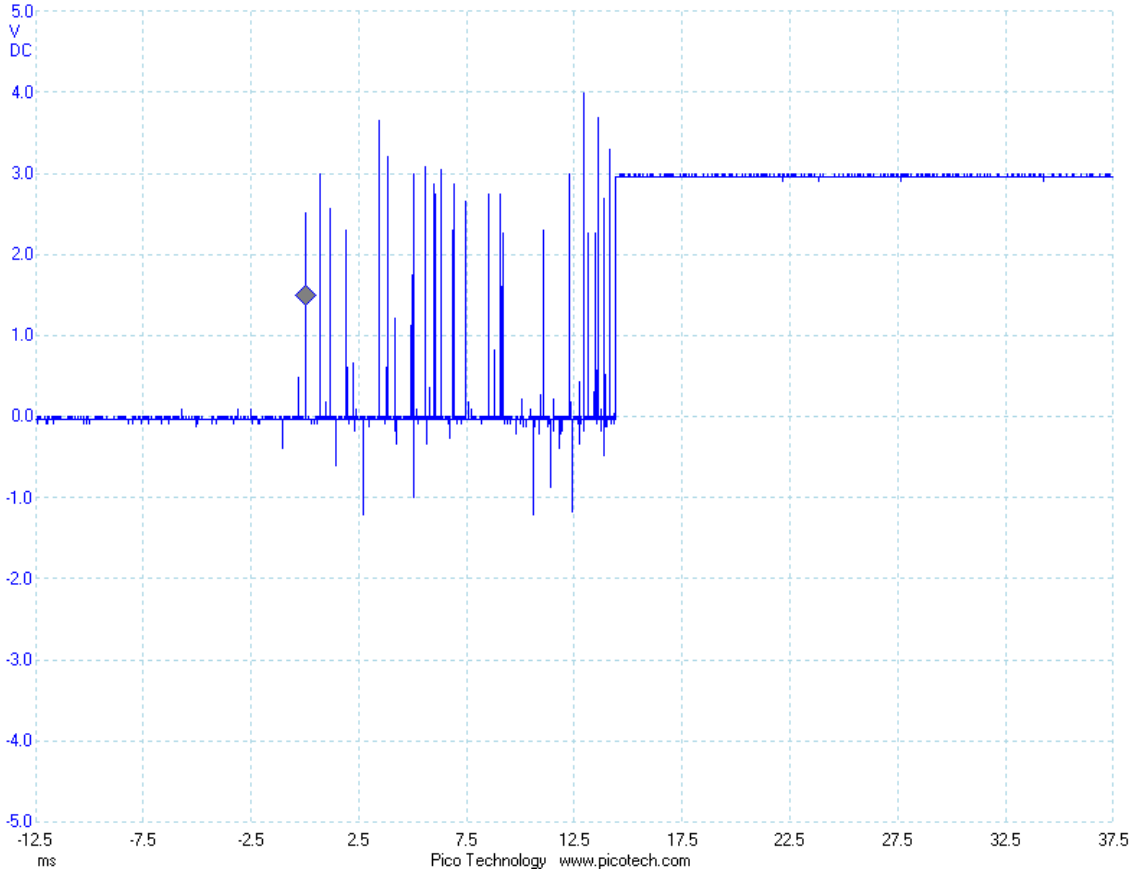



**Figure 3.4: Ringing Effect On Comparator Output.**

There are several ways to handle this ringing effect; however, the two ways considered were software filtering and hardware filtering. Software filtering entails that, after the first initiation, the interrupt capability remain disabled until a certain amount of time has passed after which ringing should have stopped. The disadvantage to this technique is that it requires a timer, thus, implying extra power consumption.

Another disadvantage is having to proceed with preliminary testing in order to choose the right amount of time to wait before re-enabling the interrupt. Therefore, to avoid these disadvantages, this study chose to utilize the hardware technique, which

creates hysteresis by allowing dynamic switching of the reference voltage. Basically, two reference voltages are connected to a switch controlled by the output of the comparator.

In this thesis's implementation, the two voltages utilized were ¼ Vcc and ¾ Vcc. The idea is that when the output of the comparator is 0, the ¾ Vcc reference is used, so that the comparator output will not change to 1 unless the input amplitude rises above ¾ Vcc. When the output of the comparator is 1, the ¼ Vcc reference is used so that the comparator output will not change to 0 unless the input amplitude falls below ¼ Vcc. This allows a fluctuation of margin of (¾ Vcc – ¼ Vcc) = ½ Vcc as illustrated in Figure 3.5.

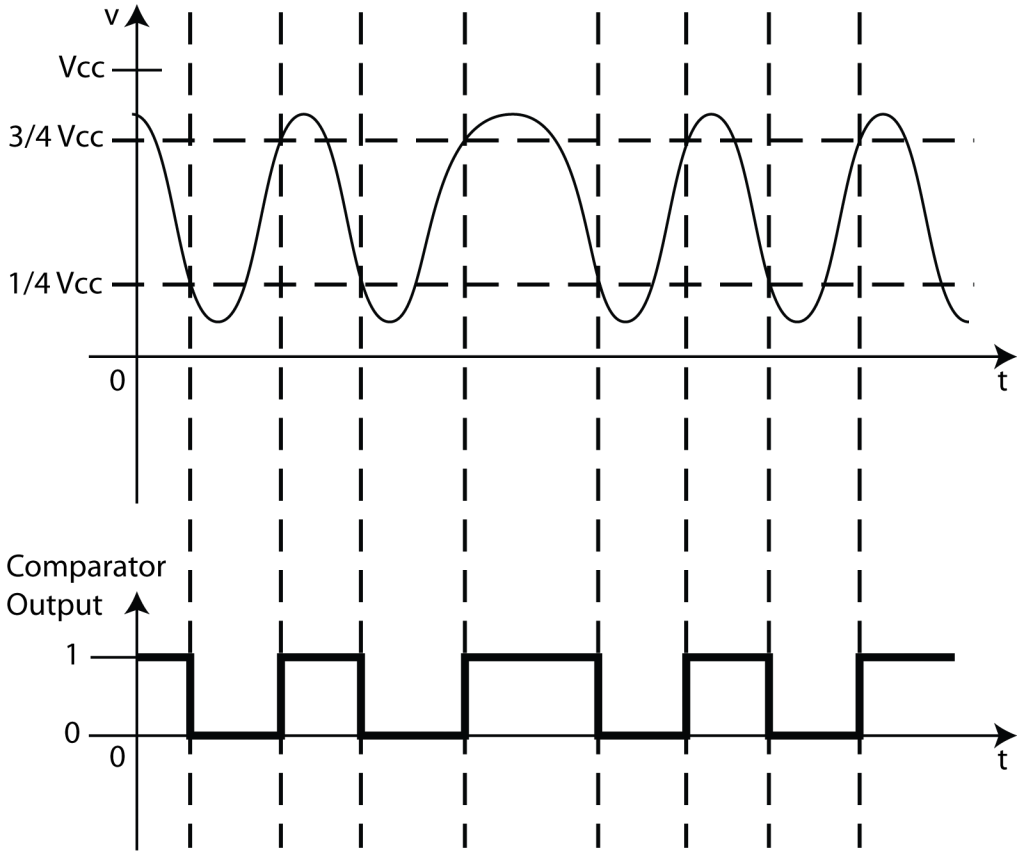



**Figure 3.5: Hardware Hysteresis To Create Comparator Output Stability.**

Lamentably, there is a limitation introduced by this comparator breath-detection implementation scheme: the amplitude of the respiratory signal must be guaranteed to cross the ¼ Vcc and ¾ Vcc amplitudes or else the breath will not be detected properly. This thesis assumes that the ASP is able to condition the respiratory input to meet this

criterion; however, this may not be the case in the end and these settings may need to be revisited when the input is finalized.

Also, in retrospect, future studies might want to directly measure the power that would have been used if a typical ADC implementation were realized. The rationale of this thesis is based on the fact that the datasheet states that the ADC requires at least 220 μA for 200 ksps sampling with a 5 MHz clock. Although this value is grossly higher than the 0.5 μA cited for the comparator, a true comparison must factor in the usage of the resistance ladder used to generate the ¼ Vcc and ¾ Vcc reference voltages. There is a possibility that, if sampling were executed at the lowest frequency, the amount of ADC current draw would be comparable to that of the comparator with the resistance ladder enabled. This thesis conjectures that the ADC will still require more power than the comparator since CPU comparisons are needed to determine whether a breath has occurred. Hypothetical proof of this claim is later demonstrated by the power measurement results in Chapter 4.

Before explaining particular low-power considerations of the breath counting algorithm, it is best to first elucidate its expected implementation functionality. Figures 3.6 and 3.7 present an overall state diagram of the SoC code implementation and a flowchart of the algorithm's general operation, respectively. For the most part, the SoC remains in LPM unless a comparator or timer interrupt occurs. Figure 3.8 depicts a crude simulation of the subsystem's general behavior with a 10-s transmission interval. A single 16-bit timer module is used to notify the subsystem of when it needs to transmit its RR estimate. The breath-detection interrupts caused by the comparator are counted until an interrupt from the 16-bit timer occurs. Then, depending on if rate projection is utilized, the comparator-interrupt count is either sent as-is (i.e., RF transmission is set to every minute) or multiplied by a certain factor prior to transmission (i.e., rate projection is selected).

The top plot in Figure 3.8 illustrates the sinusoidal input, the comparator output, and the 16-bit timer interrupt set to occur every 10 seconds, respectively; the second plot depicts the number of breath-detection interrupts stored at any given time for this 10-s rate projection setting; and, the third plot shows the error between value calculated by the

algorithm versus the actual rate of the sinusoidal respiratory signal. For the interested, Figure 3.8 was generated by MATLAB code available in Appendix B.

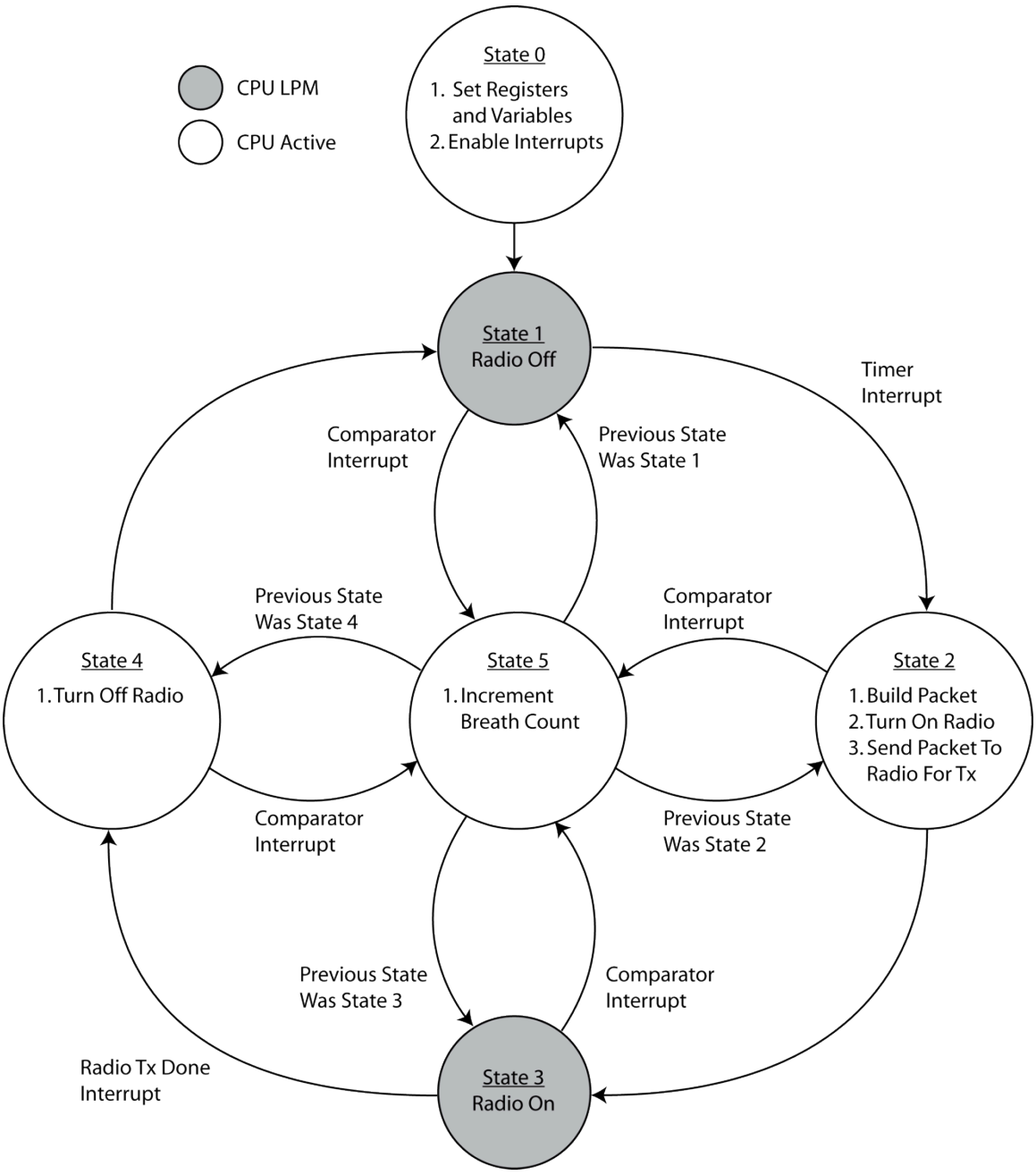



**Figure 3.6: Breath Counting Algorithm State Diagram.**

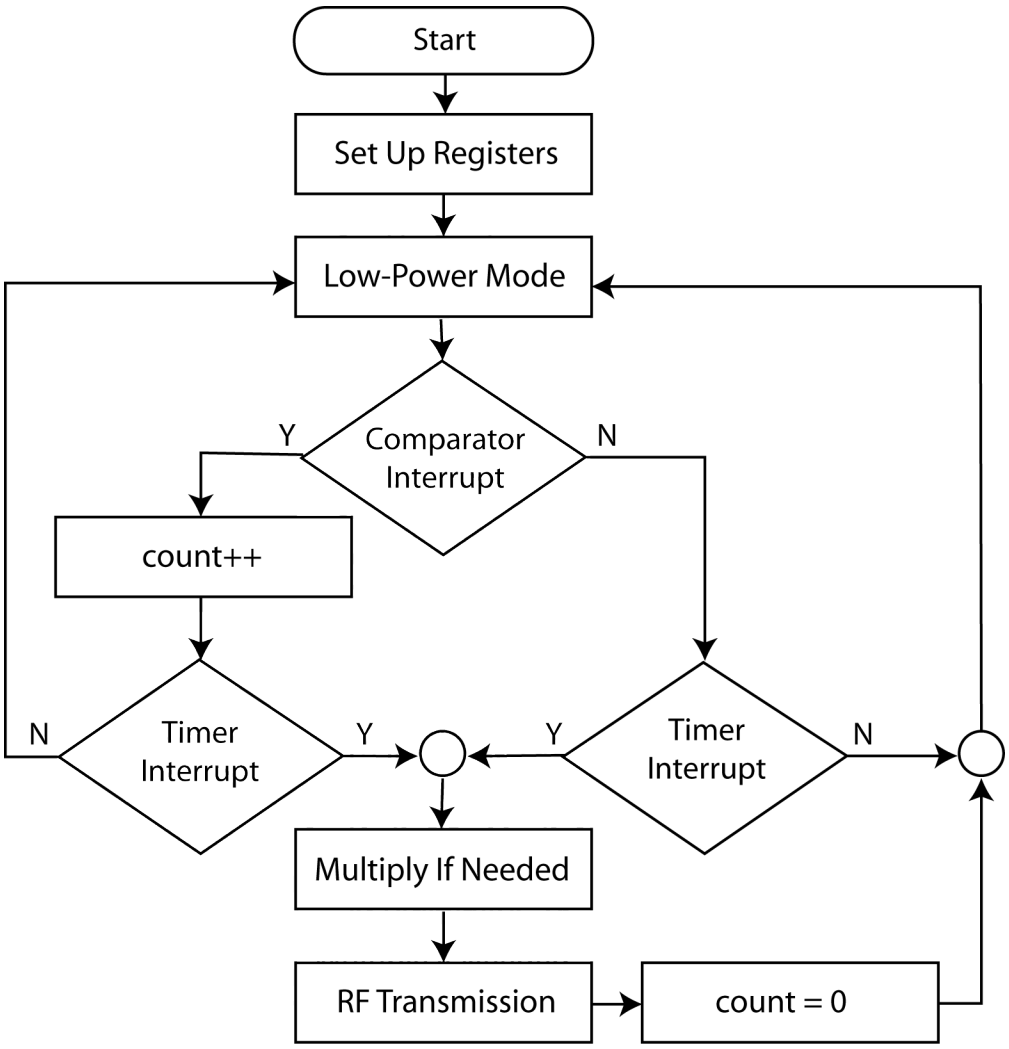



**Figure 3.7: Breath Counting Algorithm Flowchart.**

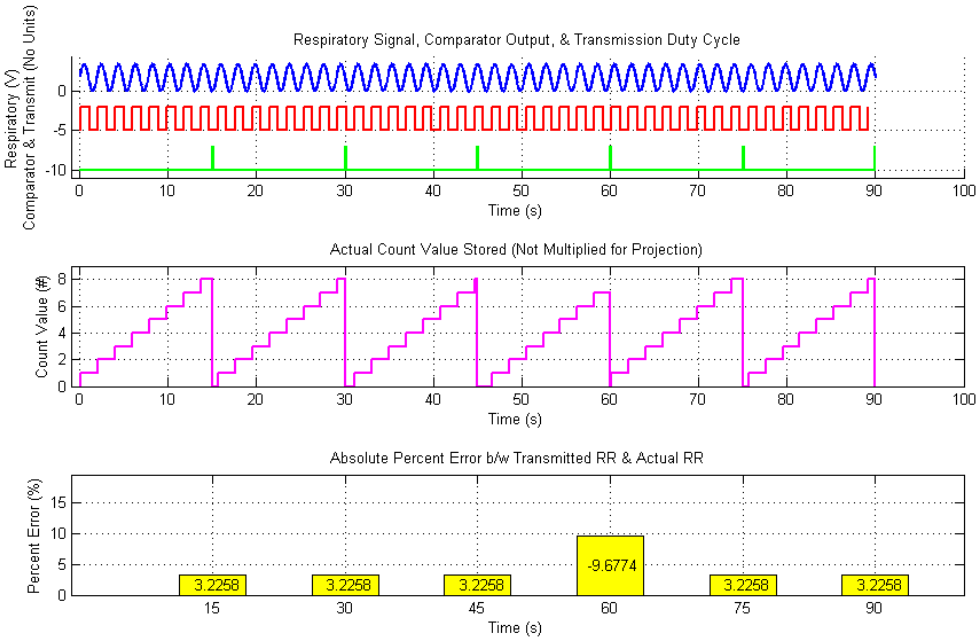



**Figure 3.8: Breath Counting Algorithm Implementation Simulation.**

There were two variations to the final breath counting implementation that the author considered: (1) moving the multiplication operation to the receiver, and (2) withholding transmission even when rate projection is used. The first variation relies on the ability of matching the transmitter and receiver so that the receiver multiplies by the correct rate projection factor. The transmitter program was coded for this option; however, power measurements were not pursed since, as will be seen in Chapter 4, the multiplication operation has a relatively minimal effect on the average power consumption. Also, such an implementation is slightly discouraged based on the inconvenience of device matching, which can cause issues with the transmitter's integration into a larger monitoring network.

The second variation is that, instead of ending the breath count and transmitting at the same time, the count could occur at an interval shorter than the transmission interval. For instance, counting could occur for only 10 seconds, but be projected and transmitted 1 min later. Although, this diminishes the frequency of transmissions, it requires additional SoC effort. For instance, one implementation of this technique would require a second timer to time the 10-s interval in addition to the 1-min timer for the RF transmission. A second implementation would involve constantly polling the timer value

to see if 10 seconds has passed. Given the drawbacks to these variations, the author felt that they should not be pursued further in this thesis.

The second algorithm implemented was the breath interval timing algorithm, whose overall state diagram and general operation flowchart are available in Figures 3.9 and 3.10, respectively. As with the breath counting algorithm, a 16-timer was utilized for RF transmission notification; however, instead of taking note of every comparator interrupt, the implementation only marked the times for the first two breaths taken after a "previous" transmission. The subtraction to find the time elapsed between breaths and the division to obtain the proper rate units were set to transpire when a timer interrupt ensued.

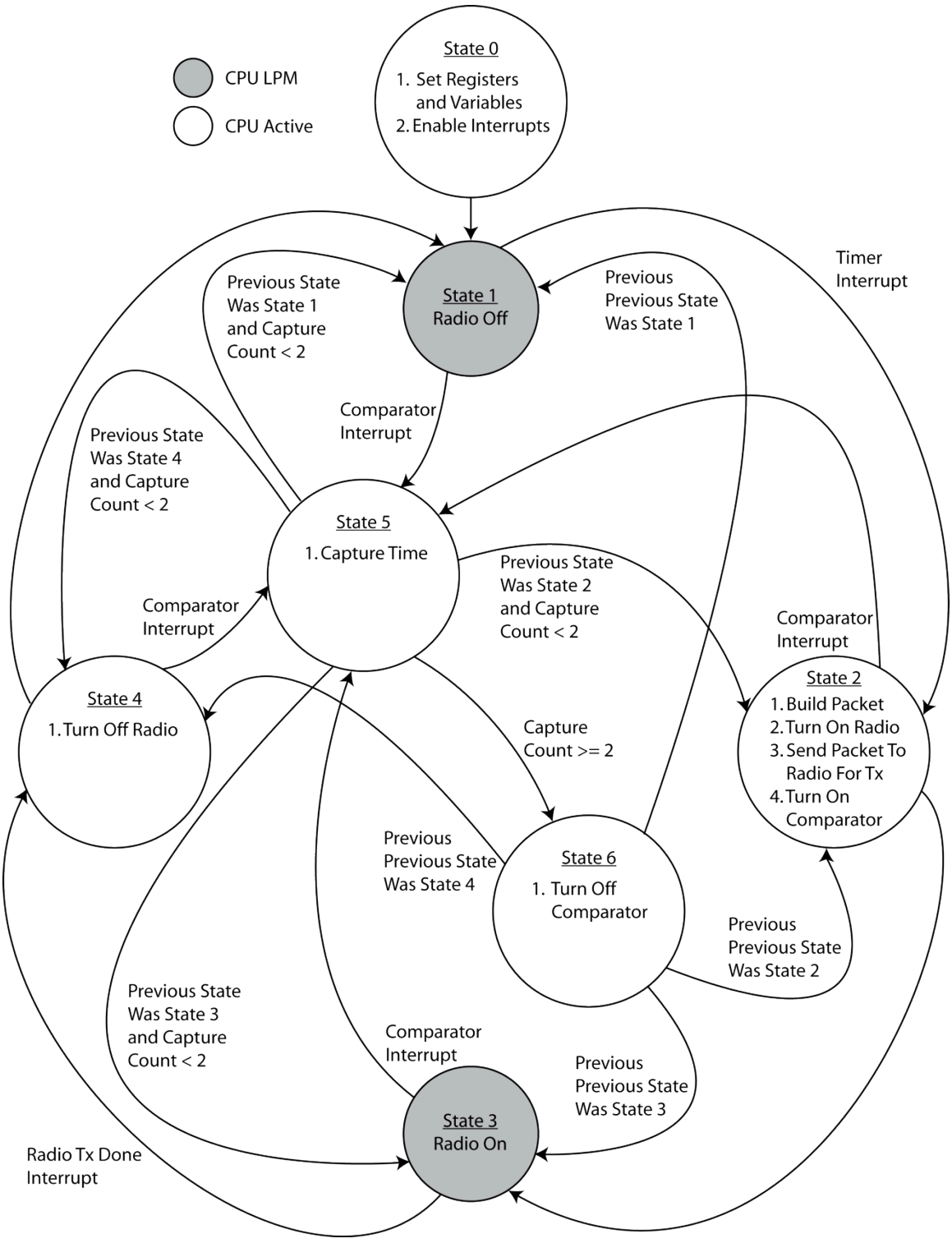



**Figure 3.9: Breath Interval Timing Algorithm State Diagram.**

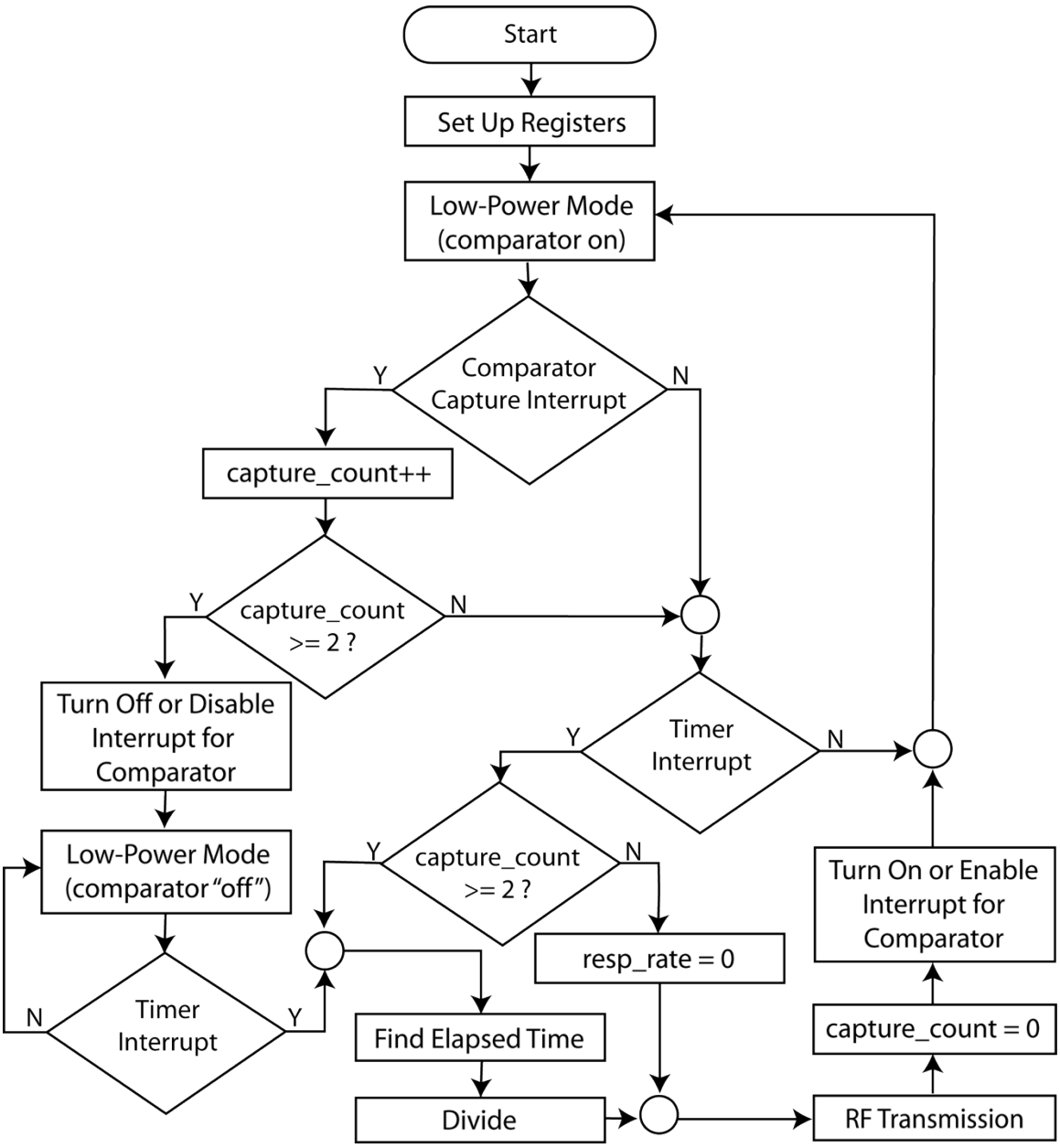



**Figure 3.10: Breath Interval Timing Algorithm Flowchart.**

A crude simulation with an RF transmission every 10 seconds is depicted in Figure 3.11. The first plot displays the sinusoidal respiratory input, the output of the comparator, and the transmission timer interrupt, respectively; the second plot exhibits the estimated RR, which is calculated, transmitted, and reset at each timer interrupt; and, the third plot reveals the error between the calculated rate and the actual rate. For the inquiring, Figure 3.11 was generated by MATLAB code available in Appendix B.

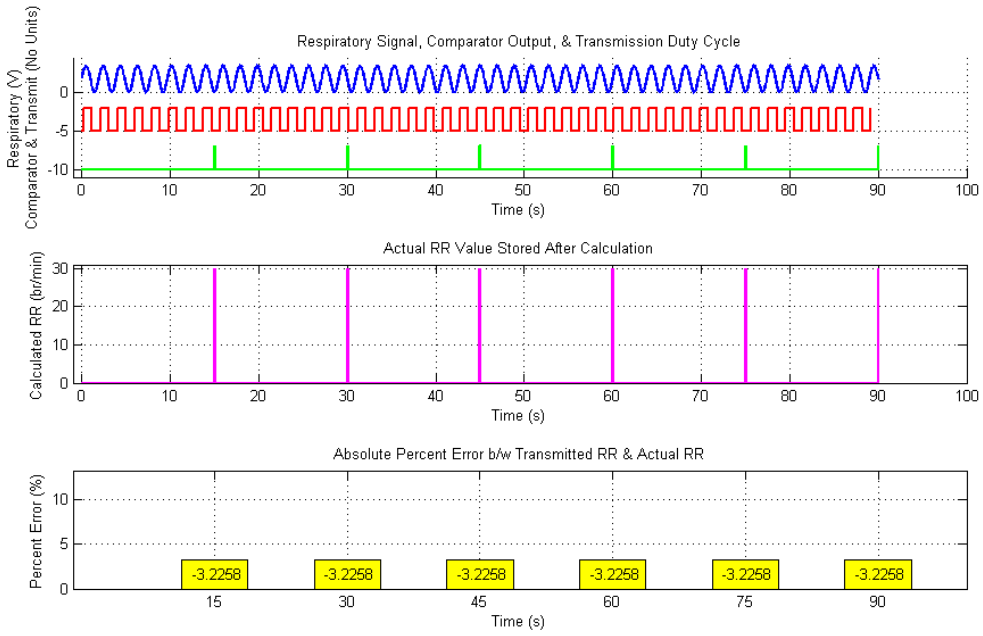



**Figure 3.11: Breath Interval Timing Algorithm Implementation Simulation.**

There are two implementation variations that the authored considered for the breath interval timing algorithm: (1) move the complex math to the receiver, and (2) toggle the comparator on and off. The first variation is similar to one of the variations proposed for breath counting, where computational work is moved to the receiver. However, there is the same drawback of device matching in which the receiver will need to know what factor to use when proceeding with the division (this depends on the timer clock source settings). In addition to this disadvantage, the transmitter would also have to send one to two 16-bit values via RF rather than the single byte containing the respiratory rate; either the subtraction is done prior to transmission and the difference is sent, or the two marked times are transmitted "raw". Therefore, since additional RF transmission time was involved, this variation was not pursued further.

In contrast, the second variation considered provided good low-power incentive to warrant implementation. Since only the first two breaths after an RF transmission are required for calculations, the services of the comparator are not required following the beginning of the second breath until the time after the next transmission. The regular implementation of the breath interval timing algorithm deals with this by simply

disabling the comparator's ability to interrupt the SoC and wake it up from LPM. This means that the comparator is still running wastefully without a purpose. Therefore, it is possible to completely shut the comparator off given the generally low-frequency respiratory input signal.

Normally, in applications that have a high-frequency signal, turning off the sampling component can cause a contingency. More specifically, when the comparator is turned off, it requires 2 μs of settling time for re-initiation. This settling time can confound timing if the signal to be sampled is too fast; however, since RR is generally very slow, a 2 μs delay can be assumed negligible. Since "toggling" the comparator does not affect the RF transmission and does not pose additional mathematical computations, the variation was included in the implemented code. In fact, the difference between the code that has the toggling ability and the code that does not is so minute that both codes are in the same file, and to turn on the toggling capability, the programmer simply needs to define the `TURN_COMP_OFF` macro.

Unfortunately, in addition to the errors mentioned in Chapter 2, both the breath counting and breath interval timing implementations of this thesis impose limitations on the range of RR that can be measured. More specifically, the transmission period chosen creates a threshold interval in which all operations must be carried out or inaccurate information is encountered. For the specific case of 10 seconds between each RF transmission, the breath counting algorithm will estimate 0 br/min if a breath *beginning* (i.e., comparator rising edge interrupt) does not occur within the 10-s interval. This means that 1 br/ 10 s = 6 br/min is the lowest RR that can be detected before transmitting 0 br/min to the receiver.

For the breath interval timing algorithm, two breaths beginnings must occur for a proper estimate before the RF transmission and rate variable reset. Thus, the lowest rate that can be transmitted assuredly is 2 br / 10 s = 12 br/min. It is possible for the breath interval timing algorithm to transmit a lower value when zero-crossings occur since the comparator need only catch the *start* of the second breath versus the entirety of the breath. This means that the display may read 1 br / 10 s = 6 br/min during one reading, but will fluctuate to 0 br/min in the next reading. If an even lower reading is desired, it is

plausible to track occurrences beyond 10 seconds. However, this implementation will require a second timer or variables to track timer overflows; therefore, it was not pursued.

In the overall picture, these readable RR range limitations might be acceptable, since rates below 12 br/min are considered abnormal. A below-12 br/min-threshold alarm might be sufficient enough to alert medical personnel to a patient's degrading health condition; the alert itself might be considered of more value than the actual rate displayed. A synopsis of limitations introduced by the implementations described in this chapter is provided in Table 3.3, and a summarized block diagram of utilized CC430 interconnections is illustrated in Figure 3.12.

**Table 3.3: Implementation Limitations.**

| Limitation | Reason |
|---|---|
| Rate projection/transmission can only occur at either 10-s, 15-s, 30-s, or 1-min intervals. | Only macros for these timing settings are coded; further hand-calculations are required if other intervals are desired. |
| Amplitude of conditioned respiratory signal from ASP must oscillate between values higher than ¾ Vcc and values lower than ¼ Vcc for a breath to be detected. | Comparator settings to create hysteresis are currently set to these thresholds. |
| Breath counting with a 10-s transmission can measure breathing as low as 6 br/min. | The transmission interval requires one breath beginning to occur every 10 s. |
| Breath interval timing with a 10-s transmission can measure breathing as low as 12 br/min. It can be even lower, down to 6 br/min, if there are zero-crossings. | The transmission interval requires two breath beginnings to occur every 10 s. |

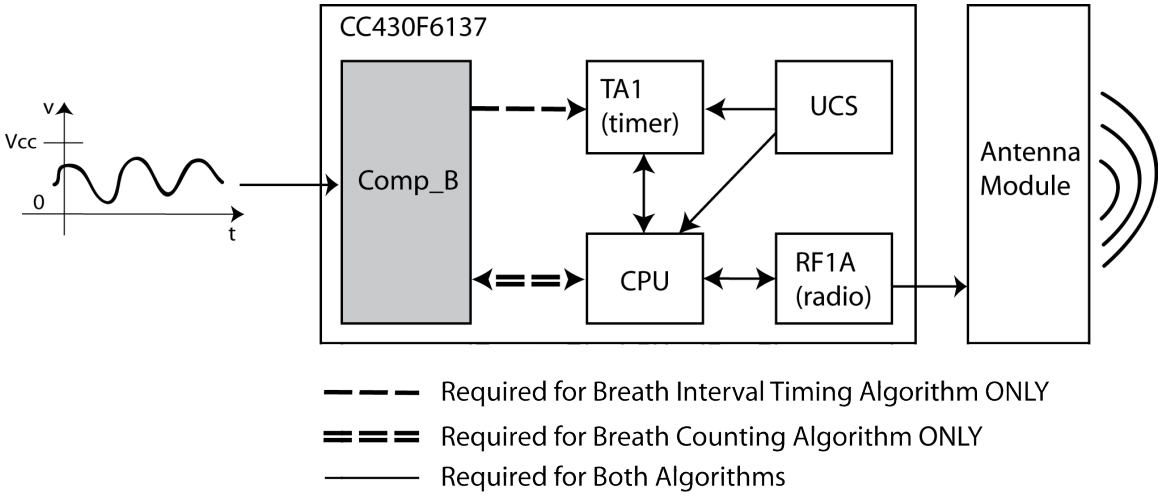



**Figure 3.12: CC430 Implementation Interconnections.**

# 4. Power Measurements and Calculations

Figure 4.1 is a block diagram of the setup used to characterize the power consumption of the implemented algorithms, and Figures 4.2 – 4.5 are photographs of both the receiver and transmitter setups. A PS1502AU supplied 3.5 V of power to the transmitting EM430F6137RF900, and a GoldStar FG-8002 provided a roughly 1-V, 1-Hz sinusoidal wave input with a vertical shift of +1.75 V. Average power consumed by the CC430 was acquired by placing a shunt resistor, R, in series with the EM430F6137RF900, measuring the average voltage over R, calculating the average current through the system, and then multiplying the average current by the voltage over the EM430F6137RF900. A PicoScope 2000 Series PC Oscilloscope was utilized to capture the voltage over R, and a 0.01μF-capacitor, C, was placed in parallel to R, to stabilize the measurement. All SoC C-code programs were written in the Code Composer Studio Version 4 (CCS4) Integrated Development Environment (IDE) and loaded onto the CC430 with an MSP-FET430UIF via JTAG connection.

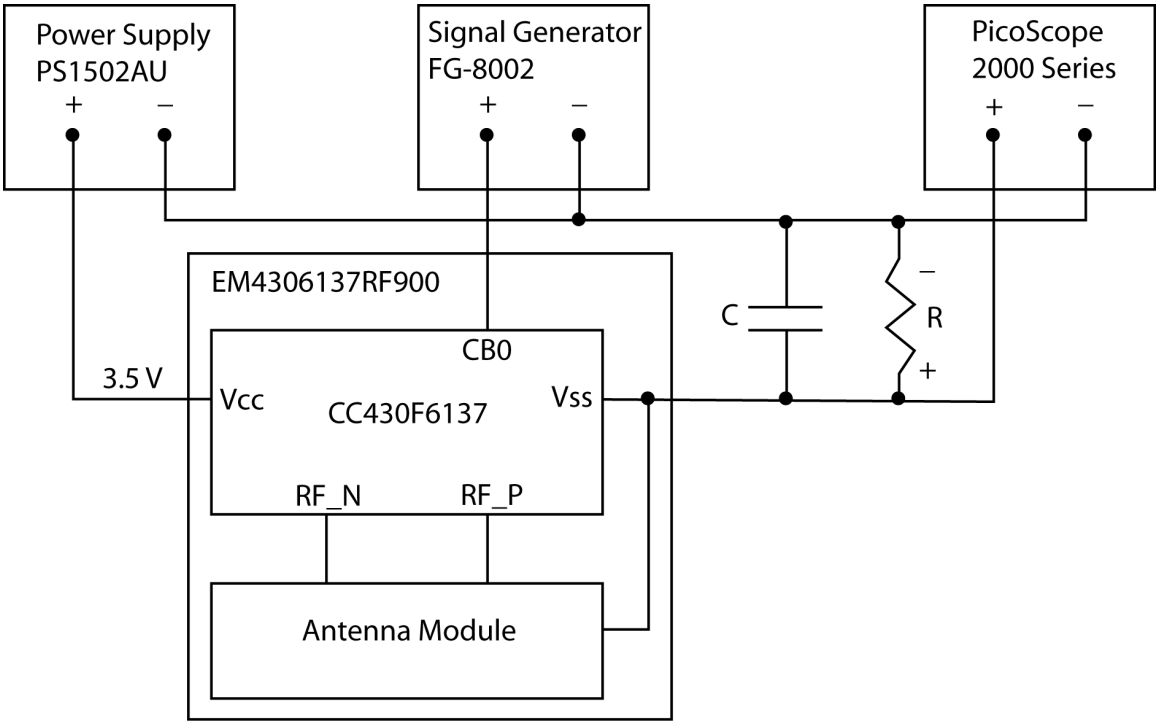



**Figure 4.1: Block Diagram of Measurement Setup.**

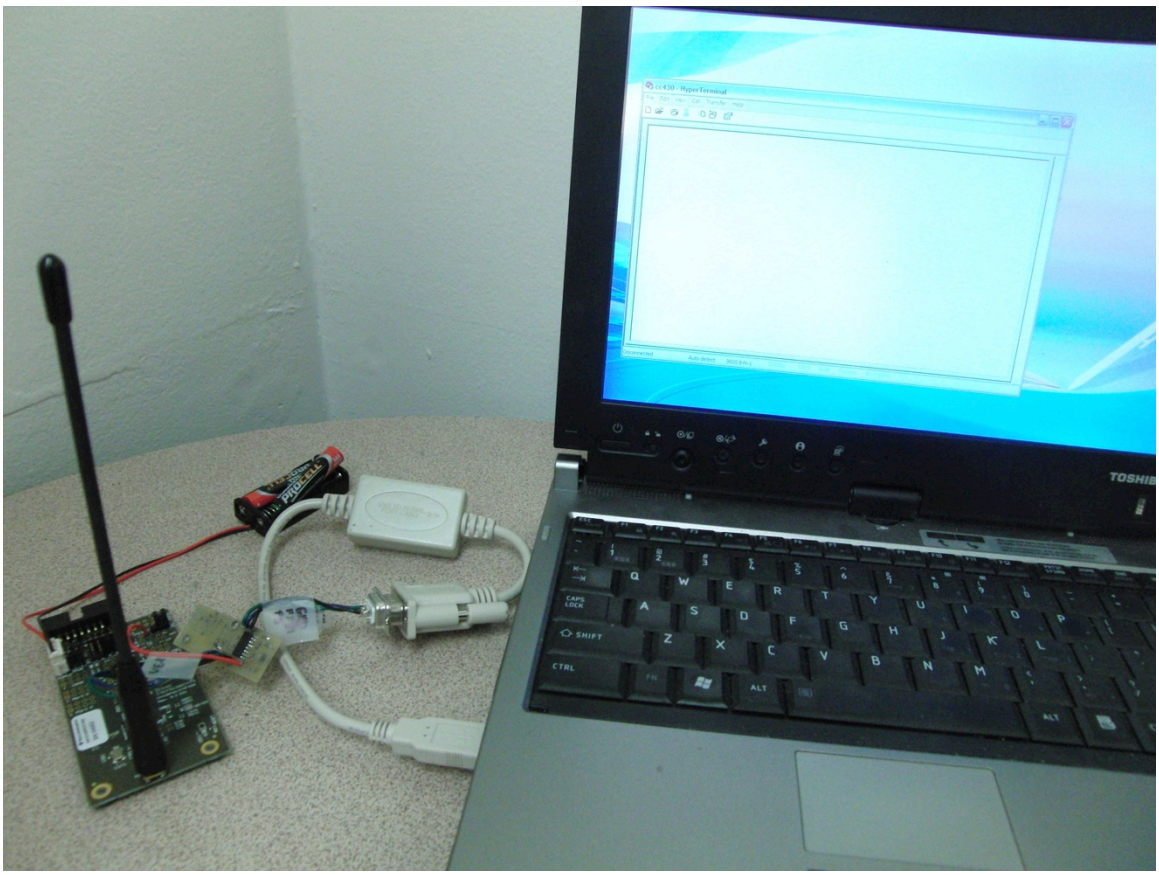


**Figure 4.2: Complete Receiver Setup Photograph.**

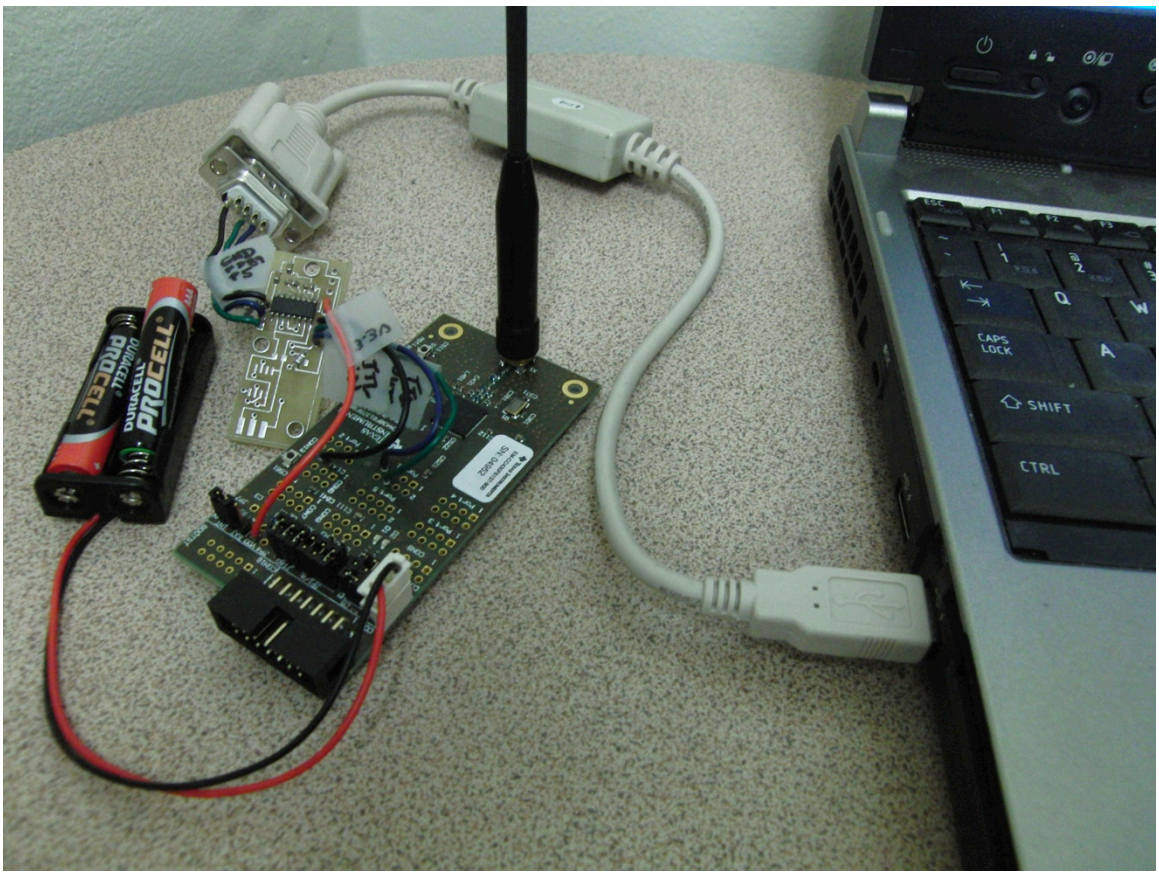



**Figure 4.3: Close-up Receiver Setup Photograph.**

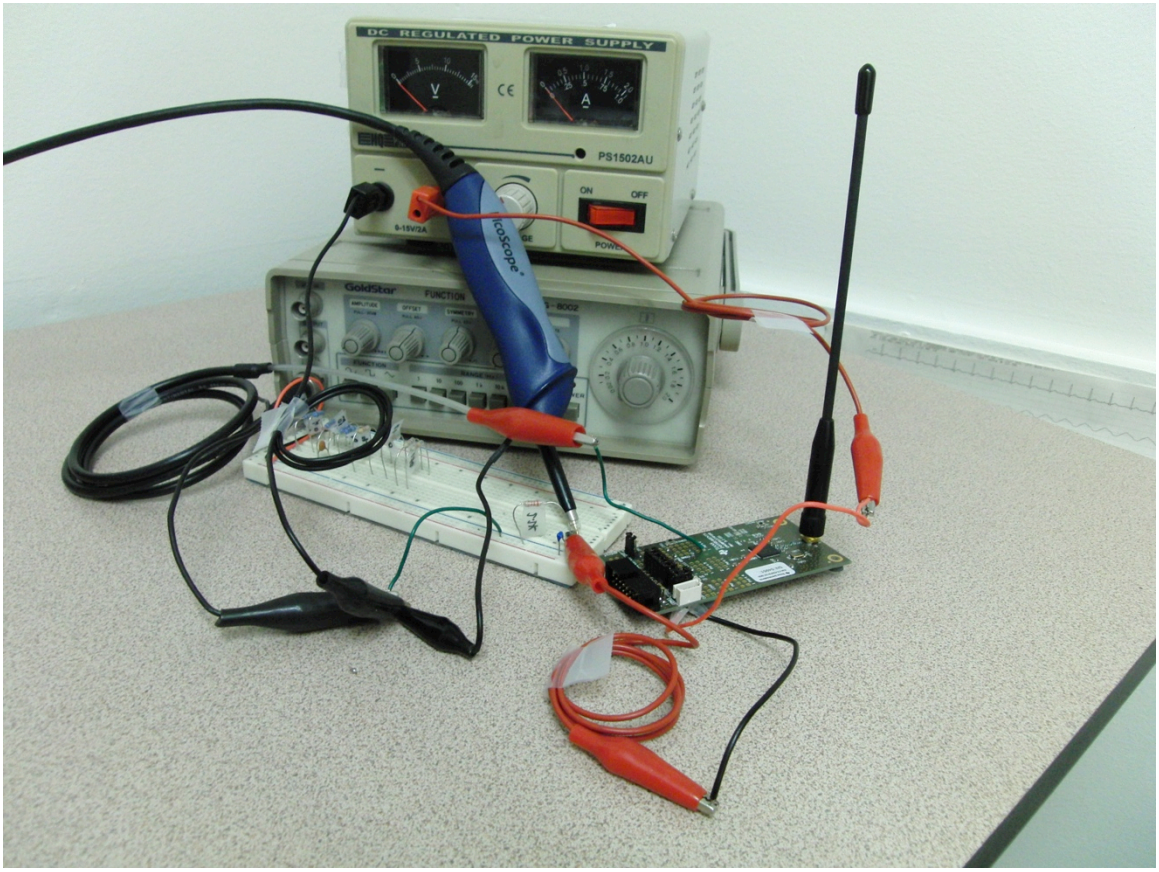



**Figure 4.4: Complete Transmitter Setup Photograph.**

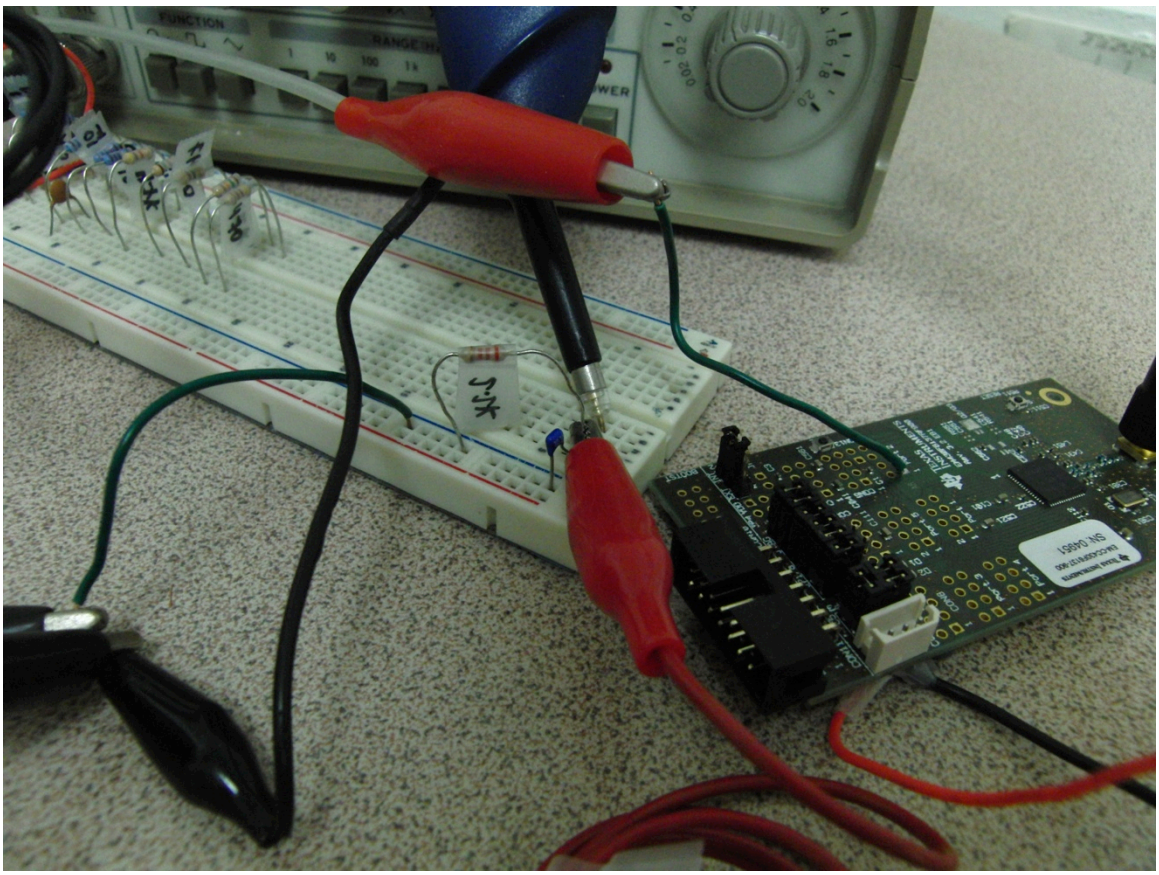


**Figure 4.5: Close-up Transmitter Setup Photograph.**

The possible values for R are dependent on the expected voltage drop and current flow. As discussed in Chapter 3, the voltage required for functionality of the CC430 is 3.0 – 3.6 V. Since the power supplied to the CC430 in this measurement setup is 3.5 V, this allows a maximum voltage drop of (3.5 V – 3.0 V) = 0.5 V over the resistor. Using Ohm's Law over the shunt resistor results in the relation expressed in Figure 4.6.

$$V_{shunt} = I_{shunt} \cdot R_{shunt}$$

**Figure 4.6: Ohm's Law Over Shunt Resistor, R.**

To calculate the maximum value of R that can be chosen, the maximum current required by the SoC must be known. This maximum current is different depending on whether the RF transmission is turned off or not. First, this thesis presents the case in which the RF transmission remains on in order to illustrate how much more instantaneous power is required from the RF portion of the code as opposed to the rest of the algorithm.

Consulting the datasheet and considering all the components used in implementation, the highest current required was found to be the RF transmission current at 36 mA. Substituting this into the equation in Figure 4.6 yields the calculations in Figure 4.7 below.

$$V_{shunt} \geq I_{shunt} \cdot R_{shunt}$$

$$R_{shunt} \leq \frac{V_{shunt}}{I_{shunt}}$$

$$R_{shunt} \leq \frac{0.5\ \text{V}}{36\ \text{mA}}$$

$$R_{shunt} \lesssim 13.89\ \Omega$$

**Figure 4.7: Maximum Shunt Resistor with RF Transmission.**

Given the available parts, a 10-Ω ±1% shunt resistor was chosen. Figure 4.8 depicts the measurement for the breath counting algorithm with an interval of 10 seconds between transmissions. As expected, it is fairly easy to see the voltage over R during the roughly 3.6-ms periods of transmission. However, during all other periods it looks like the voltage is 0 V, which is impossible since the SoC is continuously running. This is because the current consumption of the RF transmission greatly overshadows the current consumption of the other SoC operations. Figure 4.9 shows the expected decreasing correlation between the average frequency-independent transmission power consumption with respect to the transmission interval chosen. In order to better see the consumption of the other operations, a larger value for R is required.

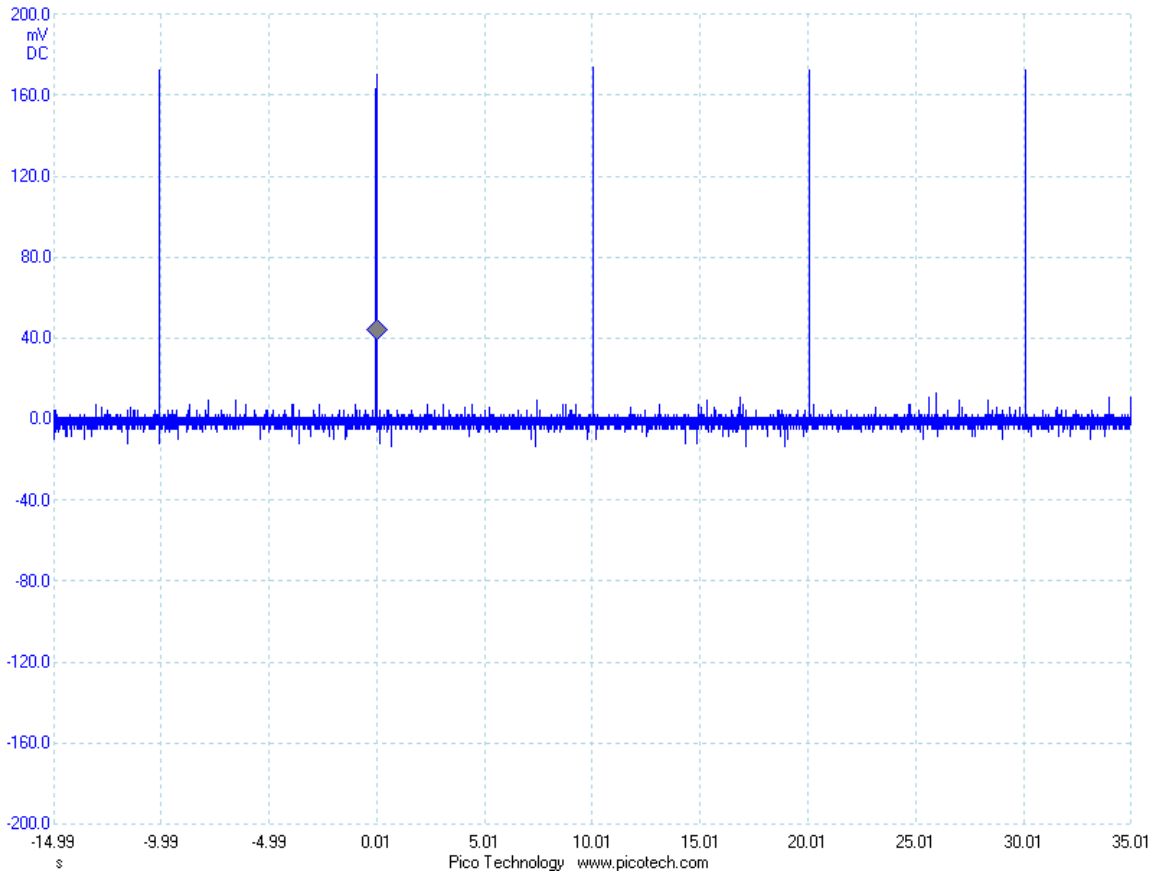



**Figure 4.8: $V_{shunt}$ For Breath Counting with Projection Set at 10 s and R = 10 Ω ±1%.**

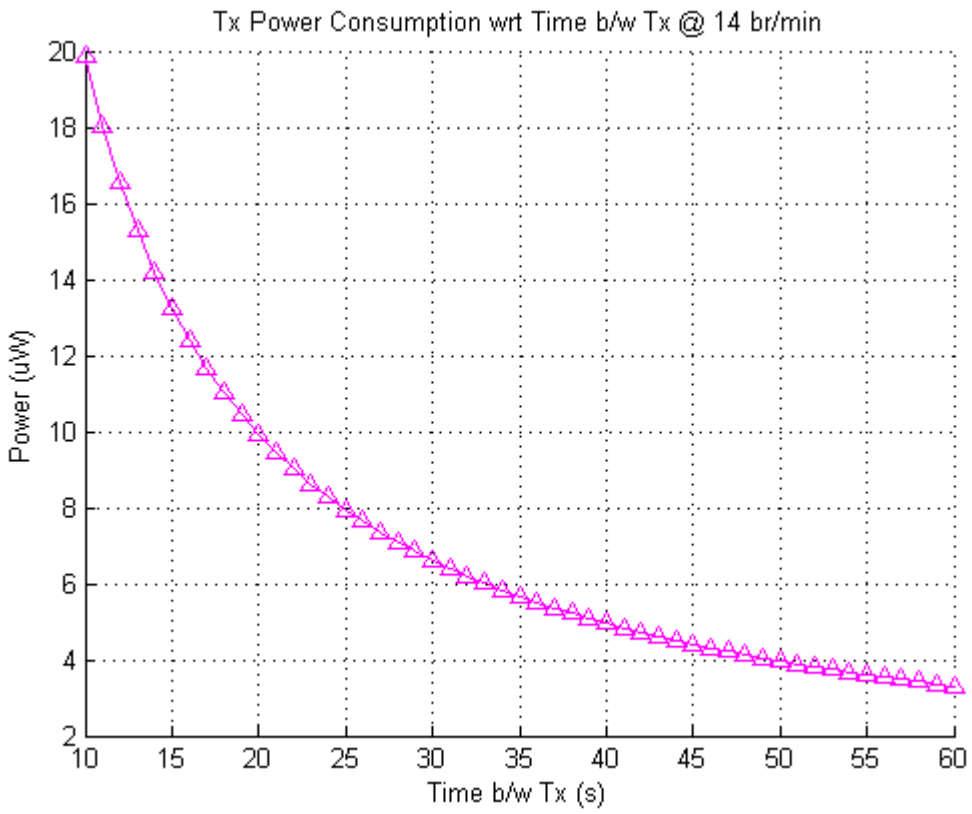



**Figure 4.9: RF Transmission Power Consumption in Relation to Transmission Interval with R = 10 Ω ±1% and 14 br/min.**

Simply selecting a higher value of R creates a voltage drop higher than the 0.5-V limit. Therefore, this study utilized RF transmission and digital extraction logic code isolation to avoid this problem. Turning off the RF transmission code allows a different maximum amount of expected current draw. Revisiting the datasheet reveals that the next current bottleneck is the flash memory, which requires a maximum of 0.30 mA when the PMMCOREV2 setting is used and a 1 MHz clock is utilized. Although 1 MHz is higher than what was implemented in this study's system, it provides a ceiling for the current consumption without RF transmission and proved sufficient enough for resistor calculation purposes. Figure 4.10 illustrates the new calculations.

$$V_{shunt} \geq I_{shunt} \cdot R_{shunt}$$

$$R_{shunt} \leq \frac{V_{shunt}}{I_{shunt}}$$

$$R_{shunt} \leq \frac{0.5\ \text{V}}{0.30\ \text{mA}}$$

$$R_{shunt} \lesssim 1666.67\ \Omega$$

**Figure 4.10: Maximum Shunt Resistor without RF Transmission.**

With the parts on hand, a shunt resistor of R = 1 kΩ with ±1% tolerance was chosen. However, for both the breath counting algorithm and breath interval timing algorithm, any features indicating processing were still indistinguishable from noise. To deal with this, increasing values of resistors were placed in the subsystem iteratively until either too much noise occurred, $V_{shunt}$ exceeded 0.5 V, or an acceptable voltage range was reached that illuminated processing features.

Figure 4.11 illustrates the voltage measurement for the breath counting algorithm with R = 2.2 kΩ ±5%, a transmission interval of 10 seconds, and a breathing rate of 90 br/min. (Note: The high rate of 90 br/min was used with this and the following estimations so that as many features as possible of the voltage signal could be captured on a single measurement screen.) The smaller triangular features are each caused by a count action, while the larger triangle is caused by a multiplication. As mentioned in Chapter 2, the number of counts within the transmission interval and respiratory accuracy depends on the input frequency of the signal. Similarly, the power consumption is also dependent on the RR. Figure 4.12 provides a generated estimation of the frequency dependence based on Figure 4.11 features.

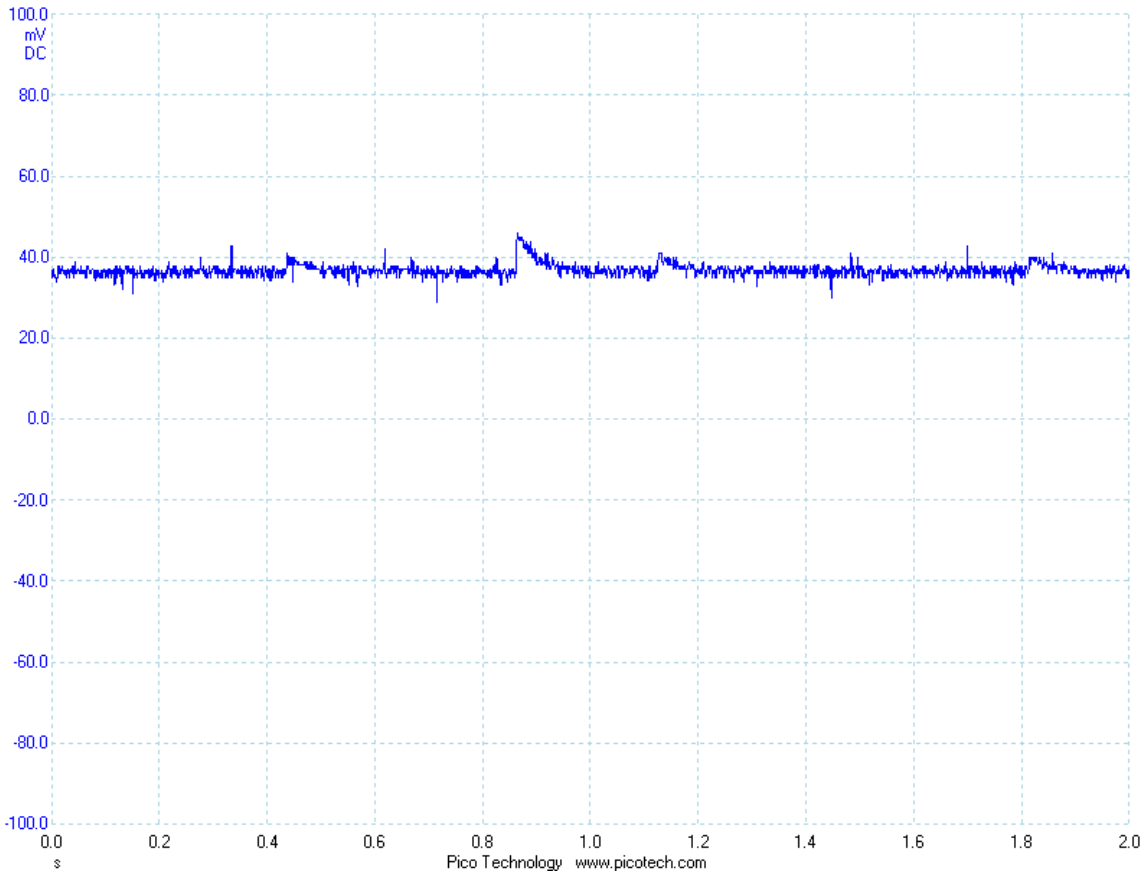



**Figure 4.11: $V_{shunt}$ with Transmission Off, Projection Set at 10 s, Respiratory Rate at 90 br/min, and R = 2.2 kΩ ±5%.**

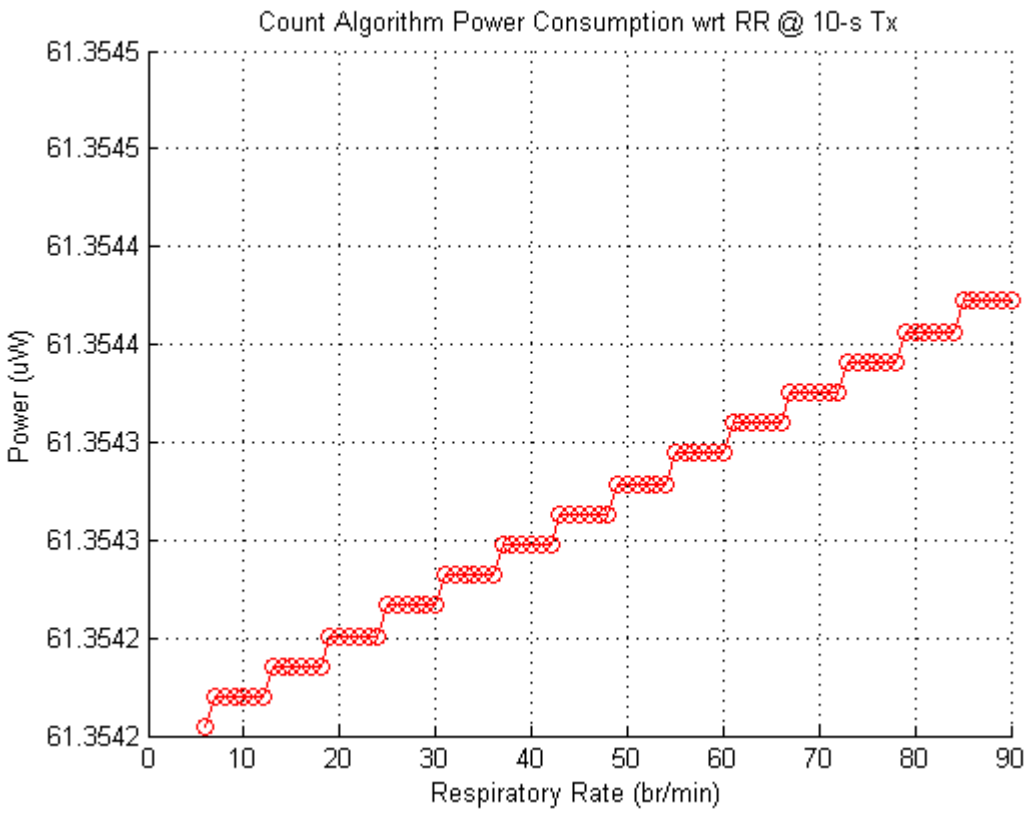



**Figure 4.12: Breath Counting Algorithm Power Estimation in Relation to RR with Transmission Off, Projection Set at 10 s, and R = 2.2 kΩ ±5%.**

The reason for the step-like shape of Figure 4.12 is that for some RR, there are only partial breaths taken. Since the current SoC implementation only considers the start of the breath versus the whole breath as a count initiator, all partial breaths were rounded up to the nearest integer breath count in the estimation program. Thus, several RR are expected to output the same amount of power in this overestimation scheme.

Plotting the power consumption of the breath counting algorithm against the transmission interval time results in Figure 4.13. The general decreasing trend is to be expected; however, the jaggedness of the plot is interesting. The reason for the small spikes is the transitions to an interval just long enough to capture another breath. For instance, if the transmission interval contains three breath counts, the power consumption of these breaths is dispersed over the whole interval. If the interval is increased slightly, the breaths are distributed over a longer time period, and the average power consumption decreases. However, when the interval is extended just long enough to incorporate just

one more breath (i.e., capturing a moment at 100% duty cycle), then there is a sudden increase in the amount of power that must be dispersed over the slight increase in time interval. The height of the spikes is related to a combined effect from the power consumption of a single count, the respiratory rate of the signal, and the transmission interval chosen.

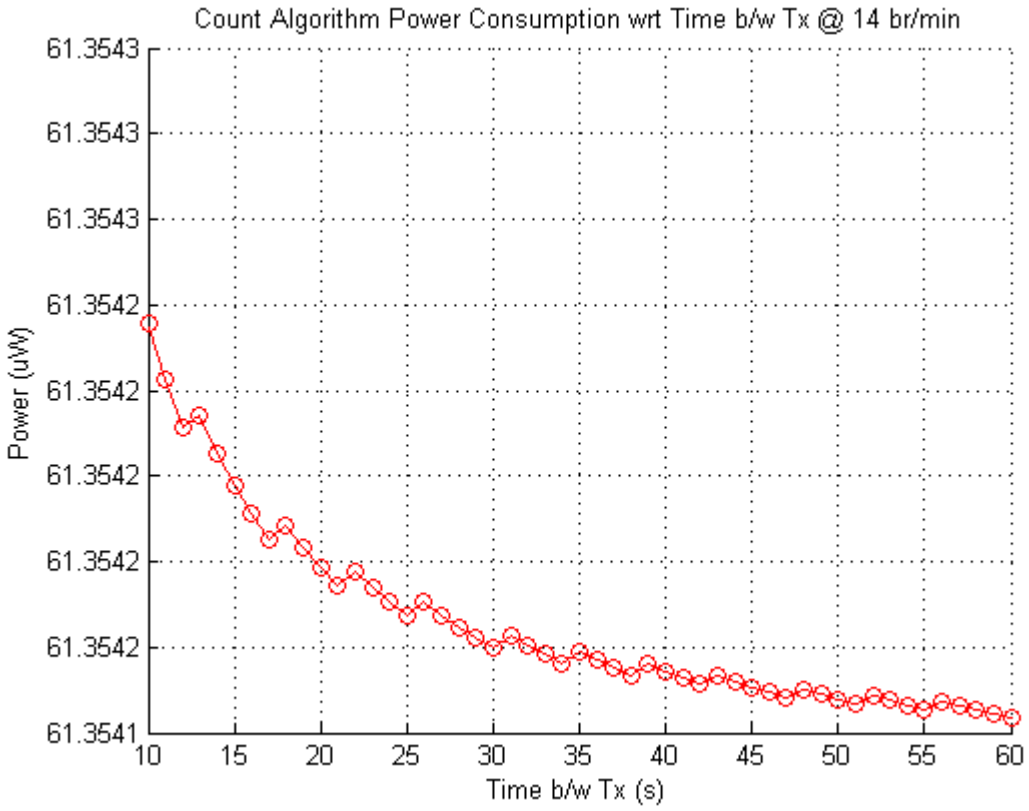



**Figure 4.13: Breath Counting Algorithm Power Estimation in Relation to the Time Between Transmissions with Transmission Off, R = 2.2 kΩ ±5%, and 14 br/min.**

In contrast, Figures 4.14 and 4.15 illustrate $V_{shunt}$ for the breath interval timing algorithm when the comparator is constantly on and the comparator is toggled, respectively. The two figures use the same value for R as Figure 4.11, and there is a large spike indicating the division that occurs right before the supposed transmission. The other smaller spikes are caused by the two required capture actions. In addition, in Figure 4.15, there is a stark voltage difference between the instances when the comparator is on and the when it is off.

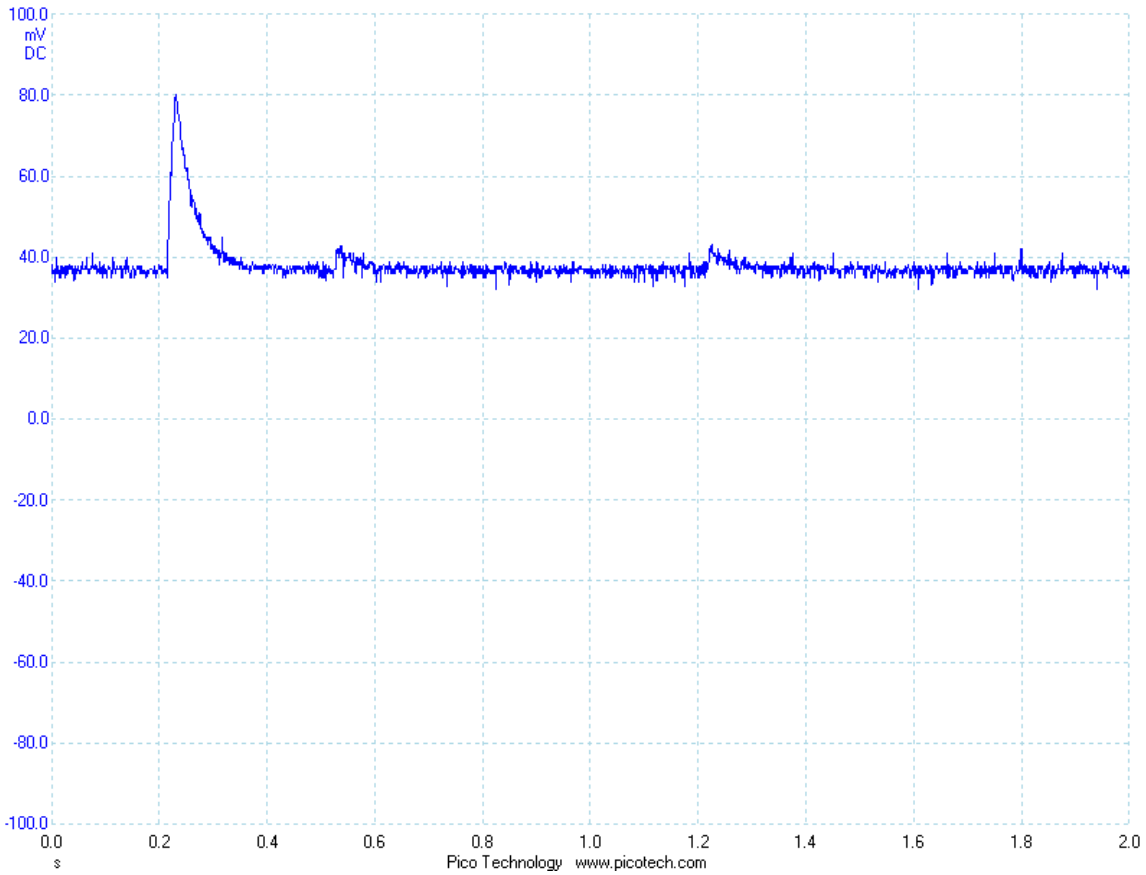



**Figure 4.14: $V_{shunt}$ with Transmission Off, Transmission Set at 10 s, Comparator Constantly On, R = 2.2 kΩ ±5%, and 90 br/min.**

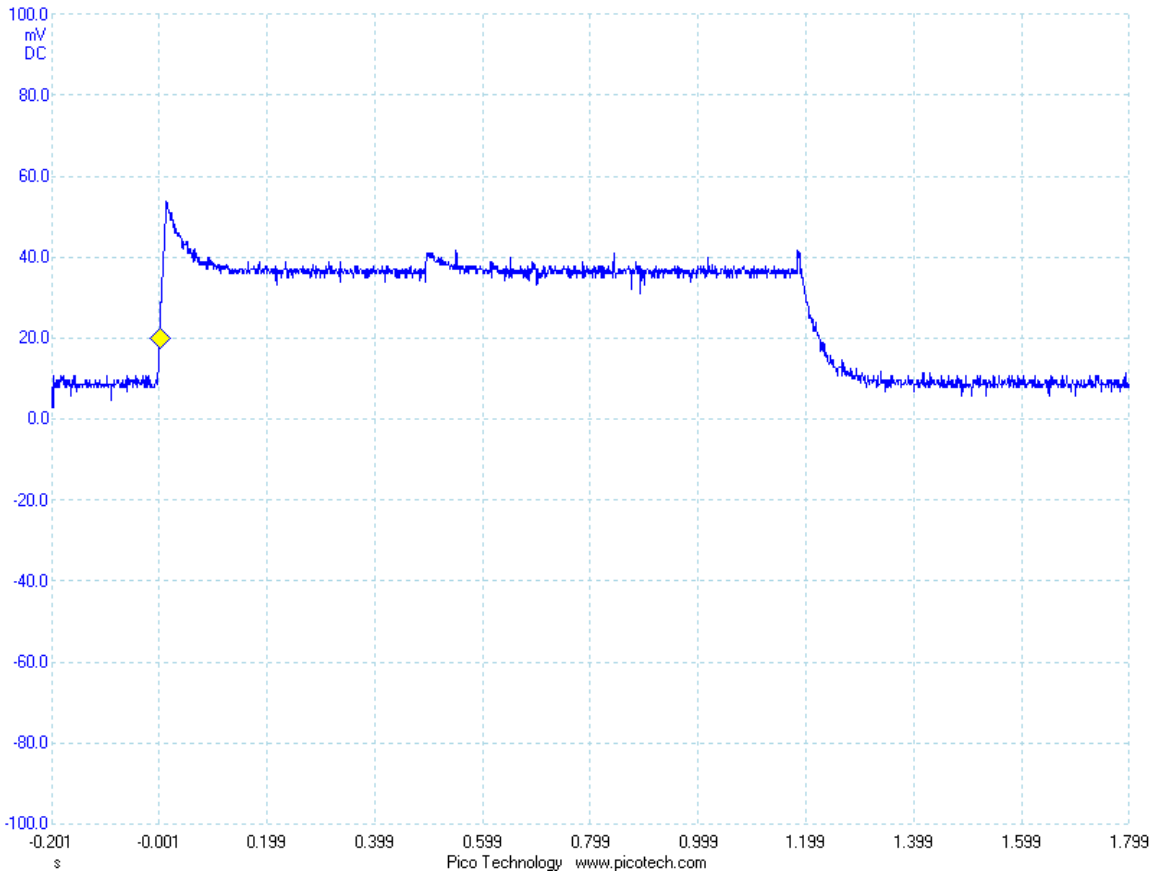



**Figure 4.15: $V_{shunt}$ Close-up with Transmission Off, Transmission Set at 10 s, Comparator Toggled, R = 2.2 kΩ ±5%, and 90 br/min.**

When the comparator is constantly on, the power consumption of the breath interval timing algorithm is independent of frequency since there will always be two captures and one division per period. However, as is manifested in Figure 4.16, its power consumption decreases smoothly with the increase in transmission interval length. As opposed to the breath counting algorithm, there is no way to cause another capture or division spike as the time interval to average over increases. This means that there is not possibility to add another power consuming feature per period, and thus, Figure 4.16 does not have the jaggedness of Figure 4.13.

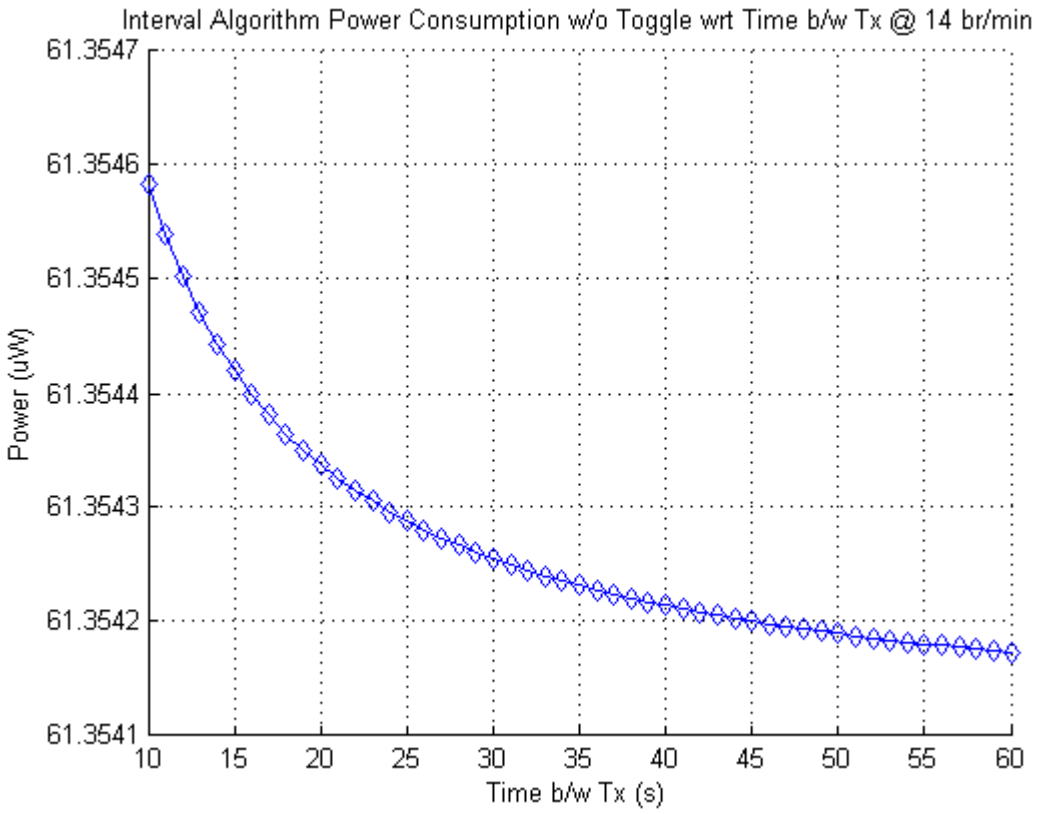



**Figure 4.16: Breath Interval Timing Algorithm Power Estimation with the Comparator Constantly On in Relation to the Time Between Transmissions with Transmission Off, R = 2.2 kΩ ±5%, and 14 br/min.**

When using the toggling method with the breath interval timing algorithm, the time per period that the comparator is on is variable. Figure 4.17 illustrates this attribute. In fact, the time the comparator is on is directly related to the breathing rate of the patient. It only remains on long enough to capture the start point of two breaths; therefore, the longest time it should remain on is for roughly two breaths, and the shortest time should be roughly one breath. The longest time occurs when a breath happens just before a transmission, when the breath count is reset, and the algorithm still needs to capture two breaths; whereas, the shortest time occurs when the breath happens just after transmission and the algorithm only needs to capture one more breath. With this sort of relation, the faster the RR, the shorter the time between breaths is, and the shorter the amount of time the comparator is left on. Since the power consumption is directly proportional to how long the comparator is on, faster breathing theoretically utilizes less power.


**Figure 4.17: $V_{shunt}$ with Transmission Off, Transmission Set at 10 s, Comparator Toggled, R = 2.2 kΩ ±5%, and 90 br/min.**

Figures 4.18 and 4.19 depict the power consumption ranges of the breath interval timing algorithm with comparator toggling in relation to the respiratory rate and transmission interval, respectively. As can be expected, the power consumption decreases as the respiratory rate or transmission interval increases. These curves are smooth for the same reason as the plot in Figure 4.16 when the comparator was constantly on. Since the power consumption per period can be variable, both Figure 4.18 and 4.19 depict a range of possible power consumption values rather than a single curve. To generate these depictions, the transmission interval was fixed to 10 seconds for the former, and the respiratory rate was fixed to 14 br/min for the latter.

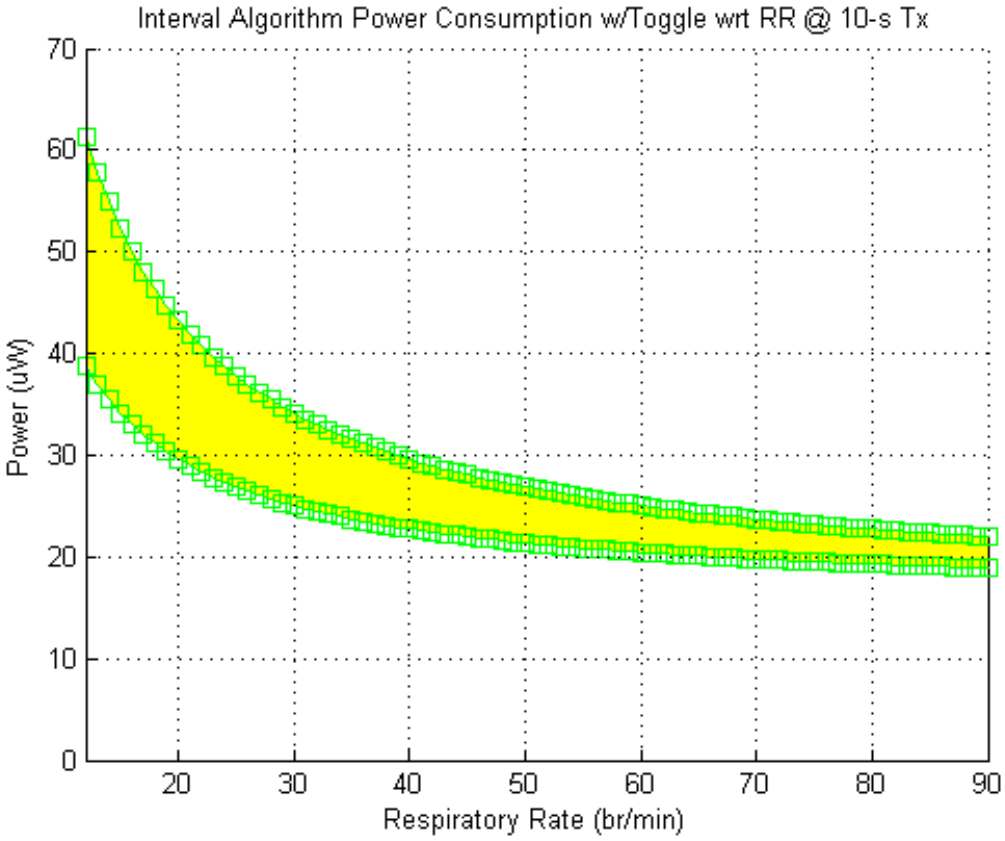



**Figure 4.18: Power Consumption of Breath Interval Timing Algorithm with Transmission Off, Transmission Set at 10 s, Comparator Toggled, and R = 2.2 kΩ ±5%.**

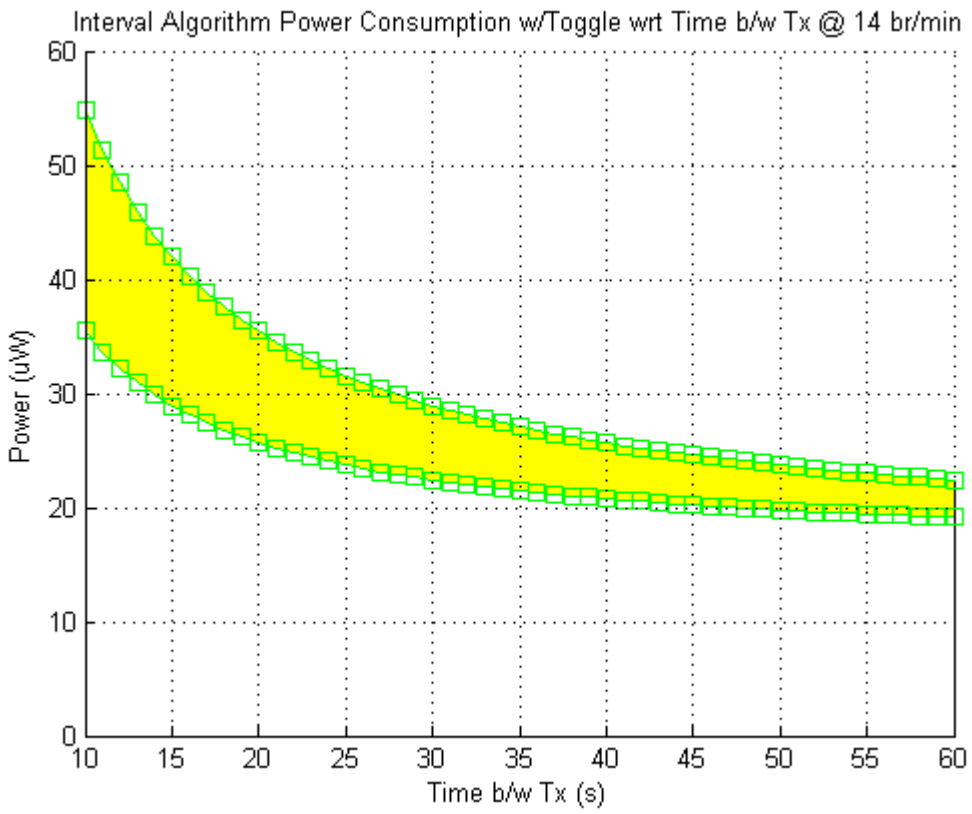



**Figure 4.19: Power Consumption of Breath Interval Timing Algorithm with Transmission Off, Respiratory Rate Set at 14 br/min, Comparator Toggled, and R = 2.2 kΩ ±5%.**

For comparison purposes, all of the previous power consumption plots were combined to form Figures 4.20 and 4.21. Although, it is a little difficult to see, both figures depict the breath counting and breath interval timing algorithms as having a relatively constant power consumption around 60 μW. Both figures also demonstrate the power savings of the breath interval timing toggle method from when either RR or the transmission interval increases. The main difference between the plots is that Figure 4.20 illustrates how the power consumed by the RF portion does not vary with respiratory input, whereas Figure 4.21 conveys that increasing the time between transmissions from 10 seconds to 30 seconds has a drastic power-saving effect.

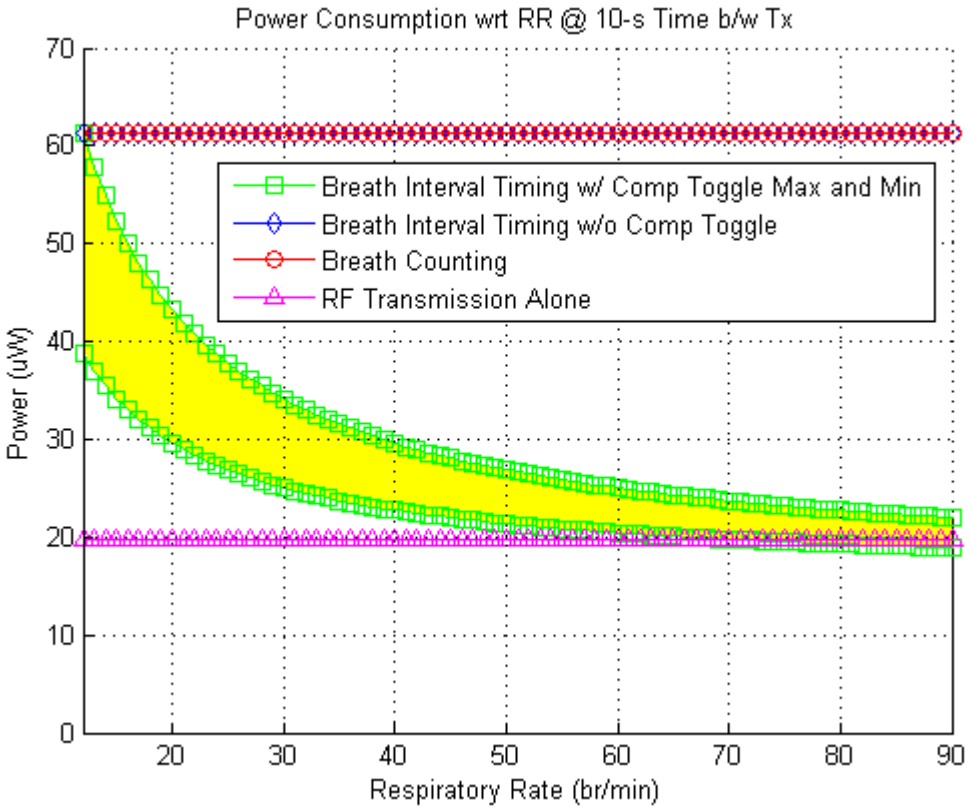



**Figure 4.20: Combined Power Estimation Plot in Relation to Respiratory Rate with Transmission Off and Transmission Set at 10 s.**

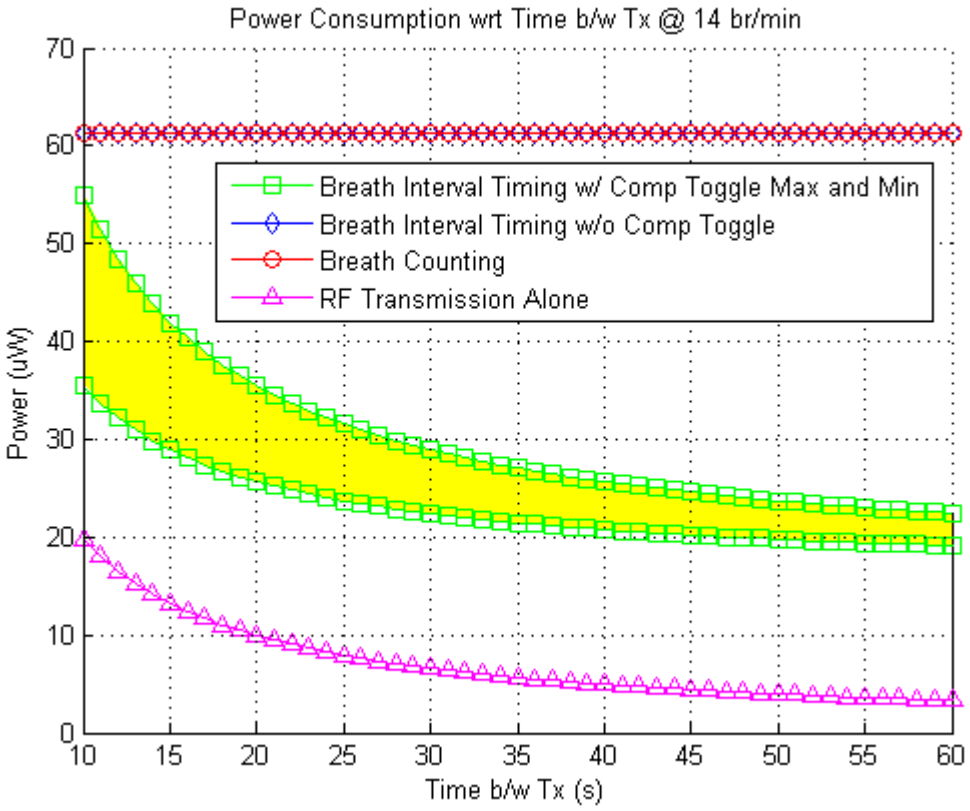



**Figure 4.21: Combined Power Estimation Plot in Relation to Transmission Interval with Transmission Off and a Respiratory Rate of 14 br/min.**

All power plots and estimations in this chapter were generated by a simple MATLAB program. Instead of integrating over each period to find the average voltage over $V_{shunt}$, an approximation approach was used by estimating the area under the voltage curve with basic geometric shapes. Figure 4.22 provides a simplified illustration of this technique. The area of each feature is averaged over the whole period, and each average is then summed to obtain the overall voltage average. As mentioned previously, this overall average $V_{shunt}$ is then used to calculate the average current, which, in turn, is multiplied by the voltage over the CC430 to provide the average power estimation of the subsystem. The equation summary for these calculations is in Figure 4.23.

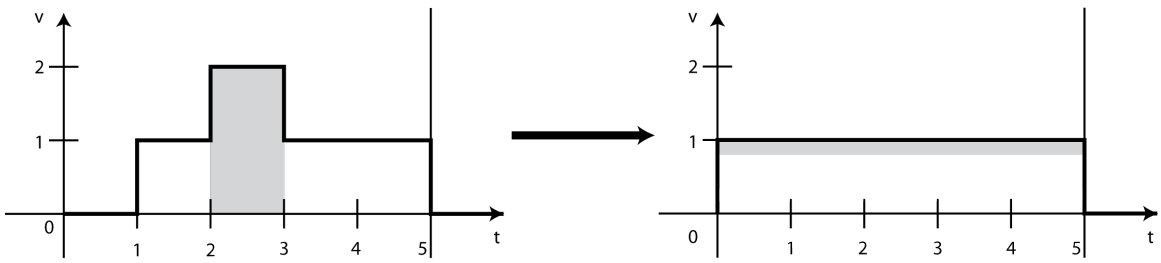



**Figure 4.22: Average $V_{shunt}$ Approximation Technique.**

$$\overline{I_{shunt}} = \frac{\overline{V_{shunt}}}{R_{shunt}}$$

$$\overline{I_{CC430}} = \overline{I_{shunt}}$$

$$\overline{P_{CC430}} = \overline{V_{CC430}} \cdot \overline{I_{CC430}}$$

$$\overline{P_{CC430}} = (Vcc - \overline{V_{shunt}}) \cdot \overline{I_{CC430}}$$

**Figure 4.23: CC430 Power Consumption Equations.**

The power estimation program along with all the figures and approximation explanations of the voltage measurements captured in this thesis are available in Appendix C. This program was also used to generate Tables 4.1 and 4.2, which provide concrete power estimations with regards to specific situations of interest. Table 4.1 discloses power estimations of the respiratory rates of 6, 15, 30, and 90 br/min while holding the transmission interval constant at 10 seconds. These RR were chosen to be comprehensive by providing information for an abnormally low RR, a normal adult RR, a child RR, and an abnormally high RR, respectively. Table 4.2 revels power estimations based on a change in transmission interval where RR is fixed to 14 br/min. The transmission intervals chosen were 10, 15, 30 and 60 seconds (i.e., 1 minute). A 10-s interval was chosen since 10 seconds of no breathing is defined as apnea; the 15-s and 30-s intervals were chosen to represent the general time breakdowns used by a physician during a checkup; and, the 60-s interval was used for baseline accuracy of the actual respiratory rate.

**Table 4.1: Average Power Consumption Comparisons at 10-s Transmission.**

| **Algorithm/Condition** | **Radio (on/off)** | **Power Consumption (μW)** | | | |
|---|---|---|---|---|---|
| | | 6 br/min | 15 br/min | 30 br/min | 90 br/min |
| RF transmission only | - | 19.84468 | 19.84468 | 19.84468 | 19.84468 |
| Breath counting | off | 61.35420 | 61.35424 | 61.35427 | 61.35442 |
| Breath interval timing w/ comparator constantly on | | * | 61.35458 | 61.35458 | 61.35458 |
| Breath interval timing w/ comparator toggled (max power) | | * | 52.31791 | 34.15232 | 21.97396 |
| Breath interval timing w/ comparator toggled (min power) | | * | 34.15232 | 25.02365 | 18.92088 |
| Breath counting | on | 81.19888 | 81.19891 | 81.19895 | 81.19910 |
| Breath interval timing w/ comparator constantly on | | * | 81.19926 | 81.19926 | 81.19926 |
| Breath interval timing w/ comparator toggled (max power) | | * | 72.16259 | 53.99700 | 41.81864 |
| Breath interval timing w/ comparator toggled (min power) | | * | 53.99700 | 44.86833 | 38.76556 |
| 12-bit ADC w/o logic (i.e., no rate extraction nor RF transmission) using a 5-MHz clock at 200 ksps and 3.0 V [**45**] | - | 660 | 660 | 660 | 660 |

* Invalid with current implementation.

**Table 4.2: Average Power Consumption Comparisons at 14 br/min.**

| Algorithm/Condition | Radio (on/off) | Power Consumption (μW) | | | |
|---|---|---|---|---|---|
| | | 10-s Transmission | 15-s Transmission | 30-s Transmission | 1-min Transmission |
| RF transmission only | - | 19.84468 | 13.22986 | 6.61496 | 3.30749 |
| Breath counting | off | 61.35424 | 61.35420 | 61.35416 | 61.35415 |
| Breath interval timing w/ comparator constantly on | | 61.35458 | 61.35442 | 61.35426 | 61.35417 |
| Breath interval timing w/ comparator toggled (max power) | | 54.90301 | 41.95230 | 28.93918 | 22.40921 |
| Breath interval timing w/ comparator toggled (min power) | | 35.45392 | 28.93943 | 22.40933 | 19.13844 |
| Breath counting | on | 81.19891 | 74.58405 | 67.96912 | 64.66163 |
| Breath interval timing w/ comparator constantly on | | 81.19926 | 74.58428 | 67.96922 | 64.66166 |
| Breath interval timing w/ comparator toggled (max power) | | 74.74770 | 55.18216 | 35.55414 | 25.71670 |
| Breath interval timing w/ comparator toggled (min power) | | 55.29860 | 42.16929 | 29.024299 | 22.44593 |
| 12-bit ADC w/o logic (i.e., no rate extraction nor RF transmission) using a 5-MHz clock at 200 ksps and 3.0 V [**45**] | - | 660 | 660 | 660 | 660 |

An observation from Tables 4.1 and 4.2 is that the estimated power consumption of just turning on the ADC module is hundreds of microwatts more than the breath counting breath interval timing algorithms with RF transmission turned on. What is interesting is that as the time between RF transmission increases, the RF dominance becomes less important, and more emphasis is placed on the algorithm implementation. Table 4.2 suggests that an ADC implementation would be capped at 660 μW just to run the ADC module, while the breath counting and breath interval timing algorithms with RF transmission enabled would decrease until about the 19 – 62 μW range when the algorithms are not using RF transmission. This indicates that even with the resistor ladder enabled, the comparator implementation utilizes less power than the ADC.

All in all, the power consumption values in Tables 4.1 and 4.2 for this thesis's implementations with and without RF transmission are definitively below the 100-μW

requirement of this study's subsystem. Thus, it has been demonstrated quantitatively that it is feasible to utilize the CC430 to extract the rate from a respiratory signal and transmit it via RF technology given a low source of power, such as that created by the respiratory effort itself.

# 5. Conclusion and Future Work

Throughout this thesis, several situations have directed interested individuals towards further research topics. In this chapter, these topics have been gathered to form the subchapter beyond the conclusion subchapter for future reference.

## 5.1. Conclusion

There are many studies that have cited utilizing a low-power SoC or DSP to prolong the issue of battery replacement or recharging for portable devices. However, they rarely expound beyond this mere reference and do not provide any further information as to how much power is saved and how much longer the batteries are expected to last. This leaves the general feasibility question slightly open; can a low-power SoC or DSP truly save that much power during actual application? If so, how much? If the battery replacement occurs every 5 – 6 days rather than every 3 – 4 days, do those few extra days really count as an improvement?

The overall project of this study does not just propose postponing the inevitable battery-change of a wireless respiratory monitoring device; it completely removes this mundane and tedious task from the monitoring routine. It suggests that the respiratory effort itself has the potential to provide enough power to support its monitoring device. This supposedly can be achieved by using a low-power SoC with RF capabilities to extract the respiratory rate from the signal and transmit it to a base station via RF technology. However, respiratory effort is not known to be a very strong, reliable source of power; in fact, utilizing a passive displacement sensor, it has been preliminarily tested to create about 100 μW to 3.3 mW. Thus, one of the major topics of this thesis was to determine if it was feasible that an SoC responsible for monitoring respiratory rate could be sustained and function properly with less than 100 μW of power. Stemming from this, the main goal of this thesis was to develop an SoC implementation for a wireless rate monitoring system utilizing the minimum amount of power.

The responsibility of the SoC can be broken down into two parts: the part that extracts the rate from the respiratory signal and the part that transmits the rate via RF technology. Since the actually RF requirements and hardware have not be settled in the overall project, the RF power consumption contribution was not considered the focus of

this thesis. However, the rate extraction logic is a controllable variable, which needs to be examined for its power efficiency. There are several algorithms that can be used to extract the rate from a respiratory signal; however, not all of them consider low-power consumption as a high priority and are quite complex. This thesis proposed some simplistic algorithms to test for power efficiency, since they were more likely to yield power consumption on the microwatt-level than more intensive algorithms. The overall project goal of this thesis was to prove feasibility of a respiratory-rate-monitoring algorithm in an incredibly low-power environment no matter how pedestrian the algorithm might seem. If these incredibly simple algorithms cause a relatively vast amount of power consumption, then feasibility under the 100-μW constraint might be hypothesized as impossible, as least for the specific SoC chosen for this thesis.

The SoC assessed for implementation feasibility was the CC430 developed by TI, and its abilities were evaluated using an EM430F6137RF900 board. Of the five algorithms proposed, only two, breath counting and breath interval timing, were considered as worthy to be studied further based on their restrained use of RF transmission through data compression and infrequent transmissions. Implementation of each algorithm was scrutinized for low power consumption, although there remains room for further study into mentioned variations. LPM was utilized whenever possible, and CPU time was used sparingly, leading to very encouraging power consumption results. Specifically, this thesis has demonstrated that a simple comparator can be used for low-power rate determination rather than an ADC.

Measurements and simulation results indicate that the power consumptions of both implemented algorithms with the comparator are definitely below 100 μW, with and without RF transmission enabled. The obtained 55 – 82 μW of total power consumption using the comparator method was considered a vast improvement over the usual ADC method, which would have utilized more than 660 μW with the same implementation. Of course, the RF transmission might be underestimated because the target environment might require more robust communication; however, this thesis has shown that it is functionally possible for both RR extraction and RF communication to utilize less than 100 μW and provides isolated power estimates of the algorithms for future studies that might concentrate only on optimizing the RF portion.

In conclusion, this thesis has shown that is it feasible to capture the respiratory rate with the CC430 utilizing the atypical analog interface of a comparator and less than 100 μW of power. It has, thus, theoretically proven that the digital processing unit of a respiratory monitoring device can be powered via the respiratory effort itself. It has also provided a general ultra-low-power SoC implementation for any rate monitoring application with a similar waveform.

## 5.2. Future Work

There is much work that can still be accomplished beyond this thesis to better understand the power requirements of a respiratory-rate-extracting SoC. The following subchapters cover a few highlights and note some additional new ideas.

### 5.2.1. Comparator Sampling

Since it was shown in Chapter 4 that the average operating power of the comparator algorithms is less than the operating power of the ADC module, an interesting new study would be to sample the comparator as would be done with an ADC. This method is expected to require even less power. A second timer module may be required; however, it can be suggested that this additional module will require less power than would be consumed by the comparator over an equivalent time period. This conjecture is made based on the observation that toggling the comparator in the breath interval timing algorithm creates a drastic drop in relative power consumption despite other units being on.

### 5.2.2. Real Respiratory Data

This thesis based functional testing on an ideal sinusoidal respiratory input; however, it is very important to test the algorithm given real respiratory input from a sensor through an ASP circuit. If the hardware is not fully developed for this portion of the overall project, functional validation may be done through utilizing a case scenario of preliminarily measured respiratory data and an arbitrary waveform generator.

### 5.2.3. ADC Power Measurement Verification

Although the operating power of the ADC is cited from the datasheet for comparison in Chapter 4, this value should be validated through equivalent implementation of all algorithms using the ADC. This will ensure similar testing conditions.

### 5.2.4. Alternate Signal Routing and Components

Specifically for the breath counting algorithm, it may be possible to replace the CPU code for counting with an actual off-chip counter or with routing the comparator output to a second SoC timer's clocking input. These methods were not pursued not only because of the need for a second counter/timer, but also because of the uncertainty of how the SoC will handle a simultaneous transmit and count. In the current implementation, these two actions are based on CPU interrupts that have a hardwired priority. However, using a manual counter/timer interface may result in a collision when the counter/timer is incrementing and a transmission must occur. The CPU would need to manually stop the counter/timer before reading it to obtain an uncorrupted reading; however, this might resolve in missing a count if the counter/timer is stopped just before the start of a new breath. Nonetheless, if this issue can be resolved reliably, it is a viable alternative low-power SoC implementation candidate.

### 5.2.5. CC340 Software and Hardware Variations

As mentioned in Chapter 3, there are several possible software and hardware variations that may yield different, hopefully better, power results. Future studies may want to pursue the implementations of these variations to find the most optimized version of this thesis's or other algorithms. However, these variations are only possible because of the design of the CC430, so they may not apply to other SoCs, which might have more or less variations.

### 5.2.6. Broader Algorithm Diversity

The algorithms in this study are very simple, especially when it comes to extracting the rate from the fixed ideal sinusoidal input. For instance, if the breathing pattern or tidal volume for the person is odd, there is no way of knowing this, since only

the rate is displayed. Also, if the patient talks or moves, the rate calculations do not compensate for this noise. More robust algorithms should be developed and tested for power consumption; since, from a medical perspective, the more information known, the better it is for the patient.

Also, at the time of this thesis, the post-calculation algorithms, addressed in Chapter 2, were ruled out of testing because it is known that constant RF transmissions would consume a great amount of power relative to the other operations required of the SoC. However, in the future, affordable, improved, extremely low-power RF technology may become available. This should cause a re-inspection of the possibility of using the post-calculation algorithms, since the CPU can be completely bypassed using the DMA. An immediate functional benefit would be the ability to calculate tidal volume and look for patterns and noise.

### 5.2.7. Optimization of RF Settings and Antenna Hardware

As stated in Chapter 3, the specifics of the RF settings and hardware were not considered to be in the scope of this thesis. Nonetheless, if true insight as to how much power the SoC really requires in total must be gained, the strength and frequency of the RF signal, as well as the transmission rate would need to be known. Such information dictates the proper RF settings required and will most likely influence the overall system's hardware design; an expected change would be the antenna circuit, of which the EM430F6137RF900's is fairly large. In fact, TI has created a product called the eZ430-Chronos watch [**70**], which utilizes the same SoC as this study. The mere existence of this device demonstrates the feasibility of creating a more compact antenna circuit. If future studies use this thesis's code as a basis for their programs, it is highly recommended that a thorough low-level optimization exploration of the RF portion of the C-code be conducted, since the RF libraries used in this study are from TI sample code and do not guarantee any sort of power, memory, or run-time optimization.

### 5.2.8. Safety, Reliability, and Security Implementations

Certain medical environments may require more safety, reliability, and security to be programmed into the end measurement device. Some considerations include having the ability to detect apnea and produce an error to set off an alarm, utilizing redundancy

checking and bidirectional communication, and encrypting data transmissions. There are many improvements that can be made; however, research much be conducted to determine if the power cost of implementing these features is outweighed by the feature necessity and benefit.

### 5.2.9. Regulation and Standard Compliance

The future end product of the overall project must be compliant with all regulations and standards, especially those pertaining to medical safety and signal interference. Although such compliance has been left for later study, it must be established and tested, since these changes will most likely alter the power consumption caused by the RF portion. This thesis's ability to isolate the algorithmic power consumption and the transmission power consumption allows the RF research to be done independently and also simultaneously, if need be.

### 5.2.10. More Robust Receiver

The current receiver code is very simplistic and is un-optimized for minimal power consumption. Future studies might want to increase the capabilities of the receiving module and also minimize its power consumption. They may also desire to add this module to a larger, more complex system that can interpret and summarize data.

### 5.2.11. New SoC

The choice of the best SoC was not thoroughly investigated with relation to cost and direct feature comparisons. Future researchers may want to revisit the integrated circuit (IC) market when designing their system.

# Appendix A: CC430 C-Code

This appendix contains the C-code for the transmitting and receiving SoCs used in this thesis. As further detailed in Appendix D, there are two iterations of the code for the receiving module, both of which are provided below for completeness and future debugging purposes. Also provided below are references to the TI samples utilized in this thesis; both code for preliminary testing and code required for the execution of the transmitter/receiver code are described.

## A.1. Transmitter Module

In the transmitter module code provided below, there are several marcos near the beginning of the file and after the comments. Most of them are provided to help during debugging; however, there are a few marcos of particular interest. For the most part, all macros involving an LED or output to a pin can be left undefined so that the compiler ignores anything unnecessary and efficiently complies and links the rest of the program; however, the macros that alter transmission interval and settings, projection time, or comparator registers are used to test the power efficiency of the SoC in particular configurations. Details of what each macro does are in the comments of the code.

### A.1.1. Breath Counting Algorithm: tx1dot3.c

```
//*****************************************************************************
// VERSION: tx1.3.c
// CREATED: 10/29/2011
// REVISED: 10/29/2011
// RX PAIR: rx0.2.c
// REQUIRED: cc430f6137.h, hal_pmm.h/c, RF1A.h/c, RfRegSettings.c
//
// COMMENTS:
// Respiratory rate is measured in br/min. We can simulate the human method of
// obtaining this rate by counting the number of breaths in 1 min. We can also
// count the number of breaths in 10 sec, 15 sec, and 30 sec, and multiply these
// by 6, 4, and 2 respectively in order to calculate br/min.
//
// To do so in software, we set up a timer that interrupts and resets after the
// duration of the timer interval is up (whether we choose 10 sec, 15 sec,
// 30 sec, or 1 min). This signals the SoC to transmit a variable called
// ucRespRate over RF. We are going to use TA1, which has an interrupt priority
// lower than the radio. Since the timer is only 16 bits, it can only count up
// to 2^16 - 1 = 65535. Let us use the 32.768 kHz internal clock provided by
// the reference oscillator REFO so that we don't have to solder on the
// 32.768 kHz crystal onto XT1 pins. We'll use it as our master clock and
// also our timer TA1 base. Let us use the auxiliary clock ACLK to host TA1
// and turn off the sub-master clock SMCLK. ACLK will be sourced by the
// REFO, but will have a divisor of 32, which creates ACLK = 1024 Hz.
// (See pg.17 of the datasheet for interrupt priority details.)
//
```

```
// RF settings are set based off of the RF_Toggle_LED_Demo using the 915 MHz
// settings and a fixed packet length of 1 (byte), which fits in the FIFO.
// 915 MHz was chosen since it required less power than 868 MHz according
// to the datasheet. 868/915 MHz are the only available options for
// evaluation board EM430F6137RF900.
//
// This program will not have the ability to receive anything via RF; it will
// only be able to transmit data.
//
// We are using a sinusoidal waveform to simulate a person's breathing pattern
// and connect it to the comparator pin. Using particular settings, the
// comparator is able to interrupt for every rising edge of the sinsoidal
// signal. This interrupt is what allows us to increment ucRespRate. This
// "counting" of breaths that is kept in the variable ucRespRate is what is used
// to calculate the respiratory rate. For instance, if we want to transmit every
// 15 sec, we can count for the duration of 15 sec and multiply the count by 4
// in order to get a projection of the respiratory rate in br/min.
//
// Noise on the comparator output is cancelled out by using an adaptive
// comparator reference voltage. The signal does not change from low to high
// unless the input gets higher than Vcc*3/4 and does not change from high to
// low unless the input gets lower than Vcc*1/4. This leaves a (3/4-1/4)*Vcc =
// Vcc/2 gap for noise.
//
// For most of the time, the radio core and the CPU will be in sleep and
// low-power mode, respectively, until a transmission/calculation is required.
// When the timer interrupts to indicate that a transmission is required, the
// ISR will set the variable ucTransmitRequest to indicate to main() that it
// needs to initiate a transmission sequence. The interrupt exits to main()
// to proceed with the code for transmitting ucRespRate. Then, main() wakes the
// radio core up for transmission and puts the CPU back into low-power mode
// until the transmission is complete. ucRespRate is reset after completion.
//
// For even more power savings, the high and low side supply voltage supervisors
// and monitors were turned off.
//
//*****************************************************************************

//*****************************************************************************
// Debug code settings (Set these carefully!)
//*****************************************************************************
//#define PUSH_BUTTON               // Push button P1.7 interrupt forces tx
//#define RED_TRANSMIT_LED          // Red LED (RED) is on during transmission
//#define NONZERO_INITIAL_RESP_RATE // Sets the initial respiratory rate to 35
//#define TEST_ACLK_ON_P1_0         // Toggles P1.0, green LED (GREEN),
                                    // @ every TA1 CCR0 interrupt
//#define TEST_COMP_INT_ON_P1_1     // Toggles P1.1 for each comp. interrupt

// Define the functionality of the push button
#ifdef PUSH_BUTTON

    #define RESP_BUTTON             // Button increments ucRespRate
    //#define TRANSMIT_BUTTON           // Button requests a transmission

    // Check if we've chosen a button config
    #if (!defined(RESP_BUTTON) && !defined(TRANSMIT_BUTTON))
    #error "Please select either RESP_BUTTON or TRANSMIT_BUTTON \
    as the active push button configuration"
    #endif

    // Check if we've erroneously chose both config
    #if (defined(RESP_BUTTON) && defined(TRANSMIT_BUTTON))
    #error "Please select either RESP_BUTTON or TRANSMIT_BUTTON \
    as the active push button configuration"
    #endif

#endif // PUSH_BUTTON

//*****************************************************************************
// Power measurement variation settings (Set these carefully!)
//*****************************************************************************
```

```
#define RATE_PROJECTION         // We can select among 10 s, 15 s, 30 s, or
                                // 1 min transmit intervals
#define TRANSMIT_OFF            // Turn off transmission to measure power for
                                // computational features

// Define the different settings for selecting the transmit interval to use
#ifdef RATE_PROJECTION

    // Select an interval
    //#define PROJECTION_10_SEC     // Select transmission to occur every 10 sec
    #define PROJECTION_15_SEC       // Select transmission to occur every 15 sec
    //#define PROJECTION_30_SEC     // Select transmission to occur every 30 sec
    //#define PROJECTION_1_MIN      // Select transmission to occur every 1 min

    // Select if projection multiplication is done on the transmitting
    // CC430 (this chip); or the receiving CC430 (not this chip)
    #define TX_CALC_PROJECTION
    //#define RX_CALC_PROJECTION

    // Check if we've chosen a projection config interval
    #if (!defined(PROJECTION_10_SEC) \
    && !defined(PROJECTION_15_SEC) \
    && !defined(PROJECTION_30_SEC) \
    && !defined(PROJECTION_1_MIN))
    #error "Please select either \
    PROJECTION_10_SEC, \
    PROJECTION_15_SEC, \
    PROJECTION_30_SEC, \
    or PROJECTION_1_MIN as the active projection configuration"
    #endif

    // Check if we've erroneously chose more than one config interval
    #if ((defined(PROJECTION_10_SEC) \
    + defined(PROJECTION_15_SEC) \
    + defined(PROJECTION_30_SEC) \
    + defined(PROJECTION_1_MIN)) > 1)
    #error "Please select either \
    PROJECTION_10_SEC, \
    PROJECTION_15_SEC, \
    PROJECTION_30_SEC, \
    or PROJECTION_1_MIN as the active projection configuration"
    #endif

    // Check if we've chosen a projection config calculation
    #if (!defined(TX_CALC_PROJECTION) \
    && !defined(RX_CALC_PROJECTION))
    #error "Please select either TX_CALC_PROJECTION \
    or RX_CALC_PROJECTION as the active projection configuration"
    #endif

    // Check if we've erroneously chose more than one config calculation
    #if ((defined(TX_CALC_PROJECTION) \
    + defined(RX_CALC_PROJECTION)) > 1)
    #error "Please select either TX_CALC_PROJECTION \
    or RX_CALC_PROJECTION as the active projection configuration"
    #endif


#endif // RATE_PROJECTION

//*****************************************************************************
// Header files
//*****************************************************************************
#include "cc430f6137.h"
#include "hal_pmm.h"
#include "RF1A.h"

//*****************************************************************************
// Macros
//*****************************************************************************
#define  PATABLE_VAL        (0x8D)      // 0 dBm output
```

```
#define  LPM_BITS           (LPM4_bits) // Select low-power mode 4

// Define timer tick based on transmit interval
//
// t = 10s = transmit interval we want in seconds
// c = 32.768kHz = CPU clock frequency
// d = 32 = ACLK frequency divisor
// c/d is the ACLK frequency that TA1 runs off of
// t(c/d) = timer clocks required for transmission
//
// Sample calculation...
// 10s ((32.768k clks/s) / 32) = 10,240 (tmr clks)/tick
//
#ifdef PROJECTION_10_SEC
    #define  TMR_CLKS_PER_TA1_TICK  (10239) // (timer clk cycles per tick - 1)
#elif defined(PROJECTION_15_SEC)
    #define  TMR_CLKS_PER_TA1_TICK  (15359) // (timer clk cycles per tick - 1)
#elif defined(PROJECTION_30_SEC)
    #define  TMR_CLKS_PER_TA1_TICK  (30719) // (timer clk cycles per tick - 1)
#else   // default is 1 minute tick interval w/o projection
    #define  TMR_CLKS_PER_TA1_TICK  (61439) // (timer clk cycles per tick - 1)
#endif

//*****************************************************************************
// Global variables
//*****************************************************************************
extern RF_SETTINGS rfSettings;      // See RfRegSettings.c
unsigned char ucRespRate;           // breaths per min
unsigned char ucTransmitRequest;    // Set to cause resp rate tx in main()

//*****************************************************************************
// Prototypes
//*****************************************************************************

//*****************************************************************************
// The main program
//*****************************************************************************
void main(void)
{
    // Create local variable to track CBOUT (the comparator output)
    // during a transmit; This is used to avoid missing a breath
    // between a transmit sequence and the reset of ucRespRate.
    // A breath occurs during a transition from 0 to 1, so the
    // default value here must be 1 to avoid any collision of the
    // initial condition with the condition to update (1 to 1 does
    // not indicate a breath was taken).
    unsigned char ucCBOUTBeforeTransmit=1;

     // Stop watchdog timer to prevent timeout reset
    WDTCTL = WDTPW + WDTHOLD;

    // Set master clock as soon as possible to save power immediately
    // CANNOT: Set master CPU clock source to be same clock as RF module
    //         (26 MHz RF crystal) since it is higher than the max freq
    //         of the chip (20 MHz)
    // CANNOT: Set master CPU clock source to be 32.678 kHz crystal, because
    //         it's not installed yet
    // THUS:   Set MCLK source as the 32.768 kHz REFO, same for ACLK and SMCLK
    UCSCTL4 = SELA__REFOCLK + SELS__REFOCLK + SELM__REFOCLK;

    // ACLK on external pin divisor = 1/1
    // ACLK divisor = 1/32
    // SMCLK divisor = 1/1
    // MCLK divisor = 1/1
    UCSCTL5 = DIVA__32;

    // (Turn off XT2 crystal oscillator when the radio core is asleep)
    // (Drive for XT1 is at max for 24-32MHz crystals)
    // (Low-freq mode is chosen for XT1, caps must be defined)
    // (XT1 is source internally rather than from a pin)
    // (Default oscillator capacitors are chosen)
```

```
// Turn off subsystem master clock since we are not using it
// (XT1 is off when not used as a source for clocks or FLL)
UCSCTL6 |= SMCLKOFF;

// Initialize global variables
ucRespRate=0;           // Initial resp rate is 0 br/min
ucTransmitRequest=0;    // No transmission of resp request at start

#ifdef NONZERO_INITIAL_RESP_RATE
ucRespRate=35;
#endif

// Increase PMMCOREV level to 2 for radio operation
// (Vcore change must be gradual)
SetVCore(2);

// Configure radio interface registers

// Set radio core to interrupt at the negative edge
// (for end-of-packet reading on RFIFG9)
RF1AIES |= BIT9;

// Set Power Management Module and Voltage Supply Supervisor (SVS)
// and Monitor (SVM)
PMMCTL0_H = 0xA5;
SVSMHCTL_H &= ~(SVSHE_H + SVMHE_H); // Disable high-side SVS and SVM
SVSMLCTL_H &= ~(SVSLE_H + SVMLE_H); // Disable low-side SVS and SVM
PMMCTL0_L |= PMMHPMRE_L;            // Allow for peripherals to request
                                    // high-power when in LPM
PMMCTL0_H = 0x00;

// Configure radio core through commands
Strobe(RF_SRES);                    // Reset radio core and put it to sleep
Strobe(RF_SNOP);                    // Reset radio pointer
WriteRfSettings(&rfSettings);       // Adjust to proper radio settings
WriteSinglePATable(PATABLE_VAL);    // Write to the power amplifier table

// Set default state of ports as outputs and put them to zero
// to avoid parasitic current consumption due to avoid floating inputs
PAOUT = 0x00;   // Ports 1 and 2
PADIR = 0xFF;
PBOUT = 0x00;   // Ports 3 and 4
PBDIR = 0xFF;
P5OUT = 0x00;   // Port 5
P5DIR = 0xFF;
PJOUT = 0x00;   // Port J (required because of JTAG debugger connection)
PJDIR = 0xFF;

#ifdef TEST_ACLK_ON_P1_0
// Set up P1.0 as an output for GREEN to toggle each
// time that TA1 runs out (counts to CCR0)
//P1SEL &= ~BIT0;       // Select I/O functionality for P1.0 (default)
//P1DS &= ~BIT0;        // Select reduced drive strength for P1.0 (default)
//P1REN &= ~BIT0;       // Pull-up/down resistor is a don't care on output
//P1IES &= BIT0;        // Low-to-high transition is a don't care on output
//P1IFG = 0;            // Clear the Port 1 vector interrupt flag
//P1IE &= ~BIT0;        // Interrupt enable is a don't care on output
P1OUT &= ~BIT0;         // Initialize GREEN as off
P1DIR |= BIT0;          // Set P1.0 as an output
#endif // TEST_ACLK_ON_P1_0

#ifdef RED_TRANSMIT_LED
// Set up P3.6 as an output for RED to indicate
// a transmission is in progress
//P3SEL &= ~BIT6;       // Select I/O functionality for P3.6 (default)
//P3DS &= ~BIT6;        // Select reduced drive strength for P3.6 (default)
P3OUT &= ~BIT6;         // Initialize RED as off
P3DIR |= BIT6;          // Set P3.6 as an output
#endif // RED_TRANSMIT_LED

#ifdef PUSH_BUTTON
```

```
// Set up P1.7 as a interrupt so that we may use
// the push bottom on it as a transmit message trigger
//P1SEL &= ~BIT7;       // Select I/O functionality for P1.7 (default)
//P1DS &= ~BIT7;        // Select reduced drive strength for P1.7 (default)
//P1DIR &= ~BIT7;       // Set P1.7 as an input (default)
P1REN |= BIT7;          // Set P1.7 REN & OUT to enable the pull-up resistor
P1OUT |= BIT7;
P1IES &= BIT7;          // Set that a low-to-high transition will set the
                        // P1.7 interrupt flag
P1IFG = 0;              // Clear the Port 1 vector interrupt flag
P1IE |= BIT7;           // Enable P1.7 as an interrupt
#endif  // PUSH_BUTTON

// Set up timer TA1 to interrupt every minute

// Set TA1CCR0 value to have a 1 sec tick
// We want TMR_CLKS_PER_TA1_TICK + 1 counts and the number of timer counts
// in a period is (TA1CCR0 + 1), thus we set it to TMR_CLKS_PER_TA1_TICK
TA1CCR0 = TMR_CLKS_PER_TA1_TICK;

// (Keep no capture on)
// (Keep default capture/compare input)
// (Keep default async capture source)
// (Stay in default compare mode)
// (Keep default output mode)
// Enable CCR0 interrupt
// (Keep default output)
// (Clear capture overflow indicator)
// (Clear interrupt flag)
TA1CCTL0 |= CCIE;

// LOGIC: Using TA1 since it has less capture/compare registers
// Select Timer A1 clock as ACLK (starts timer)
// Keep TA1 divisor as 1
// Set TA1 to be in count up mode (count to value in TA1CCR0)
// Clear TA1
// Disable TAIFG interrupts for CCR1-CCR2
// Clear the interrupt flag at onset
TA1CTL = TASSEL__ACLK + ID__1 + MC__UP + TACLR;

// Set up the comparator for adaptive VREF to achieve hysteresis

// Disable input channel that allows external input for the - terminal
// We don't need to choose a channel for the - terminal since it's disabled
// Enable input channel that allows external input for + terminal
// Select channel CB0 for the + terminal
CBCTL0 = CBIPEN + CBIPSEL_0;

// Comparator reference accuracy is set to static mode.
// Reference voltage is disabled
// Resistor tap for VREF1 (CBOUT = 1) is set to Vcc*1/4
//      Ref p.69 of datasheet for CBREF1
//      Vcc*1/4 = Vcc*(n+1)/32, n = 7
// Comparator reference source is Vcc applied to the resistance ladder
// Reference voltage produced is connected to the negative terminal of comp.
// Resistor tap for VREF0 (CBOUT = 0) is set to Vcc*3/4
//      Ref p.69 of datasheet for CBREF0
//      Vcc*3/4 = Vcc*(n+1)/32, n = 23
CBCTL2 = CBREF1_7 + CBRS_1 + CBRSEL + CBREF0_23;

// Disable input buffer @ P2.0/CB0
CBCTL3 = CBPD0;

// Comparator output selects b/w VREF1 and VREF0 as comparator reference
// CBMRVS is a manual select for the comparator reference, which we don't
// care about.
// Turn on the comparator
// Choose ultra-low-power mode
// Select lowest filter delay of 450 ns
// Do not exchange the comparator 0 inputs
// Do not short the +/- comparator inputs
```

```
// Select the rising edge as the interrupt trigger
// Enable the comparator filter
// Do not invert CBOUT
// CBOUT is read-only so we don't care about this bit
CBCTL1 = CBON + CBPWRMD_2 + CBF;

// Disable inverted polarity interrupt
// Enable non-inverted polarity interrupt
// Clear inverted polarity interrupt flag
// Clear non-inverted polarity interrupt flag
CBINT = CBIE;

// The bulk the program
while(1)
{
    // Enter low-power mode and enable interrupts
    __bis_SR_register(LPM_BITS + GIE);

    // After an interrupt wakes us up from a low-power mode
    // we need to check if we should be transmitting a packet
    if(ucTransmitRequest)
    {
        // Since we've had a request for transmission made,
        // we can begin transmission

        #ifdef RED_TRANSMIT_LED
        // Turn on RED at beginning of transmit
        P3OUT |= BIT6;
        #endif // RED_TRANSMIT_LED


        // Projection multiplication can possibly be postponed until
        // after transmit and done by the receiving CC430 in order to save
        // computation time. Check which setting we have.
        #ifdef RX_CALC_PROJECTION
            // No multiplication scaling must occur
        #else // TX_CALC_PROJECTION
            // Calculate the respiratory rate to send for the
            // projection that we are using.
            #ifdef PROJECTION_10_SEC
                ucRespRate*=6;   // 6 * 10 sec = 1 min
            #elif defined(PROJECTION_15_SEC)
                ucRespRate<<=2;  // 4 * 15 sec = 1 min
            #elif defined(PROJECTION_30_SEC)
                ucRespRate<<=1;  // 2 * 30 sec = 1 min
            #else   // default is 1 minute tick interval w/o projection
                // ucRespRate*=1; 1 * 1 min = 1 min
            #endif
        #endif // RX_CALC_PROJECTION

        // Clear RFIFG9 flag for the end-of-packet interrupt
        RF1AIFG &= ~BIT9;

        // Take note of the output of the comparator before transmission
        ucCBOUTBeforeTransmit=CBOUT;

        // Check if we turned transmission on or off to determine
        // if the following code should be compiled.
        #ifdef TRANSMIT_OFF
            // No transmission should occur

            // Clear the request for transmission since we've just
            // finished transmitting
            ucTransmitRequest=0;
        #else
            // Enable TX end-of-packet interrupt
            RF1AIE |= BIT9;

            // Write the rate to the TX FIFO
            WriteSingleReg(RF_TXFIFOWR,ucRespRate);
        #endif // TRANSMIT_OFF
```

```
            // Reset the respiratory rate to beginning counting again
            ucRespRate=0;

            // Check if a breath occured during transmission and account
            // for it. Suppose that the fastest that breathing can occur
            // is 10 Hz and we are using a 32.768kHz CPU clock, then
            // we have the following calculation:
            //
            // (1 s / 10 br)(32.768k CPU instr / s) = 3276.8 instr/br
            //
            // This means that we can execute at least 3276 CPU instructions
            // between the resetting of ucRespRate and it's transmission
            // in order to properly catch an single breath that might occur
            // during this time that the ucRespRate variable can be compromised.
            if(!ucCBOUTBeforeTransmit && CBOUT)
                ++ucRespRate;   // A rise from 0 -> 1 occurred & must be tracked

            #ifdef NONZERO_INITIAL_RESP_RATE
            ucRespRate=35;
            #endif

            // Check if we turned transmission on or off to determine
            // if the following code should be compiled.
            #ifdef TRANSMIT_OFF
                // No transmission should occur
            #else
                // Wake up the radio core and enable TX processing
                Strobe(RF_STX);
            #endif // TRANSMIT_OFF

        } // End of check for transmit request

        #if defined(PUSH_BUTTON) && defined(RESP_BUTTON)
        // Re-enable the push button, enabling P1.7 interrupt
        // (done here for debouncing)
        P1IE |= BIT7;
        #endif // PUSH_BUTTON && RESP_BUTTON

    } // End of the while(1) of main()
} // End of main()

//*****************************************************************************
// Interrupt routines in order of priority from highest to lowest
//*****************************************************************************

// ISR for the comparator
#pragma vector=COMP_B_VECTOR
__interrupt void COMP_B_ISR(void)
{
    #ifdef TEST_COMP_INT_ON_P1_1
    P1OUT ^= BIT1;          // Toggle P1.1
    #endif // TEST_COMP_INT_ON_P1_1

    ++ucRespRate;           // Update respiratory rate
    CBINT &= ~CBIFG;        // Clear interrupt flag for non-inverted polarity

    // Do NOT exit low-power mode when we return from
    // the interrupt because there is nothing that needs to
    // be run in main().
} // End of ISR for the comparator

// ISR for the RF radio
#pragma vector=CC1101_VECTOR
__interrupt void CC1101_ISR(void)
{
    // Check for end-of-packet and the transmit request
    if(RF1AIV==RF1AIV_RFIFG9 && ucTransmitRequest)
    {
        // Since the end-of-packet was received and we
        // were requesting a transmission, we must be done transmitting
```

```
        Strobe(RF_SXOFF);       // Put radio back to sleep from IDLE mode
        RF1AIE &= ~BIT9;        // Disable TX end-of-packet interrupt

        // Clear the request for transmission since we've just finished
        // transmitting
        ucTransmitRequest=0;

        #ifdef RED_TRANSMIT_LED
        // Turn off RED after transmit
        P3OUT &= ~BIT6;
        #endif // RED_TRANSMIT_LED

        #if defined(PUSH_BUTTON) && defined(TRANSMIT_BUTTON)
        // Re-enable the push button, enabling P1.7 interrupt
        P1IE |= BIT7;
        #endif // PUSH_BUTTON && TRANSMIT_BUTTON

        // Do NOT exit low-power mode when we return from
        // the interrupt because there is nothing that needs to
        // be run in main().
    }
} // End of ISR for RF radio

// ISR for timer A1 CCR0, used to time 1 min
// (they have a really strange naming convention)
#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMER1_A0_ISR(void)
{
    #ifdef TEST_ACLK_ON_P1_0
    P1OUT ^= 0x01;                  // Toggle P1.0, GREEN
    #endif // TEST_ACLK_ON_P1_0

    // We've reached the value in TA1CCR0, which means we've hit a minute
    ucTransmitRequest=1;    // Request the transmit in main()

    // Exit low-power mode when we return from the interrupt
    // so that we may run the CPU code in main()
    __bic_SR_register_on_exit(LPM_BITS);

} // End of ISR for timer 1

#ifdef PUSH_BUTTON
// ISR for port 1, so that we can use the button on P1.7 as an interrupt
#pragma vector=PORT1_VECTOR
__interrupt void PORT1_ISR(void)
{
    // P1IFG7 is the interrupt flag for P1.7, so we
    // are checking if this flag is what cause the port
    // interrupt.
    if(P1IV==P1IV_P1IFG7)
    {
        #ifdef TRANSMIT_BUTTON
        // If P1.7 did cause the interrupt, then
        // someone must have pushed the button on P1.7,
        // so we can issue a transmit request.
        ucTransmitRequest=1;

        // We can debounce the button by disabling it
        // here and then re-enabling it when the
        // transmission is complete (we got an end-of-packet
        // interrupt from the radio core).
        P1IE &= ~BIT7;
        #endif // TRANSMIT_BUTTON

        #ifdef RESP_BUTTON
        // If P1.7 did cause the interrupt, then
        // someone must have pushed the button on P1.7,
        // so we can increment the number of breaths.
        ++ucRespRate;

        // We can debounce the button by disabling it
```

```
        // here and then re-enabling it in main().
        // Just checking it in main() after the if-statement
        // seems to be enough time since MCLK = 32.768 kHz
        P1IE &= ~BIT7;
        #endif // RESP_BUTTON

        // Exit low-power mode when we return from the interrupt
        // so that we may run the CPU code in main()
        __bic_SR_register_on_exit(LPM_BITS);
    }
} // End of ISR for port 1
#endif // PUSH_BUTTON

//*****************************************************************************
// Functions
//*****************************************************************************

// Start
// End
```

## A.1.2. Breath Interval Timing Algorithm: tx2dot3.c

```
//*****************************************************************************
// VERSION: tx2.3.c
// CREATED: 10/29/2011
// REVISED: 10/29/2011
// RX PAIR: rx0.2.c
// REQUIRED: cc430f6137.h, hal_pmm.h/c, RF1A.h/c, RfRegSettings.c
//
// COMMENTS:
// Respiratory rate is measured in br/min. We can use a stopwatch approach,
// which times the time between two breaths and inverts it to obtain br/min.
// This result can then be transmitted over RF.
//
// To do so in software, we set up a timer that interrupts and resets after the
// duration of the timer interval is up (whether we choose 10 sec, 15 sec,
// 30 sec, or 1 min). This signals the SoC to transmit a variable called
// ucRespRate over RF. We are going to use TA1, which has an interrupt priority
// lower than the radio. Since the timer is only 16 bits, it can only count up
// to 2^16 - 1 = 65535. Let us use the 32.768 kHz internal clock provided by
// the reference oscillator REFO so that we don't have to solder on the
// 32.768 kHz crystal onto XT1 pins. We'll use it as our master clock and
// also our timer TA1 base. Let us use the auxiliary clock ACLK to host TA1
// and turn off the sub-master clock SMCLK. ACLK will be sourced by the
// REFO, but will have a divisor of 32, which creates ACLK = 1024 Hz.
// (See pg.17 of the datasheet for interrupt priority details.)
//
// RF settings are set based off of the RF_Toggle_LED_Demo using the 915 MHz
// settings and a fixed packet length of 1 (byte), which fits in the FIFO.
// 915 MHz was chosen since it required less power than 868 MHz according
// to the datasheet. 868/915 MHz are the only available options for
// evaluation board EM430F6137RF900.
//
// This program will not have the ability to receive anything via RF; it will
// only be able to transmit data.
//
// We are using a sinusoidal waveform to simulate a person's breathing pattern
// and connect it to the comparator pin. Using particular settings, the
// comparator is able to interrupt for every rising edge of the sinsoidal
// signal. This interrupt is what allows determine when the "start" of a new
// breath has been detected. The time is logged for two breath "starts". When
// a transmission is required, the difference is found between these two logged
// times. This difference is then used to calculate the respiratory rate based
// on the timer units. For proper br/min units, timer units must be converted
// in to units of minutes. Also, when the two breaths have been captured,
// the comparator must either be turned off or have its interrupt ignored
// until transmission to avoid extra comparator captures. Turning off the
```

```
// comparator will most likely save the most power; however, understanding
// how its startup delay works is important. Thus, this code allows for both
// options of just disabling the interrupt and of disabling the whole comparator
// module for functional and power testing comparisons.
//
// Noise on the comparator output is cancelled out by using an adaptive
// comparator reference voltage. The signal does not change from low to high
// unless the input gets higher than Vcc*3/4 and does not change from high to
// low unless the input gets lower than Vcc*1/4. This leaves a (3/4-1/4)*Vcc =
// Vcc/2 gap for noise.
//
// For most of the time, the radio core and the CPU will be in sleep and
// low-power mode, respectively, until a transmission/calculation is required.
// When the timer interrupts to indicate that a transmission is required, the
// ISR will set the variable ucTransmitRequest to indicate to main() that it
// needs to initiate a transmission sequence. The interrupt exits to main()
// to proceed with the code for transmitting ucRespRate. Then, main() wakes the
// radio core up for transmission and puts the CPU back into low-power mode
// until the transmission is complete. ucRespRate is reset after completion.
//
// For even more power savings, the high and low side supply voltage supervisors
// and monitors were turned off.
//
//*****************************************************************************

//*****************************************************************************
// Debug code settings (Set these carefully!)
//*****************************************************************************
//#define PUSH_BUTTON               // Push button P1.7 interrupt forces tx
//#define RED_TRANSMIT_LED          // Red LED (RED) is on during transmission
//#define NONZERO_INITIAL_RESP_RATE // Sets the initial respiratory rate to 35
//#define GREEN_CAPTURE_LED         // The green LED (GREEN) is on during timer
                                    // capturing (time b/w captures on CBOUT)
//#define TEST_COMP_INT_ON_P1_1     // Toggles P1.1 for each comp. interrupt

//*****************************************************************************
// Power measurement variation settings (Set these carefully!)
//*****************************************************************************
#define TRANSMIT_INTERVAL_SELECT    // We can select among 10 s, 15 s, 30 s,
                                    // or 1 min transmit intervals
#define TURN_COMP_OFF               // This will turn the comparator off when
                                    // we aren't capturing
#define TRANSMIT_OFF                // Turn off transmission to measure power
                                    // for computational features

// Define the different settings for selecting the transmit interval to use
#ifdef TRANSMIT_INTERVAL_SELECT

    #define TRANSMIT_10_SEC     // Select transmission to occur every 10 sec
    //#define TRANSMIT_15_SEC       // Select transmission to occur every 15 sec
    //#define TRANSMIT_30_SEC       // Select transmission to occur every 30 sec
    //#define TRANSMIT_1_MIN        // Select transmission to occur every 1 min

    // Check if we've chosen a projection config
    #if (!defined(TRANSMIT_10_SEC) \
    && !defined(TRANSMIT_15_SEC) \
    && !defined(TRANSMIT_30_SEC) \
    && !defined(TRANSMIT_1_MIN))
    #error "Please select either \
    TRANSMIT_10_SEC, \
    TRANSMIT_15_SEC, \
    TRANSMIT_30_SEC, \
    or TRANSMIT_1_MIN as the active projection configuration"
    #endif

    // Check if we've erroneously chose more than one config
    #if ((defined(TRANSMIT_10_SEC) \
    + defined(TRANSMIT_15_SEC) \
    + defined(TRANSMIT_30_SEC) \
    + defined(TRANSMIT_1_MIN)) > 1)
    #error "Please select either \
```

```
    TRANSMIT_10_SEC, \
    TRANSMIT_15_SEC, \
    TRANSMIT_30_SEC, \
    or TRANSMIT_1_MIN as the active projection configuration"
    #endif

#endif // TRANSMIT_INTERVAL_SELECT

//*****************************************************************************
// Header files
//*****************************************************************************
#include "cc430f6137.h"
#include "hal_pmm.h"
#include "RF1A.h"

//*****************************************************************************
// Macros
//*****************************************************************************
#define  PATABLE_VAL        (0x8D)      // 0 dBm output
#define  LPM_BITS           (LPM4_bits) // Select low-power mode 4
#define  TMR_CLKS_PER_MIN   (61440)     // # of timer clks required for a minute

// Define timer tick based on transmit interval
//
// t = 10s = transmit interval we want in seconds
// c = 32.768kHz = CPU clock frequency
// d = 32 = ACLK frequency divisor
// c/d is the ACLK frequency that TA1 runs off of
// t(c/d) = timer clocks required for transmission
//
// Sample calculation...
// 10s ((32.768k clks/s) / 32) = 10,240 (tmr clks)/tick
//
#ifdef TRANSMIT_10_SEC
    #define  TMR_CLKS_PER_TA1_TICK  (10239) // (timer clk cycles per tick - 1)
#elif defined(TRANSMIT_15_SEC)
    #define  TMR_CLKS_PER_TA1_TICK  (15359) // (timer clk cycles per tick - 1)
#elif defined(TRANSMIT_30_SEC)
    #define  TMR_CLKS_PER_TA1_TICK  (30719) // (timer clk cycles per tick - 1)
#else   // default is 1 minute tick interval w/o projection
    #define  TMR_CLKS_PER_TA1_TICK  (61439) // (timer clk cycles per tick - 1)
#endif

//*****************************************************************************
// Global variables
//*****************************************************************************
extern RF_SETTINGS rfSettings;      // See RfRegSettings.c
unsigned int uiRespRate;            // br/min; also holds first time captured
unsigned char ucTransmitRequest;    // Set to cause resp rate tx in main()
unsigned char ucCaptureCount;       // # of captures done so far since last tx
unsigned int uiSecondCapture;       // Will hold the second time captured

//*****************************************************************************
// Prototypes
//*****************************************************************************

//*****************************************************************************
// The main program
//*****************************************************************************
void main(void)
{
     // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTHOLD;

    // Set master clock as soon as possible to save power immediately
    // CANNOT: Set master CPU clock source to be same clock as RF module
    //         (26 MHz RF crystal) since it is higher than the max freq
    //         of the chip (20 MHz)
    // CANNOT: Set master CPU clock source to be 32.678 kHz crystal, because
    //         it's not installed yet
    // THUS:   Set MCLK source as the 32.768 kHz REFO, same for ACLK and SMCLK
```

```
UCSCTL4 = SELA__REFOCLK + SELS__REFOCLK + SELM__REFOCLK;

// ACLK on external pin divisor = 1/1
// ACLK divisor = 1/32
// SMCLK divisor = 1/1
// MCLK divisor = 1/1
UCSCTL5 = DIVA__32;

// (Turn off XT2 crystal oscillator when the radio core is asleep)
// (Drive for XT1 is at max for 24-32MHz crystals)
// (Low-freq mode is chosen for XT1, caps must be defined)
// (XT1 is source internally rather than from a pin)
// (Default oscillator capacitors are chosen)
// Turn off subsystem master clock since we are not using it
// (XT1 is off when not used as a source for clocks or FLL)
UCSCTL6 |= SMCLKOFF;

// Initialize global variables
uiRespRate=0;           // Initial resp rate is 0 br/min
ucTransmitRequest=0;    // No transmission of resp request at start
ucCaptureCount=0;       // We haven't captured anything yet
uiSecondCapture=0;      // Nothing is captured yet

#ifdef NONZERO_INITIAL_RESP_RATE
uiRespRate=35;
#endif

// Increase PMMCOREV level to 2 for proper radio operation
// (Vcore change must be gradual)
SetVCore(2);

// Configure radio interface registers

// Set radio core to interrupt at the negative edge
// (for end-of-packet reading on RFIFG9)
RF1AIES |= BIT9;

// Set Power Management Module and Voltage Supply Supervisor (SVS)
// and Monitor (SVM)
PMMCTL0_H = 0xA5;
SVSMHCTL_H &= ~(SVSHE_H + SVMHE_H); // Disable high-side SVS and SVM
SVSMLCTL_H &= ~(SVSLE_H + SVMLE_H); // Disable low-side SVS and SVM
PMMCTL0_L |= PMMHPMRE_L;            // Allow for peripherals to request
                                    // high-power when in LPM
PMMCTL0_H = 0x00;

// Configure radio core through commands
Strobe(RF_SRES);                    // Reset radio core and put it to sleep
Strobe(RF_SNOP);                    // Reset radio pointer
WriteRfSettings(&rfSettings);       // Adjust to proper radio settings
WriteSinglePATable(PATABLE_VAL);    // Write to the power amplifier table

// Set default state of ports as outputs and put them to zero
// to unnecessary current consumption due to avoid floating inputs
PAOUT = 0x00;   // Ports 1 and 2
PADIR = 0xFF;
PBOUT = 0x00;   // Ports 3 and 4
PBDIR = 0xFF;
P5OUT = 0x00;   // Port 5
P5DIR = 0xFF;
PJOUT = 0x00;   // Port J (required because of JTAG debugger connection)
PJDIR = 0xFF;

#ifdef GREEN_CAPTURE_LED
// Set up P1.0 as an output for GREEN to indicate
// capturing is in progress
//P1SEL &= ~BIT0;       // Select I/O functionality for P1.0 (default)
//P1DS &= ~BIT0;        // Select reduced drive strength for P1.0 (default)
//P1REN &= ~BIT0;       // Pull-up/down resistor is a don't care on output
//P1IES &= BIT0;        // Low-to-high transition is a don't care on output
//P1IFG = 0;            // Clear the Port 1 vector interrupt flag
```

```
//P1IE &= ~BIT0;        // Interrupt enable is a don't care on output
P1OUT &= ~BIT0;         // Initialize GREEN as off
P1DIR |= BIT0;          // Set P1.0 as an output
#endif // GREEN_CAPTURE_LED

#ifdef RED_TRANSMIT_LED
// Set up P3.6 as an output for RED to indicate
// a transmission is in progress
//P3SEL &= ~BIT6;       // Select I/O functionality for P3.6 (default)
//P3DS &= ~BIT6;        // Select reduced drive strength for P3.6 (default)
P3OUT &= ~BIT6;         // Initialize RED as off
P3DIR |= BIT6;          // Set P3.6 as an output
#endif // RED_TRANSMIT_LED

#ifdef PUSH_BUTTON
// Set up P1.7 as a interrupt so that we may use
// the push bottom on it as a transmit message trigger
//P1SEL &= ~BIT7;       // Select I/O functionality for P1.7 (default)
//P1DS &= ~BIT7;        // Select reduced drive strength for P1.7 (default)
//P1DIR &= ~BIT7;       // Set P1.7 as an input (default)
P1REN |= BIT7;          // Set P1.7 REN & OUT to enable the pull-up resistor
P1OUT |= BIT7;
P1IES &= BIT7;          // Set that a low-to-high transition will set the
                        // P1.7 interrupt flag
P1IFG = 0;              // Clear the Port 1 vector interrupt flag
P1IE |= BIT7;           // Enable P1.7 as an interrupt
#endif  // PUSH_BUTTON

// Set up the comparator for adaptive VREF to achieve hysteresis

// Disable input channel that allows external input for the - terminal
// We don't need to choose a channel for the - terminal since it's disabled
// Enable input channel that allows external input for + terminal
// Select channel CB0 for the + terminal
CBCTL0 = CBIPEN + CBIPSEL_0;

// Comparator reference accuracy is set to static mode
// Reference voltage is disabled
// Resistor tap/string for VREF1 (CBOUT = 1) is set to Vcc*1/4
//      Ref p.69 of datasheet for CBREF1
//      Vcc*1/4 = Vcc*(n+1)/32, n = 7
// Comparator reference source is Vcc applied to the resistance ladder
// Reference voltage produced is connected to the negative terminal of comp.
// Resistor tap/string for VREF0 (CBOUT = 0) is set to Vcc*3/4
//      Ref p.69 of datasheet for CBREF0
//      Vcc*3/4 = Vcc*(n+1)/32, n = 23
CBCTL2 = CBREF1_7 + CBRS_1 + CBRSEL + CBREF0_23;

// Disable input buffer @ P2.0/CB0
CBCTL3 = CBPD0;

// Comparator output selects b/w VREF1 and VREF0 as comparator reference
// CBMRVS is a manual select for the comparator reference, which we don't
// care about.
// Turn on the comparator
// Choose ultra-low-power mode
// Select lowest filter delay of 450 ns
// Do not exchange the comparator 0 inputs
// Do not short the +/- comparator inputs
// Select the rising edge as the interrupt trigger
// Enable the comparator filter
// Do not invert CBOUT
// CBOUT is read-only so we don't care about this bit
CBCTL1 = CBON + CBPWRMD_2 + CBF;

// Set up timer TA1 to interrupt every transmission interval and also set
// it up for capturing the rising edges of a sinusoidal signal. TA1CCR0 is
// the register that keeps the value that TA1 counts up to. TA1CCR1 will
// capture the first rising edge of CBOUT and store it to the variable
// uiRespRate. Then, it will capture the rising edge of CBOUT a second time.
// The clk time between breaths will be equal to
```

```
// (second capture - first capture). This of course must be
// modified to it's true value in time (min) versus in clock cycles (clk).

// Set TA1CCR0 value to have a 1 sec tick
// We want TMR_CLKS_PER_TA1_TICK + 1 counts and the number of timer counts
// in a period is (TA1CCR0 + 1), thus we set it to TMR_CLKS_PER_TA1_TICK
TA1CCR0 = TMR_CLKS_PER_TA1_TICK;

// (Keep no capture on)
// (Keep default capture/compare input)
// (Keep default async capture source)
// (Stay in default compare mode)
// (Keep default output mode)
// Enable CCR0 interrupt
// (Keep default output)
// (Clear capture overflow indicator)
// (Clear interrupt flag)
TA1CCTL0 |= CCIE;

// Turn capture on for rising edge
// Set input to CCI1B input (CBOUT for CCR1 according to datasheet pg.27)
// Select synchronous capture
// Set to capture mode
// (Keep default output mode)
// Enable CCR1 interrupt
// (Keep default output)
// (Clear capture overflow indicator)
// (Clear interrupt flag)
TA1CCTL1 |= CM_1 + CCIS_1 + SCS + CAP + CCIE;

// Turn on GREEN on P1.0 to indicate capturing is occurring
#ifdef GREEN_CAPTURE_LED
P1OUT |= BIT0;
#endif // GREEN_CAPTURE_LED

// LOGIC: Using TA1 since it has less capture/compare registers
// Select Timer A1 clock as ACLK (starts timer)
// Keep TA1 divisor as 1
// Set TA1 to be in count up mode (count to value in TA1CCR0)
// Clear TA1
// Disable TAIFG interrupts for CCR1-CCR2
// Clear the interrupt flag at onset
TA1CTL = TASSEL__ACLK + ID__1 + MC__UP + TACLR;

#ifdef TEST_COMP_INT_ON_P1_1
// Disable inverted polarity interrupt
// Enable non-inverted polarity interrupt
// Clear inverted polarity interrupt flag
// Clear non-inverted polarity interrupt flag
CBINT = CBIE;
#endif // TEST_COMP_INT_ON_P1_1

// The bulk the program
while(1)
{
    // Enter low-power mode and enable interrupts
    __bis_SR_register(LPM_BITS + GIE);

    // After an interrupt wakes us up from a low-power mode
    // we need to check if we should be transmitting a packet
    if(ucTransmitRequest)
    {
        // Since we've had a request for transmission made,
        // we can begin transmission

        #ifdef RED_TRANSMIT_LED
        // Turn on RED at beginning of transmit
        P3OUT |= BIT6;
        #endif // RED_TRANSMIT_LED

        // Clear RFIFG9 flag for the end-of-packet interrupt
```

```
RF1AIFG &= ~BIT9;

// Check if we turned transmission on or off to determine
// if the following code should be compiled.
#ifdef TRANSMIT_OFF
    // No transmission should occur

    // Clear the request for transmission since we've
    // just finished transmitting
    ucTransmitRequest=0;
#else
    // Enable TX end-of-packet interrupt
    RF1AIE |= BIT9;
#endif // TRANSMIT_OFF

// Check if we have enough captures to make a
// valid rate estimate (req. two)
if(ucCaptureCount<2)
    uiRespRate=0;
else
{   // Calculate the respiratory rate, where uiRespRate
    // was used as a temporary holder and is now being
    // filled with its true value. The formula used will
    // depend of the timer has cycled around from the
    // value stored in TACCR0 to 0.
    //
    // If there is no overlap:
    // resp = (number of timer clks in a minute)/
    //        (second capture - first capture)
    //
    // If there is overlap:
    // resp = (number of timer clks in a minute)/
    //        (max number of timer clks -
    //        second capture + first capture)
    //
    if(uiSecondCapture>uiRespRate)      // No overlap
        uiRespRate=TMR_CLKS_PER_MIN/
        (uiSecondCapture-uiRespRate);
    else                                // Overlap
        uiRespRate=TMR_CLKS_PER_MIN/
        (TMR_CLKS_PER_TA1_TICK - uiRespRate + uiSecondCapture);
} // End if(ucCaptureCount<2)

// Check if we turned transmission on or off to determine
// if the following code should be compiled.
#ifdef TRANSMIT_OFF
    // No transmission should occur
#else
    // Write the rate to the TX FIFO
    WriteSingleReg(RF_TXFIFOWR,(unsigned char)uiRespRate);
#endif // TRANSMIT_OFF

// uiRespRate reset is unnecessary since we overwrite it anyway;
// this is unlike the count algorithm that needs to reset to
// function correctly; uiRespRate=0;

#ifdef NONZERO_INITIAL_RESP_RATE
uiRespRate=35;
#endif

// Turn the comparator back on if we turned it off in
// an attempt to save power
#ifdef TURN_COMP_OFF
CBCTL1 |= CBON;
#endif // TURN_COMP_OFF

// Check if we turned transmission on or off to determine
// if the following code should be compiled.
#ifdef TRANSMIT_OFF
    // No transmission should occur
#else
```

```
                // Wake up the radio core and enable TX processing
                Strobe(RF_STX);
            #endif // TRANSMIT_OFF

            // Reset the capture count and re-enable timer capturing
            ucCaptureCount=0;       // Clear the # of captures b/w transmissions
            TA1CCTL1 &= ~CCIFG;     // Clear capture flag
            TA1CCTL1 |= CCIE;       // Re-enable CCR1-CCR2 interrupts

        } // End of check for transmit request
    } // End of the while(1) of main()
} // End of main()

//*****************************************************************************
// Interrupt routines in order of priority from highest to lowest
//*****************************************************************************

#ifdef TEST_COMP_INT_ON_P1_1
// ISR for the comparator, this is replaced by the capture
// interrupt of TA1 (v1.0->2.0)
#pragma vector=COMP_B_VECTOR
__interrupt void COMP_B_ISR(void)
{
    P1OUT ^= BIT1;          // Toggle P1.1
    CBINT &= ~CBIFG;        // Clear interrupt flag for non-inverted polarity

    // Do NOT exit low-power mode when we return from
    // the interrupt because there is nothing that needs to
    // be run in main().
} // End of ISR for the comparator
#endif //TEST_COMP_INT_ON_P1_1

// ISR for the RF radio
#pragma vector=CC1101_VECTOR
__interrupt void CC1101_ISR(void)
{
    // Check for end-of-packet and the transmit request
    if(RF1AIV==RF1AIV_RFIFG9 && ucTransmitRequest)
    {
        // Since the end-of-packet was received and we
        // were requesting a transmission, we must be done transmitting
        Strobe(RF_SXOFF);       // Put radio back to sleep from IDLE mode
        RF1AIE &= ~BIT9;        // Disable TX end-of-packet interrupt

        // Clear the request for transmission since we've just
        // finished transmitting
        ucTransmitRequest=0;

        #ifdef RED_TRANSMIT_LED
        // Turn off RED after transmit
        P3OUT &= ~BIT6;
        #endif // RED_TRANSMIT_LED

        #ifdef PUSH_BUTTON
        // Re-enable the push button, enabling P1.7 interrupt
        P1IE |= BIT7;
        #endif // PUSH_BUTTON && TRANSMIT_BUTTON

        // Do NOT exit low-power mode when we return from
        // the interrupt because there is nothing that needs to
        // be run in main().
    }
} // End of ISR for RF radio

// ISR for timer A1 CCR0, used to time 1 min
// (they have a really strange naming convention)
#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMER1_A0_ISR(void)
{
    // We've reached the value in TA1CCR0, which means we've hit a minute
    ucTransmitRequest=1;    // Request the transmit in main()
```

```
    // Exit low-power mode when we return from the interrupt
    // so that we may run the CPU code in main()
    __bic_SR_register_on_exit(LPM_BITS);

} // End of ISR for timer 1 one-minute indicator

// ISR for timer A1 CCR1-CCR2, used to capture CBOUT
// (they have a really strange naming convention)
#pragma vector=TIMER1_A1_VECTOR
__interrupt void TIMER1_A1_ISR(void)
{
    // Check if a CBOUT rising edge have prompted a capture
    if(TA1IV==TA1IV_TA1CCR1)
    {
        // Check if this is our first capture or second
        if(!ucCaptureCount)
        {
            // This is our first capture, so we must store
            // what was captured while we wait for the next one.
            // Thus, we are going to temporarily store the number
            // of timer ticks within the respiratory rate variable.
            uiRespRate=TA1CCR1;      // Store first capture for calc during tx

            // Turn on GREEN on P1.0 to indicate first capture has occurred
            #ifdef GREEN_CAPTURE_LED
            P1OUT |= BIT0;
            #endif // GREEN_CAPTURE_LED
        }
        else
        {
            // This is our second capture so we can properly calculate
            // the respiratory rate and store it until the next RF
            // transmission. We can also disable this capture interrupt
            // until the next transmission.
            uiSecondCapture=TA1CCR1; // Store second capture for calc during tx
            TA1CCTL1 &= ~CCIE;       // Disable CCR1-CCR2 interrupts

            // Turn off GREEN on P1.0 to indicate second capture has occurred
            #ifdef GREEN_CAPTURE_LED
            P1OUT &= ~BIT0;
            #endif // GREEN_CAPTURE_LED

            // Turn the comparator off in an attempt to save power
            #ifdef TURN_COMP_OFF
            CBCTL1 &= ~CBON;
            #endif // TURN_COMP_OFF
        }

        // Increment the number of captures processed
        ++ucCaptureCount;

        // Do NOT exit low-power mode when we return from
        // the interrupt because there is nothing that needs to
        // be run in main().
    }
} // End of ISR for timer 1 capture

#ifdef PUSH_BUTTON
// ISR for port 1, so that we can use the button on P1.7 as an interrupt
#pragma vector=PORT1_VECTOR
__interrupt void PORT1_ISR(void)
{
    // P1IFG7 is the interrupt flag for P1.7, so we
    // are checking if this flag is what cause the port
    // interrupt.
    if(P1IV==P1IV_P1IFG7)
    {
        // If P1.7 did cause the interrupt, then
        // someone must have pushed the button on P1.7,
        // so we can issue a transmit request.
```

```
        ucTransmitRequest=1;

        // We can debounce the button by disabling it
        // here and then re-enabling it when the
        // transmission is complete (we got an end-of-packet
        // interrupt from the radio core).
        P1IE &= ~BIT7;

        // Exit low-power mode when we return from the interrupt
        // so that we may run the CPU code in main()
        __bic_SR_register_on_exit(LPM_BITS);
    }
} // End of ISR for port 1
#endif // PUSH_BUTTON

//*****************************************************************************
// Functions
//*****************************************************************************

// Start
// End
```

## A.2. Receiver Module

The following are two iterations of code for the receiver module. The first iteration utilizes two LEDs to indicate to the user the respiratory rate that was received from an RF transmission. Assuming that the respiratory rate would not reach the hundreds, one LED blinked to indicate the tens digit of the rate and other blinked for the ones digit. Each blink occurred every other half-second.

There are two issues with this LED method that prompted the second RS232 iteration of code: (1) Watching an LED blink for a “large-digit” number took a very long time and the user had to carefully count each blink, allowing the result to be prone to error from human reaction time. “Large-digit” refers to a number that has a high number in each decimal place. Another way to view it is a number in which the sum of the digits is high relative to all other numbers; for example “99” is the highest two-digit “large-digit” number with a sum of 18. Thus, 18 blinks are required in total, which results in 18 seconds of viewing since a full blink requires a half-second “on” time and another half-second “off” time. On the other hand, “11” is a “small-digit” number which would only require 2 seconds of viewing. (2) Since the data-displaying algorithm has the potential to be quite long relative the reception rate of new data, there is a possibility for the LED indications to interfere with the proper RF reception of data. Thus, RS232 was implemented in order to assist in mitigating the time required to display the received data to the user.

### A.2.1. LED Iteration: rx0dot0.c

```
//******************************************************************************
// VERSION: rx0.0.c
// CREATED: 08/16/2011
// REVISED: 08/16/2011
// TX PAIR: tx0.0.c - tx0.5.c, tx1.0.c
// REQUIRED: cc430f6137.h, hal_pmm.h/c, RF1A.h/c, RfRegSettings.c
//
// COMMENTS:
// This program is heavily based on the RF_Toggle_LED_Demo.c. If a message is
// not received within a minute, the red LED remains on. Otherwise, when a
// message is received the ucRespRate within the packet is indicated to the
// user via LED flashes (timed for every other 500 ms so that the blinks can
// be perceived by human eyes). The green LED flashes the number of
// times equivalent to the tens digit of the ucRespRate, while the red LED
// flashes the number of times equivalent to the ones digit. For example,
// a respiratory rate of 53 would be five green flashes and 3 red flashes.
// The flaw with this is that to be human readable, the 500 ms flashes must be
// done before the next transmission is received. Another flaw, is that a zero
// ucRespRate will not flash anything, which can be confused with the board
// malfunctioning. There is also some unnecessary TX code from the sample code.
//
// Note: The clock settings are actually incorrect; however, this was not
// noticed until a later version of this code. This paragraph is being
// commented after-the-fact, and we do not want to alter anything that might
// create functional discrepancies.
//
//******************************************************************************
#include "cc430x613x.h"
#include "RF1A.h"
#include "hal_pmm.h"

void Transmit(unsigned char *buffer, unsigned char length);
void ReceiveOn(void);
void ReceiveOff(void);

void InitButtonLeds(void);
void InitRadio(void);

#define  PACKET_LEN         (0x01)          // PACKET_LEN <= 61
#define  RSSI_IDX           (PACKET_LEN)    // Index of appended RSSI
#define  CRC_LQI_IDX        (PACKET_LEN+1)  // Index of appended LQI, checksum
#define  CRC_OK             (BIT7)          // CRC_OK bit
#define  PATABLE_VAL        (0x8D)          // 0 dBm output
#define  RX_TIMEOUT         (120)           // (# of seconds to wait x 2) =
                                            // (# of 500ms ticks)

extern RF_SETTINGS rfSettings;

unsigned char packetReceived;
unsigned char packetTransmit;

unsigned char RxBuffer[PACKET_LEN+2];
unsigned char RxBufferLength = 0;
const unsigned char TxBuffer[PACKET_LEN]={69};
unsigned char buttonPressed = 0;
unsigned int i = 0;

unsigned char transmitting = 0;
unsigned char receiving = 0;
unsigned char ucReceivedMessage = 0;
unsigned char uc500msTick = 0;

void main( void )
{
    unsigned char x=0;

  // Stop watchdog timer to prevent time out reset
```

```
WDTCTL = WDTPW + WDTHOLD;

  // Keep default XT1CLK (currently not on unless selected as source)
  //      as ACLK source
  // Set sub-master clock source to be XT1CLK (32.768K crystal)
  // CANNOT: Set master CPU clock to be same clock as RF module
  //         (26 MHz RF crystal) since it is higher than the max
  //         freq of the chip (20 MHz)
  // THUS: Set MCLK as the 32.768K crystal
  UCSCTL4 = SELA__XT1CLK + SELS__XT1CLK + SELM__XT1CLK;

  // Turn off subsystem master clock since we are not using it
  UCSCTL6 |= SMCLKOFF;

  // Keep default 1/1 divisor for ACLK on external pin
  // Keep default 1/1 divisor for ACLK
  // Keep default 1/1 divisor for SMCLK
  // Keep default 1/1 divisor for MCLK
  //UCSCTL5 = DIVPA__1 + DIVA__1 + DIVS__1 + DIVM__1;

// Increase PMMCOREV level to 2 for proper radio operation
SetVCore(2);

ResetRadioCore();
InitRadio();
InitButtonLeds();

ReceiveOn();
receiving = 1;

  // (Keep no capture on)
  // (Keep default capture/compare input)
  // (Keep default async capture source)
  // (Stay in default compare mode)
  // (Keep default output mode)
  // Enable CCR0 interrupt
  // (Keep default output)
  // (Clear capture overflow indicator)
  // (Clear interrupt flag)
  TA1CCTL0 = CCIE;

  // Set TA1CCR0 value to have a 500ms tick
  // We want 16,384 counts and the number of timer counts
  // in a period is (TA1CCR0 + 1), thus we set it to 15,3843
  TA1CCR0=16383;

  // (Using TA1 since it has less capture/compare registers)
  // Select Timer A1 clock as SMCLK (starts timer)
  // Keep TA1 divisor as 1
  // Set TA1 to be in count up mode (count to value in TA1CCR0)
  // Clear TA1
  // (Disable TAIFG interrupts)
  // (Clear the interrupt flag at onset)
  TA1CTL = TASSEL__ACLK + ID__1 + MC__UP + TACLR;

while(1)
{
  __bis_SR_register(LPM4_bits + GIE);
  __no_operation();

  // Check if we have successfully received a message
  if(ucReceivedMessage)
  {
      // Turn off red timeout LED
      P3OUT &= ~BIT6;

      // We can reset the timer and use it
      // for timing the LED blinks to the user

      // Pulse the green LED a many times as the tens digit
      // of the number received through RF comm; toggle every 500ms
```

```
        for(x=0;x<2*(RxBuffer[0]/10);++x)
        {
            P1OUT ^= BIT0;                  // Toggle GREEN
            TA1CTL|=TACLR;                  // Clear TA1
            TA1CTL&=~TAIFG;                 // Reset TA1 Flag
            while(!(TA1CTL&TAIFG));         // Wait 500ms
            TA1CTL&=~TAIFG;                 // Reset TA1 Flag
        }

        // Pulse the red LED a many times as the ones digits
        // of the number received through RF comm; toggle every 500ms
        for(x=0;x<2*(RxBuffer[0]%10);++x)
        {
            P3OUT ^= BIT6;                  // Toggle RED
            TA1CTL|=TACLR;                  // Clear TA1
            TA1CTL&=~TAIFG;                 // Reset TA1 Flag
            while(!(TA1CTL&TAIFG));         // Wait 500ms
            TA1CTL&=~TAIFG;                 // Reset TA1 Flag
        }

        // Reset number of ticks we have been waiting
        uc500msTick=0;

        // Reset that we need to receive a message
        // (this means that any message received while we were blinking is
        // ignored; need++ to not ignore)
        ucReceivedMessage=0;
    }

    // Poll to see if we've have gotten a reply w/in a min and turn on
    // the red LED if we have timed out.
    if(uc500msTick>=RX_TIMEOUT)
    {
        P3OUT |= BIT6;
        uc500msTick=0;
    }

    // Transmitting/Receiving operations
    if (buttonPressed)                      // Process a button press->transmit
    {
      P3OUT |= BIT6;                        // Pulse LED during Transmit
      buttonPressed = 0;
      P1IFG = 0;

      ReceiveOff();
      receiving = 0;
      Transmit( (unsigned char*)TxBuffer, sizeof TxBuffer);
      transmitting = 1;

      P1IE |= BIT7;                         // Re-enable button press
    }
    else if(!transmitting)
    {
      ReceiveOn();
      receiving = 1;
    }
  }
}

void InitButtonLeds(void)
{
  // Set up the button as interruptible
  P1DIR &= ~BIT7;
  P1REN |= BIT7;
  P1IES &= BIT7;
  P1IFG = 0;
  P1OUT |= BIT7;
  P1IE  |= BIT7;

  // Initialize Port J
  PJOUT = 0x00;
```

```
  PJDIR = 0xFF;

  // Set up LEDs
  P1OUT &= ~BIT0;
  P1DIR |= BIT0;
  P3OUT &= ~BIT6;
  P3DIR |= BIT6;
}

void InitRadio(void)
{
  // Set the High-Power Mode Request Enable bit so LPM3 can be entered
  // with active radio enabled
  PMMCTL0_H = 0xA5;
  PMMCTL0_L |= PMMHPMRE_L;
  PMMCTL0_H = 0x00;

  WriteRfSettings(&rfSettings);

  WriteSinglePATable(PATABLE_VAL);
}

#pragma vector=PORT1_VECTOR
__interrupt void PORT1_ISR(void)
{
  switch(__even_in_range(P1IV, 16))
  {
    case  0: break;
    case  2: break;                          // P1.0 IFG
    case  4: break;                          // P1.1 IFG
    case  6: break;                          // P1.2 IFG
    case  8: break;                          // P1.3 IFG
    case 10: break;                          // P1.4 IFG
    case 12: break;                          // P1.5 IFG
    case 14: break;                          // P1.6 IFG
    case 16:                                 // P1.7 IFG
      P1IE = 0;                              // Debounce by disabling buttons
      buttonPressed = 1;
      __bic_SR_register_on_exit(LPM4_bits); // Exit active
      break;
  }
}

void Transmit(unsigned char *buffer, unsigned char length)
{
  RF1AIES |= BIT9;
  RF1AIFG &= ~BIT9;                          // Clear pending interrupts
  RF1AIE |= BIT9;                            // Enable TX end-of-packet interrupt

  WriteBurstReg(RF_TXFIFOWR, buffer, length);

  Strobe( RF_STX );                          // Strobe STX
}

void ReceiveOn(void)
{
  RF1AIES |= BIT9;                           // Falling edge of RFIFG9
  RF1AIFG &= ~BIT9;                          // Clear a pending interrupt
  RF1AIE  |= BIT9;                           // Enable the interrupt

  // Radio is in IDLE following a TX, so strobe SRX to enter Receive Mode
  Strobe( RF_SRX );
}

void ReceiveOff(void)
{
  RF1AIE &= ~BIT9;                           // Disable RX interrupts
  RF1AIFG &= ~BIT9;                          // Clear pending IFG

  // It is possible that ReceiveOff is called while radio is receiving a packet.
  // Therefore, it is necessary to flush the RX FIFO after issuing IDLE strobe
```

```
  // such that the RXFIFO is empty prior to receiving a packet.
  Strobe( RF_SIDLE );
  Strobe( RF_SFRX  );
}

#pragma vector=CC1101_VECTOR
__interrupt void CC1101_ISR(void)
{
  switch(__even_in_range(RF1AIV,32))        // Prioritizing Radio Core Interrupt
  {
    case  0: break;                         // No RF core interrupt pending
    case  2: break;                         // RFIFG0
    case  4: break;                         // RFIFG1
    case  6: break;                         // RFIFG2
    case  8: break;                         // RFIFG3
    case 10: break;                         // RFIFG4
    case 12: break;                         // RFIFG5
    case 14: break;                         // RFIFG6
    case 16: break;                         // RFIFG7
    case 18: break;                         // RFIFG8
    case 20:                                // RFIFG9
      if(receiving)                 // RX end of packet
      {
        // Read the length byte from the FIFO
        RxBufferLength = ReadSingleReg( RXBYTES );
        ReadBurstReg(RF_RXFIFORD, RxBuffer, RxBufferLength);

        // Stop here to see contents of RxBuffer
        __no_operation();

        // Check the CRC results
        if(RxBuffer[CRC_LQI_IDX] & CRC_OK)
            {//P1OUT ^= BIT0;       // Toggle LED1
            ucReceivedMessage=1;}
      }
      else if(transmitting)         // TX end of packet
      {
        RF1AIE &= ~BIT9;            // Disable TX end-of-packet interrupt
        P3OUT &= ~BIT6;             // Turn off LED after Transmit
        transmitting = 0;
      }
      else while(1);                // trap
      break;
    case 22: break;                 // RFIFG10
    case 24: break;                 // RFIFG11
    case 26: break;                 // RFIFG12
    case 28: break;                 // RFIFG13
    case 30: break;                 // RFIFG14
    case 32: break;                 // RFIFG15
  }
    __bic_SR_register_on_exit(LPM4_bits);
}

// Timer A0 interrupt service routine
#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMER1_A0_ISR(void)
{
  ++uc500msTick;
  __bic_SR_register_on_exit(LPM4_bits);
}
```

## A.2.2. RS232 Iteration: rx0dot3.c

```
//*****************************************************************************
// VERSION: rx0.3.c
// CREATED: 10/29/2011
// REVISED: 10/29/2011
```

```
// TX PAIR: tx1.1.c, tx2.1.c
// REQUIRED: cc430f6137.h, hal_pmm.h/c, RF1A.h/c, RfRegSettings.c
//
// COMMENTS:
// This code receives the respiratory rate via RF communication at 915 MHz from
// a transmitting module. It then displays this rate using RS232. If a message
// is not received within a minute, the red LED remains on. There seems to be a
// bug somewhere; after a seemingly random number of correct correspondences,
// this module will timeout and refuse to receive any RF information. However,
// this module is only used to illustrate the an RF signal can be received
// correctly and that the rate can be correctly extracted with the transmitting
// unit. Thus, since the code generally works, complete debugging was not
// pursued in the interest of time. The blinking light version in rx0.0.c
// was reliable; the only reason we switched to RS232 is that it is faster to
// display the result and less prone to human error.
//
//*****************************************************************************
// Debug code settings (Set these carefully!)
//*****************************************************************************
#define GREEN_SEND_LED          // Green LED will remain on during an RS232 send

//*****************************************************************************
// Header files
//*****************************************************************************
#include "cc430x613x.h"
#include "RF1A.h"
#include "hal_pmm.h"

//*****************************************************************************
// Macros
//*****************************************************************************
#define  PATABLE_VAL        (0x8D)      // 0 dBm output
#define  LPM_BITS           (LPM4_bits) // Select low-power mode 4
#define  CLKS_PER_TA1_TICK  (61439)     // (clk cycles per tick - 1) = 1 minute

#define  PACKET_LEN         (0x01)          // PACKET_LEN <= 61
#define  RSSI_IDX           (PACKET_LEN)    // Index of appended RSSI
#define  CRC_LQI_IDX        (PACKET_LEN+1)  // Index of appended LQI, checksum
#define  CRC_OK             (BIT7)          // CRC_OK bit

//*****************************************************************************
// Global variables
//*****************************************************************************
extern RF_SETTINGS rfSettings;          // RfRegSettings0dot1.c
unsigned char ucReceivedMessage;        // Set to indicate resp rate
                                        // reception to main()
unsigned char RxBuffer[PACKET_LEN+2];
unsigned char RxBufferLength = 0;

//*****************************************************************************
// Prototypes
//*****************************************************************************

//*****************************************************************************
// The main program
//*****************************************************************************
void main( void )
{
    // Stop watchdog timer to prevent timeout reset
    WDTCTL = WDTPW + WDTHOLD;

    // Set master clock as soon as possible to save power immediately
    // CANNOT: Set master CPU clock source to be same clock as RF module
    //         (26 MHz RF crystal) since it is higher than the max freq
    //         of the chip (20MHz)
    // CANNOT: Set master CPU clock source to be 32.678 kHz crystal, because
    //         it's not installed yet
    // THUS:   Set MCLK source as the 32.768 kHz REFO, same for ACLK and SMCLK
    UCSCTL4 = SELA__REFOCLK + SELS__REFOCLK + SELM__REFOCLK;

    // ACLK on external pin divisor = 1/1
```

```
// ACLK divisor = 1/32
// SMCLK divisor = 1/1
// MCLK divisor = 1/1
UCSCTL5 = DIVA__32;

// (Turn off XT2 crystal oscillator when the radio core is asleep)
// (Drive for XT1 is at max for 24-32MHz crystals)
// (Low-freq mode is chosen for XT1, caps must be defined)
// (XT1 is source internally rather than from a pin)
// (Default oscillator capacitors are chosen)
// Turn off subsystem master clock since we are not using it
// (XT1 is off when not used as a source for clocks or FLL)
//UCSCTL6 |= SMCLKOFF; we need this clock for the UART for now...

// Initialize global variables
ucReceivedMessage=0;    // No transmission of resp request at start

// Increase PMMCOREV level to 3 for proper MAX3232 operation
// since it needs 3.3 V
SetVCore(3);

// Configure radio interface registers

// Set radio core to interrupt at the negative edge
// (for end-of-packet reading on RFIFG9)
RF1AIES |= BIT9;

// Set the High-Power Mode Request Enable bit so LPM3 can be entered
// with active radio enabled
PMMCTL0_H = 0xA5;
PMMCTL0_L |= PMMHPMRE_L;
PMMCTL0_H = 0x00;

// Configure radio core through commands
Strobe(RF_SRES);                    // Reset radio core and put it to sleep
Strobe(RF_SNOP);                    // Reset radio pointer
WriteRfSettings(&rfSettings);       // Adjust to proper radio settings
WriteSinglePATable(PATABLE_VAL);    // Write to the power amplifier table

// Set default state of ports as outputs and put them to zero
// to unnecessary current consumption due to avoid floating inputs
PAOUT = 0x00;   // Ports 1 and 2
PADIR = 0xFF;
PBOUT = 0x00;   // Ports 3 and 4
PBDIR = 0xFF;
P5OUT = 0x00;   // Port 5
P5DIR = 0xFF;
PJOUT = 0x00;   // Port J (required because of JTAG debugger connection)
PJDIR = 0xFF;

// Set up the UART module

// Set up the TX pin for the UART module
// P1DIR |= BIT6;           // Set P1.6 as TX output
P1SEL |= BIT6;              // Select P1.6 for TX UART functionality

// First hold the USCI module while we edit its settings
// for UART functionality
UCA0CTL1 |= UCSWRST;

// (Disable parity)
// (Default select odd parity)
// (LSB first for TX/RX shift register)
// (Character length is 8-bit)
// (One stop bit)
// (UART mode)
// (Asynchronous mode)
// UCA0CTRL0 = 0x00;

// Select SMCLK as USCI BRCLK clock source
// (Reject erroneous char and don't set the RX flag)
```

```
// (Receive break char and don' set the RX flag)
// Use sleep mode
// (Next frame transmitted is data, not address)
// (Next frame transmitted is not a break)
// (DO NOT TOUCH UCSWRST (bit0) !)
UCA0CTL1 |= UCSSEL__SMCLK + UCDORM;

// Set the baud rate to be 9600; 32kHz/9600=3.41 (See User's Guide)
UCA0BR1 = 0x00;     // Set the high byte
UCA0BR0 = 0x03;     // Set the low byte

// Set modulation for 9600 baud (See User's Guide)
// Modulation UCBRSx=3, UCBRFx=0, no oversampling
UCA0MCTL = UCBRS_3 + UCBRF_0;

// Release USCI from reset since we are done setting it up
UCA0CTL1 &= ~UCSWRST;

#ifdef GREEN_SEND_LED
// Set up P1.0 as an output for GREEN to indicate
// an RS232 send is in progress
//P1SEL &= ~BIT0;       // Select I/O functionality for P1.0 (default)
//P1DS &= ~BIT0;        // Select reduced drive strength for P1.0 (default)
//P1REN &= ~BIT0;       // Pull-up/down resistor is a don't care on output
//P1IES &= BIT0;        // Low-to-high transition is a don't care on output
//P1IFG = 0;            // Clear the Port 1 vector interrupt flag
//P1IE &= ~BIT0;        // Interrupt enable is a don't care on output
P1OUT &= ~BIT0;         // Initialize GREEN as off
P1DIR |= BIT0;          // Set P1.0 as an output
#endif // GREEN_SEND_LED

// Set up timer TA1 to interrupt every minute

// Set TA1CCR0 value to have a 1 sec tick
// We want CLKS_PER_TA1_TICK + 1 counts and the number of timer counts
// in a period is (TA1CCR0 + 1), thus we set it to CLKS_PER_TA1_TICK
TA1CCR0 = CLKS_PER_TA1_TICK;

// (Keep no capture on)
// (Keep default capture/compare input)
// (Keep default async capture source)
// (Stay in default compare mode)
// (Keep default output mode)
// Enable CCR0 interrupt
// (Keep default output)
// (Clear capture overflow indicator)
// (Clear interrupt flag)
TA1CCTL0 |= CCIE;

// LOGIC: Using TA1 since it has less capture/compare registers
// Select Timer A1 clock as ACLK (starts timer)
// Keep TA1 divisor as 1
// Set TA1 to be in count up mode (count to value in TA1CCR0)
// Clear TA1
// Disable TAIFG interrupts for CCR1-CCR2
// Clear the interrupt flag at onset
TA1CTL = TASSEL__ACLK + ID__1 + MC__UP + TACLR;

// Set the radio to receive mode as it will remain here forever
RF1AIFG &= ~BIT9;               // Clear a pending interrupt
RF1AIE  |= BIT9;                // Enable the interrupt
Strobe(RF_SRX);                 // Send command to radio to enter RX mode

// The bulk the program
while(1)
{
    // Enter low-power mode and enable interrupts
    __bis_SR_register(LPM_BITS + GIE);

    // Check if we have successfully received a message
    if(ucReceivedMessage)
```

```
        {
            #ifdef GREEN_SEND_LED
            // Turn on green led to indicate we are sending
            P1OUT |= BIT0;
            #endif // GREEN_SEND_LED

            // Send the respiratory rate received
            while (!(UCA0IFG&UCTXIFG)); // Wait for USCI_A0 TX buff to be empty
            UCA0TXBUF = '0' + (RxBuffer[0]/10);
            while (!(UCA0IFG&UCTXIFG)); // Wait for USCI_A0 TX buff to be empty
            UCA0TXBUF = '0' + (RxBuffer[0]%10);
            while (!(UCA0IFG&UCTXIFG)); // Wait for USCI_A0 TX buff to be empty
            UCA0TXBUF = ' ';

            #ifdef GREEN_SEND_LED
            // Turn off green led to indicate we finished sending
            P1OUT &= ~BIT0;
            #endif // GREEN_SEND_LED

            // Set the radio to receive mode again
            RF1AIFG &= ~BIT9;          // Clear a pending interrupt
            RF1AIE  |= BIT9;           // Enable the interrupt
            Strobe(RF_SRX);            // Send command to radio to enter RX mode

            // Reset timeout since we've received something
            TA1CTL |= TACLR;    // Clear timeout timer
            P3OUT &= ~BIT6;     // Turn off timeout light

            // Reset that we need to receive a message
            // (this means that any message received while we were
            // blinking is ignored; need++ to not ignore)
            ucReceivedMessage=0;

        } // End of if(ucReceivedMessage)
    } // End of while(1)
} // End of main()

//*****************************************************************************
// Interrupt routines in order of priority from highest to lowest
//*****************************************************************************

// ISR for the RF radio
#pragma vector=CC1101_VECTOR
__interrupt void CC1101_ISR(void)
{
    // Check for end-of-packet while in RX mode
    if(RF1AIV==RF1AIV_RFIFG9)
    {
        // Read the length byte from the FIFO
        RxBufferLength = ReadSingleReg( RXBYTES );
        ReadBurstReg(RF_RXFIFORD, RxBuffer, RxBufferLength);

        // Check the CRC results
        if(RxBuffer[CRC_LQI_IDX] & CRC_OK)
            ucReceivedMessage=1;

        // Exit low-power mode when we return from the interrupt
        // so that we may run the CPU code in main()
        __bic_SR_register_on_exit(LPM_BITS);
    }
} // End of ISR for RF radio

// ISR for timer A1 CCR0, used to time 1 min
// (they have a really strange naming convention)
#pragma vector=TIMER1_A0_VECTOR
__interrupt void TIMER1_A0_ISR(void)
{
    // We've reached the value in TA1CCR0, which means we've hit a minute
    P3OUT |= BIT6;  // Turn on RED to indicate a timeout

    // Do NOT exit low-power mode when we return from
```

```
    // the interrupt because there is nothing that needs to
    // be run in main().
} // End of ISR for timer 1

//*****************************************************************************
// Functions
//*****************************************************************************

// Start
// End
```

## A.3. TI Sample Code

In addition to the code that contains the algorithms, there are other files that need to be included in order for the compiler to build the programs properly. These files are based off of or taken directly from TI sample code and are briefly described in the following sub-appendices; The first sub-appendix indicates code that is necessary for the both the transmitter and receiver codes in the previous sub-appendix to run, and the second sub-appendix elicits code used for the preliminary testing of the SoC's RF functionality mentioned in Appendix D.

### A.3.1. Libraries Required For This Study

The files cc430f6137.h, hal_pmm.h, hal_pmm.c, RF1A.h, RF1A.c, and RFRegSettings.c are all required for the proper compilation of the transmitter and receiver programs of this study. Figure A.1 depicts the hierarchical file-dependency tree of these files, indicating their relationships to the transmitter and receiver programs; files lower in the tree depend on and require the files higher than them. All of the files are available at TI's website [**45**]. For RFRegSettings.c to function properly, the macro "`MHZ_915`" must be defined to select the proper 915 MHz antenna settings, and the struct parameter "`PKTLEN`" must be set to `0x01`.

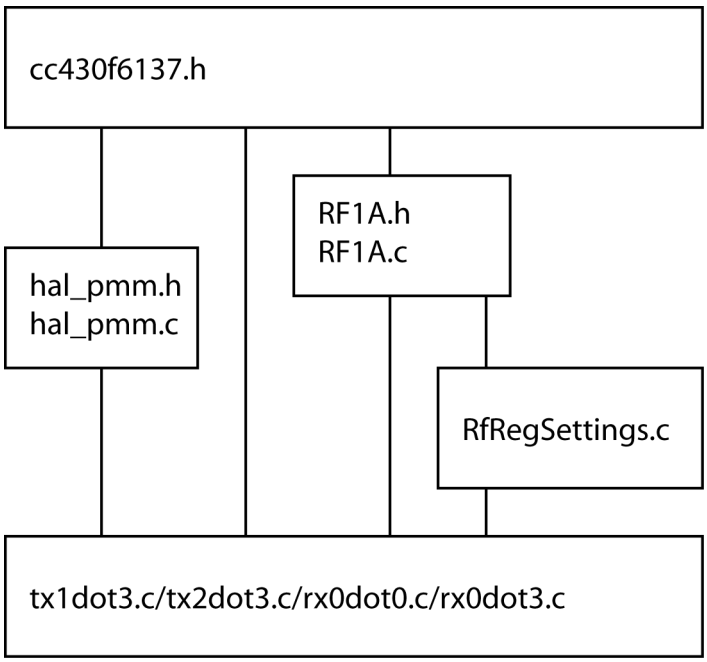



**Figure A.1: File-Dependency Tree.**

### A.3.2. Other Code Used During Development

As mentioned in Appendix D, preliminary testing was conducted to test the RF functionality of the SoCs utilizing the sample code files RF_Toggle_LED_Demo.h and RF_Toggle_LED_Demo.c available at TI's website [**45**]. These files and the files in the Sub-appendix A.3.1 should *all* be compiled together, and the result should then be programmed onto both the transmitter and receiver SoCs (since the code allows for bidirectional communication). If programmed onto EM430F6137RF900 boards, pressing the button, connected to Pin 7 on Port 1, on one board should cause a transmission commanding the other board's green LED, connected to Pin 0 of Port 1, to toggle.

# Appendix B: MATLAB Code For Algorithm Simulations

In order to better illustrate how the two algorithms compared in this thesis function, MATLAB was used to simulate their expected behavior. This code was developed for the demonstration of the algorithm logic and provides estimated error calculations in certain case scenarios. However, these simulations should not be considered a full representation of the problem. Converting C-code meant for a microcontroller to m-code meant for MATLAB poses various logistical challenges and several areas for possible misrepresentations of the C-code's logic.

For instance, a microcontroller contains several elements that MATLAB lacks; these include interrupt capabilities, the ability to run special hardware modules simultaneously, and physical analog interfaces to the outside world. To compensate for these "missing" capabilities, sequential software emulation is required.

Also, the concept of "real-time execution" must be estimated and rationalized into a pre-defined set of "time points" matching with a set of "value points". These points must be stored in equally sized vectors so that they can be plotted on an XY-graph. For example, in order to illustrate the continuous analog sinusoidal input to the comparator, two vectors of discrete points are required.

Normally, XY points are given in a manner that each point is isolated from the other, such as in the case of points (1,2), (3,4), and (5,6). However, MATLAB does not use this sort of "isolated point" storage system; it combines the X-coordinates into a vector and the Y-coordinates into another vector. The aforementioned set of points can be can be alternatively represented as X- and Y- vectors [1,3,5] and [2,4,6], respectively. MATLAB requires this vector format to efficiently handle and plot large sets of points.

Continuing with the sinusoidal comparator input example, the equation that needs to be plotted is $y(t) = A*sin(\omega t + \phi) + B$, where the constants $A$, $\omega$, $\phi$, and $B$ represent amplitude, angular frequency, phase shift, and signal offset, respectively. The function $y(t)$ only varies with the value of time, $t$; thus, to plot the sine function with the X-axis as time and the Y-axis as the value of $y(t)$ at that time, a set of values for $t$ and a set of corresponding values for $y(t)$ are required to form a set of points to plot. The time values for $t$ would be stored in the X-vector, and the sine values for $y(t)$ at those time values

would be stored in the Y-vector. The resulting X- and Y- vectors would look something like $[t_1,t_2,t_3]$ and $[y(t_1),y(t_2),y(t_3)]$, respectively. In particular, for the simulations created in this study, the X-vector for the "analog" sinusoid contains the times at which the rising edge of the CPU clock should occur, while the Y-vector contains the corresponding values at those times. The CPU clock's rising edge was chosen as the most appropriate time base, since, in general, code calculations depend on or reference this clock for execution.

Given that the time an event occurs must be precisely calculated in MATLAB, as opposed to the timing of microcontroller events, which happen situationally, this thesis's MATLAB simulations required some trickery in order to sidestep microcontroller and MATLAB differences. One of the subtleties that were dealt with was that MATLAB does not have physical hardware. Therefore, if there is considerable amount of computational or propagational delay for the emulated microcontroller, these delays must be coded to occur in MATLAB.

For time critical occurrences that have a variable delay, MATLAB simulation becomes complex with respect to the proper point values to plot. However, for this thesis, MATLAB is only used to provide the reader with a general idea of how the algorithms work, so exact timing is not too critical and can generally be estimated using the time base and units of CPU clock cycles.

Despite this thesis's ability to utilize this simplification, there are still a few somewhat complex considerations in regards to the "real-time" emulation and the actual procedural execution of the m-code. More specifically, careful inspection of the code sequence must be done since m-code execution can occur within a single "real-time" time point.

Consider a case scenario specific to this thesis; a transmission occurs after one minute has elapsed. The transmission is displayed using points which oscillate between having a Y-value of logical "1" or "0". The code should plot a "1" to illustrate when a transmission is occurring, and a "0" otherwise. The time points at which the Y-value should switch from "1" to "0", or vice versa, are stored in the in the transmission signal's X-vector with the corresponding Y-values in the transmission signal's Y-vector. Keeping in mind that the time base of this study is the CPU clock, it is possible to determine when

to transition the transmission signal from high-to-low or low-to-high in a "for-loop" that has the CPU clock cycle as its "auto-incrementing" variable.

During the iteration of the loop in which around 60 seconds is reached (actually stored in units of CPU clock cycles), the transmission should occur. However, this iteration of the loop cannot be marked as the time to set the transmission, since at the "beginning" of loop, the $60^{th}$ second is just beginning and a full minute has not elapsed yet. For a full minute to be valid, the loop must emulate going completely through the $60^{th}$ second, before setting the transmission signal, and thus, the m-code should not be programmed to mark transmission on the $60^{th}$ second iteration of the loop. In the actual microcontroller code, the detection of 60 seconds is done by a timer, and the CPU is notified through an interrupt. However, in the m-code, to actually begin a transmission properly, the transmission must be made to occur on the next iteration ($61^{st}$ second) of the loop.

This sort of timing discrepancy also affects the order in which the m-code is written. For a brief example, since the interrupts in the microcontroller have different priorities relative to each other, the programmer must make sure to write the m-code in such a way that higher priority interrupts are "executed" before lower priority ones. A concrete instance of this is that the comparator has a higher priority interrupt than the one-minute timer, so the comparator's m-code counterpart must be evaluated before the timer's m-code counter part.

Another challenge that MATLAB poses is its own execution time. To simulate both analog and digital signals, a great amount of points need to be calculated and plotted. Generally, points for these signals would be generated by a "for-loop". However, MATLAB was created to work more efficiently with matrices and vectors; it functions terribly slowly with loops. In order to minimize computation time for this study's simulations, which could be on the order of tens of minutes, some slight code alterations were done to cause execution to be on the order of only a few minutes.

There are two highlights other programmers might want to take note of: (1) Although it is possible to use a "for-loop" to calculate the sine function at each time point, MATLAB allows mathematical functions to be executed upon vectors. This means that, given that an X-vector contains the points in time at which evaluation of sine is desired, it

is possible to compute the sine of the whole X-vector and store it in a corresponding Y-vector for plotting. This change from “loop” logic to “matrix” logic cuts down on MATLAB execution time dramatically. (2) Also, for simulation of values of microcontroller variables, which tend to remain constant for a relatively long period of time, it is possible to create vectors in a reduced size to lessen the plotting time required. There is a special “stairs()” function in MATLAB, which connects points with a step-like pattern rather than connecting the points via a straight line; Figure B.1 illustrates the difference. This means that instead of constantly setting a simulated microcontroller variable at each time base interval, as in the situation with the sine function, it is possible only to store values for the time points where the microcontroller variable changes. These specific time points and their values would have to be stored in their own X- and Y-vectors, separate from the X- and Y- vectors of sine, since MATLAB requires equally sized vectors for plotting.

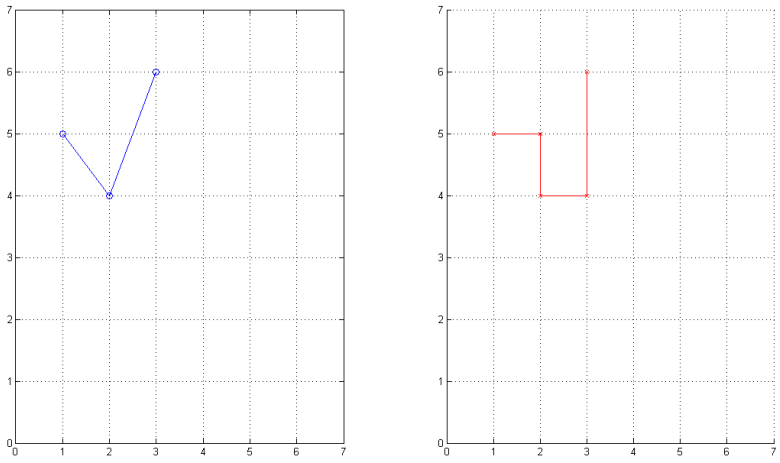



**Figure B.1: (a) MATLAB plot() Function, (b) MATLAB stairs() Function.**

For more information about the MATLAB simulation code in detail, please refer to the comments in the code.

## B.1. Breath Counting Algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Code to simulate counting program
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clear screen and variables
clc;
clear;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% User variables that must be understood for every run
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Time to run simulation in seconds
ucSimulationTimeS = 90;

% Set resp rate
ucRespRateBPM = 31;

% Choose what interval to transmit at
% 10-s transmission = 10240 timer clks
% 15-s transmission = 15360 timer clks
% 30-s transmission = 30720 timer clks
% 1-min transmission = 61440 timer clks
uiTimerClksRequiredForTransmission = 15360;

% Set the voltage of the microcontroller in volts
ucVcc = 3.5;

% Set MCLK freq
fMCLKfreq = 32.768*10^3;  % ~ 32kHz

% Phase shift of resp signal in degrees
ucRespPhaseDegrees = 0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Pre-algorithm calculations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calc resp rate in terms of Hz
ucRespRateHz = ucRespRateBPM/60;

% Calc sec b/w MCLK reads
ucMCLKIntervalS = 1/fMCLKfreq;

% Amplitude of respiratory signal in volts
ucRespAmplitudeV = ucVcc/2;

% Amount shifted in volts to bias the input
ucRespBiasV = ucRespAmplitudeV;

% Phase shift of resp signal in radians
ucRespPhaseRadians = ucRespPhaseDegrees * pi / 180;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Variable initialization
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize the transmission timer
uiTimerClks = 0;

% Variable ucRespRate in MCU
ucRespCount = 0;
vectRespCountx = [0];
vectRespCounty = [0];

% Calculate CPU clk points, this is going to be our plotting time basis
vectCPUClks = [0:ucMCLKIntervalS:ucSimulationTimeS];

% Set up the vector to hold the sine wave that will represent the
% respiratory signal inputted at the comparator input
vectRespx = vectCPUClks;
vectRespy = [];
```

```
% Set up CBOUT vectors
vectCBOUTx = [0]; % Vector to hold the time points that CBOUT changes at
% vectCBOUTy is set later since its inital value depends on the initial
% value of the respiratory signal

% Transmission indicator analogous to actual code; this variable is
% necessary to create the effect of having at least a 1 CPU cycle delay
% between transmission and the update of the respiratory count (since the
% respiratory count is based on the comparator interrupt and has a higher
% priority than the transmission timer interrupt
ucTransmit = 0;
vectTransmitx = [0];
vectTransmity = [0];

% Error plots
vectErrorx = [];
vectErrory = [];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Algorithm Code
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate the sine wave wrt to the CPU clk
vectRespy = ...
    ucRespAmplitudeV * ...
    sin(2*pi*ucRespRateHz*vectCPUClks + ucRespPhaseRadians) + ...
    ucRespBiasV;

% Vector to hold the value of CBOUT;
% The initial value of CBOUT depends on the respiratory signal
if(vectRespy(1) >= 3 * ucVcc / 4)
    vectCBOUTy = [1];
else
    vectCBOUTy = [0];
end

% For every CPU clk cycle, we must determine what affect occur to the
% system; making it wrt the CPU gives us a better idea of what the
% microcontroller is seeing given its "sample rate" used for computation
for CPUClk = 1:size(vectRespy,2)

    % Check if we need to transmit a signal this clock cycle; at most, one
    % CPU cycle will occur between an update of the respiratory rate
    % and the transmission of the respiratory rate
    if(ucTransmit)

        % Calculate the rate that is actually projected by multiplying by
        % the appropriate factor
        if(uiTimerClksRequiredForTransmission == 10240)     % 10 s
            ucRespCount = ucRespCount * 6;
        elseif(uiTimerClksRequiredForTransmission == 15360) % 15 s
            ucRespCount = ucRespCount * 4;
        elseif(uiTimerClksRequiredForTransmission == 30720) % 30 s
            ucRespCount = ucRespCount * 2;
        else                                                % 1 min
            % No multiplication is needed for 1 minute
        end

        % Calculate percent error between what was transmitted and what
        % the actual rate is
        vectErrorx = ...
            [vectErrorx,vectRespx(CPUClk)];
        vectErrory = ...
            [vectErrory,100 *(ucRespCount - ucRespRateBPM) / ...
            ucRespRateBPM];

        % Handle respiratory rate variables and vectors
        ucRespCount = 0;  % Reset the respiratory count on this clock cycle
        vectRespCountx = ...
            [vectRespCountx,vectRespx(CPUClk)];  % Mark time
        vectRespCounty = ...
```

```
            [vectRespCounty,ucRespCount];        % Mark zeroing

        % Handle transmission variables and vectors; transmissions is
        % estimated to take about 4 ms with preliminary testing
        ucTransmit = 0;                          % Reset transmission flag
        vectTransmitx = ...
            [vectTransmitx,vectRespx(CPUClk),...
            vectRespx(CPUClk) + 0.004];          % 4 ms tx...
        vectTransmity = ...
            [vectTransmity,1,0];

    end

    % Check if the state of CBOUT will change; this interrupt has priority
    % over the timer (set transmission FLAG) interrupt
    if(vectRespy(CPUClk) >= 3 * ucVcc / 4 && ...
            vectCBOUTy(size(vectCBOUTy,2)) == 0)

        % We have just had a low-to-high transition and need to note it
        vectCBOUTx = [vectCBOUTx,vectRespx(CPUClk)];
        vectCBOUTy = [vectCBOUTy,1];

        % Update on low-to-high transition...
        ucRespCount = ucRespCount + 1;
        vectRespCountx = [vectRespCountx,vectRespx(CPUClk)];
        vectRespCounty = [vectRespCounty,ucRespCount];

    elseif(vectRespy(CPUClk) <= ucVcc / 4 && ...
            vectCBOUTy(size(vectCBOUTy,2)) == 1)

        % We have just had a high-to-low transition and need to note it
        vectCBOUTx = [vectCBOUTx,vectRespx(CPUClk)];
        vectCBOUTy = [vectCBOUTy,0];

    end

    % Check if we have a timer clk
    if(mod(CPUClk,32) == 0)
       uiTimerClks = uiTimerClks + 1;   % Increase tx interval timer clk
    end

    % Check if a transmission interval has elapsed
    if(uiTimerClks >= uiTimerClksRequiredForTransmission)
        uiTimerClks = 0;  % Reset the timer
        ucTransmit = 1;   % Notify that transmission must occur next cycle
    end
end % End of the for-loop

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot results
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot the respiratory wave, CBOUT, and transmission
subplot(3,1,1);
hold on;
plot(vectCPUClks,vectRespy,'b','LineWidth',2);
stairs(vectCBOUTx,3*vectCBOUTy-5,'r','LineWidth',2);
stairs(vectTransmitx,3*vectTransmity-10,'g','LineWidth',2);
grid on;
title('Respiratory Signal, Comparator Output, & Transmission Duty Cycle');
xlabel(sprintf('Time (s)\n'));
ylabel(sprintf('Respiratory (V)\nComparator & Transmit (No Units)'));
axis([-1,ucSimulationTimeS+10,min(3*vectTransmity-10)-1,max(abs(vectRespy))+1]);

% Plot the value in count
subplot(3,1,2);
stairs(vectRespCountx,vectRespCounty,'m','LineWidth',2);
grid on;
title('Actual Count Value Stored (Not Multiplied for Projection)');
xlabel(sprintf('Time (s)\n'));
ylabel('Count Value (#)');
axis([-1,ucSimulationTimeS+10,0,max(abs(vectRespCounty))+1]);
```

```
% Plot the error between the count transmitted and the actual resp rate
subplot(3,1,3);
bar(vectErrorx,abs(vectErrory),0.5,'y');
text((vectErrorx - 2.5)', ...
    (abs(vectErrory)./2)', ...
    num2str(vectErrory'));
grid on;
title('Absolute Percent Error b/w Transmitted RR & Actual RR');
xlabel('Time (s)');
ylabel('Percent Error (%)');
axis([-1,ucSimulationTimeS+10,0,max(abs(vectErrory))+10]);
```

## B.2. Breath Interval Timing Algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Code to simulate interval program
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Clear screen and workspace and set number format
clc; clear;
format long;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% User variables that must be understood for every run
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Time to run simulation in seconds
ucSimulationTimeS = 90;

% Set resp rate
ucRespRateBPM = 31;

% Choose what interval to transmit at
% 10-s transmission = 10240 timer clks
% 15-s transmission = 15360 timer clks
% 30-s transmission = 30720 timer clks
% 1-min transmission = 61440 timer clks
uiTimerClksRequiredForTransmission = 15360;

% Set the voltage of the microcontroller in volts
ucVcc = 3.5;

% Set MCLK freq
fMCLKfreq = 32.768*10^3;  % ~ 32kHz

% Phase shift of resp signal in degrees
ucRespPhaseDegrees = 0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Pre-algorithm calculations
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calc resp rate in terms of Hz
ucRespRateHz = ucRespRateBPM/60;

% Calc sec b/w MCLK reads
ucMCLKIntervalS = 1/fMCLKfreq;

% Amplitude of respiratory signal in volts
ucRespAmplitudeV = ucVcc/2;

% Amount shifted in volts to bias the input
ucRespBiasV = ucRespAmplitudeV;

% Phase shift of resp signal in radians
ucRespPhaseRadians = ucRespPhaseDegrees * pi / 180;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Variable initialization
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initialize the transmission timer
uiTimerClks = 0;
```

```
% Variable uiRespRate in MCU
ucResetRespCalc = 0;  % Indicator to allow 1 clk cycle delay
ucRespCalc = 0;
ucNumberOfBreathsRecorded = 0;
uiClkOfFirstBreath = 0;
uiClkOfSecondBreath = 0;
vectRespCalcx = [0];
vectRespCalcy = [0];

% Calculate CPU clk points, this is going to be our plotting time basis
vectCPUClks = [0:ucMCLKIntervalS:ucSimulationTimeS];

% Set up the vector to hold the sine wave that will represent the
% respiratory signal inputted at the comparator input
vectRespx = vectCPUClks;
vectRespy =[];

% Set up CBOUT vectors
vectCBOUTx = [0]; % Vector to hold the time points that CBOUT changes at
% vectCBOUTy is set later since its inital value depends on the initial
% value of the respiratory signal

% Transmission indicator analogous to actual code; this variable is
% necessary to create the effect of having at least a 1 CPU cycle delay
% between transmission and the update of the respiratory count (since the
% respiratory count is based on the comparator interrupt and has a higher
% priority than the transmission timer interrupt
ucTransmit = 0;
vectTransmitx = [0];
vectTransmity = [0];

% Error plots
vectErrorx=[];
vectErrory=[];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Algorithm Code
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate the sine wave wrt to the CPU clk
vectRespy = ...
    ucRespAmplitudeV * ...
    sin(2*pi*ucRespRateHz*vectCPUClks + ucRespPhaseRadians) + ...
    ucRespBiasV;

% The initial value of CBOUT depends on the respiratory signal
if(vectRespy(1) >= 3 * ucVcc / 4)
    vectCBOUTy = [1];
else
    vectCBOUTy = [0];
end

% For every CPU clk cycle, we must determine what affect occur to the
% system; making it wrt the CPU gives us a better idea of what the
% microcontroller is seeing given its "sample rate" used for computation
for CPUClk = 1:size(vectRespy,2)

    % Check for the clearing of stuff for delay, since matrix math weird
    if(ucResetRespCalc == 1)
        vectRespCalcx = ...
            [vectRespCalcx,vectRespx(CPUClk)];  % Mark time
        vectRespCalcy = ...
            [vectRespCalcy,0];                  % Zero out next cycle
        ucResetRespCalc = 0;
    end

    % Check if we need to transmit a signal this clock cycle; at most, one
    % CPU cycle will occur between an update of the respiratory rate
    % and the transmission of the respiratory rate
    if(ucTransmit)
```

```
    % Check if there are enough breaths to calculate the rate
    if(ucNumberOfBreathsRecorded == 2)

        % Handle respiratory rate variables and vectors; we also have
        % to be aware of overlap/overflow of the timer
        if(uiClkOfSecondBreath > uiClkOfFirstBreath)
            ucRespCalc = floor(61440 / ...
                (uiClkOfSecondBreath - uiClkOfFirstBreath));
        else
            ucRespCalc = floor(61440 / ...
                (uiTimerClksRequiredForTransmission - 1 - ...
                uiClkOfFirstBreath + uiClkOfSecondBreath));
        end

    else

        % Not enough breaths caught or too many
        ucRespCalc = 0;

    end

    ucNumberOfBreathsRecorded = 0;  % Reset number of breaths recorded
    vectRespCalcx = [vectRespCalcx,vectRespx(CPUClk)];  % Mark time
    vectRespCalcy = [vectRespCalcy,ucRespCalc];         % Update calc

    % Calculate percent error between what was transmitted and what
    % the actual rate is before we clear the respiratory calculation
    vectErrorx = ...
        [vectErrorx,vectRespx(CPUClk)];
    vectErrory = ...
        [vectErrory,100 * (ucRespCalc - ucRespRateBPM) / ...
        ucRespRateBPM];
    ucRespCalc = 0;

    % Reset in the actual code, but commented out so we
    % can see the results here, since the duration that value is
    % actually kept in the code is too small to see
    ucResetRespCalc = 1;

    % Handle transmission variables and vectors; transmissions is
    % estimated to take about 4 ms with preliminary testing
    ucTransmit = 0;                         % Reset transmission flag
    vectTransmitx = ...
        [vectTransmitx,vectRespx(CPUClk),...
        vectRespx(CPUClk) + 0.004];         % 4 ms tx...
    vectTransmity = ...
        [vectTransmity,1,0];

end

% Check if the state of CBOUT will change; this interrupt has priority
% over the timer (set transmission FLAG) interrupt
if(vectRespy(CPUClk) >= 3 * ucVcc / 4 && ...
        vectCBOUTy(size(vectCBOUTy,2)) == 0)

    % We have just had a low-to-high transition and need to note it
    vectCBOUTx = [vectCBOUTx,vectRespx(CPUClk)];
    vectCBOUTy = [vectCBOUTy,1];

    % Update on low-to-high transition and only if we don't have enough
    % measurements (i.e. 2), since in the actual code, we turn off the
    % interrupt after 2 and only turn it back on after transmission
    if(ucNumberOfBreathsRecorded < 2)

        % We don't enough measurements and need to capture more;
        % Let us check which breath we are trying to capture
        if(ucNumberOfBreathsRecorded == 0)

            % We are recording the first breath
            uiClkOfFirstBreath = uiTimerClks;
        else
```

```
                % We are recording the second breath
                uiClkOfSecondBreath = uiTimerClks;
            end

            % Update the number of breaths recorded
            ucNumberOfBreathsRecorded = ucNumberOfBreathsRecorded + 1;
        end

    elseif(vectRespy(CPUClk) <= ucVcc / 4 && ...
            vectCBOUTy(size(vectCBOUTy,2)) == 1)

        % We have just had a high-to-low transition and need to note it
        vectCBOUTx = [vectCBOUTx,vectRespx(CPUClk)];
        vectCBOUTy = [vectCBOUTy,0];

    end

    % Check if we have a timer clk
    if(mod(CPUClk,32) == 0)
       uiTimerClks = uiTimerClks + 1;   % Increase tx interval timer clk
    end

    % Check if a transmission interval has elapsed
    if(uiTimerClks >= uiTimerClksRequiredForTransmission)
        uiTimerClks = 0;  % Reset the timer
        ucTransmit = 1;   % Notify that a transmission must occur next cycle
    end
end % End of the for-loop

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot results
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot the respiratory wave, CBOUT, and transmission
subplot(3,1,1);
hold on;
plot(vectCPUClks,vectRespy,'b','LineWidth',2);
stairs(vectCBOUTx,3*vectCBOUTy-5,'r','LineWidth',2);
stairs(vectTransmitx,3*vectTransmity-10,'g','LineWidth',2);
grid on;
title('Respiratory Signal, Comparator Output, & Transmission Duty Cycle');
xlabel(sprintf('Time (s)\n'));
ylabel(sprintf('Respiratory (V)\nComparator & Transmit (No Units)'));
axis([-1,ucSimulationTimeS+10,min(3*vectTransmity-10)-1,max(abs(vectRespy))+1]);

% Plot the value calculated for respiratory rate
subplot(3,1,2);
stairs(vectRespCalcx,vectRespCalcy,'m','LineWidth',2);
grid on;
title('Actual RR Value Stored After Calculation');
xlabel(sprintf('Time (s)\n'));
ylabel('Calculated RR (br/min)');
axis([-1,ucSimulationTimeS+10,0,max(abs(vectRespCalcy))+1]);

% Plot the error b/w the calculation transmitted and the actual resp rate
subplot(3,1,3);
bar(vectErrorx,abs(vectErrory),0.5,'y');
text((vectErrorx - 2.5)', ...
    (abs(vectErrory)./2)', ...
    num2str(vectErrory'));
grid on;
title('Absolute Percent Error b/w Transmitted RR & Actual RR');
xlabel('Time (s)');
ylabel('Percent Error (%)');
axis([-1,ucSimulationTimeS+10,0,max(abs(vectErrory))+10]);
```

# Appendix C: Power Consumption Calculations

The power approximations and calculations mentioned in Chapter 4 are detailed here. The first sub-appendix discusses what geometric approximations were used to determine the average voltage over the shunt resistor, and the second sub-appendix contains the MATLAB program created for calculation automation in case of an approximation change. Specific values for the voltage, time, and resistor approximations are not stated in the first sub-appendix, which discusses the shapes used in general, but are available in the program code in the second sub-appendix.

## C.1. Rationale

The following figures were used to define the geometric approximations for the average voltage estimations in Chapter 4; they illustrate the voltage reaction to a 1.5-Hz (i.e., 90 br/min) sinusoidal input over a 2.2 kΩ ±5% resistor with a RF transmission occurring every 10 seconds. Each feature within this 10-s period has its area estimated in volts-seconds, which is distributed over 10 seconds to obtain the average voltage contributed by that feature per period.

Figure C.1 depicts the voltage of an RF transmission estimated as two rectangles: one about 90-V high and 1-ms wide and another about 180-mV high and 2.6-ms wide. The areas of these rectangles were summed and divided over 10 seconds to obtain the average voltage over the shunt resistor. Usually, to acquire the total average voltage, the rest of the signal outside of the two rectangles would also need to be distributed. However, since their voltage is completely overshadowed by the transmission, they can be relatively estimated as having zero contribution to the RF power consumption. Therefore, when calculating the power consumption of the data acquisition algorithms with RF transmission enabled, it is acceptable to gain the result by adding this RF power estimation to the power required by the algorithms with RF transmission disabled.

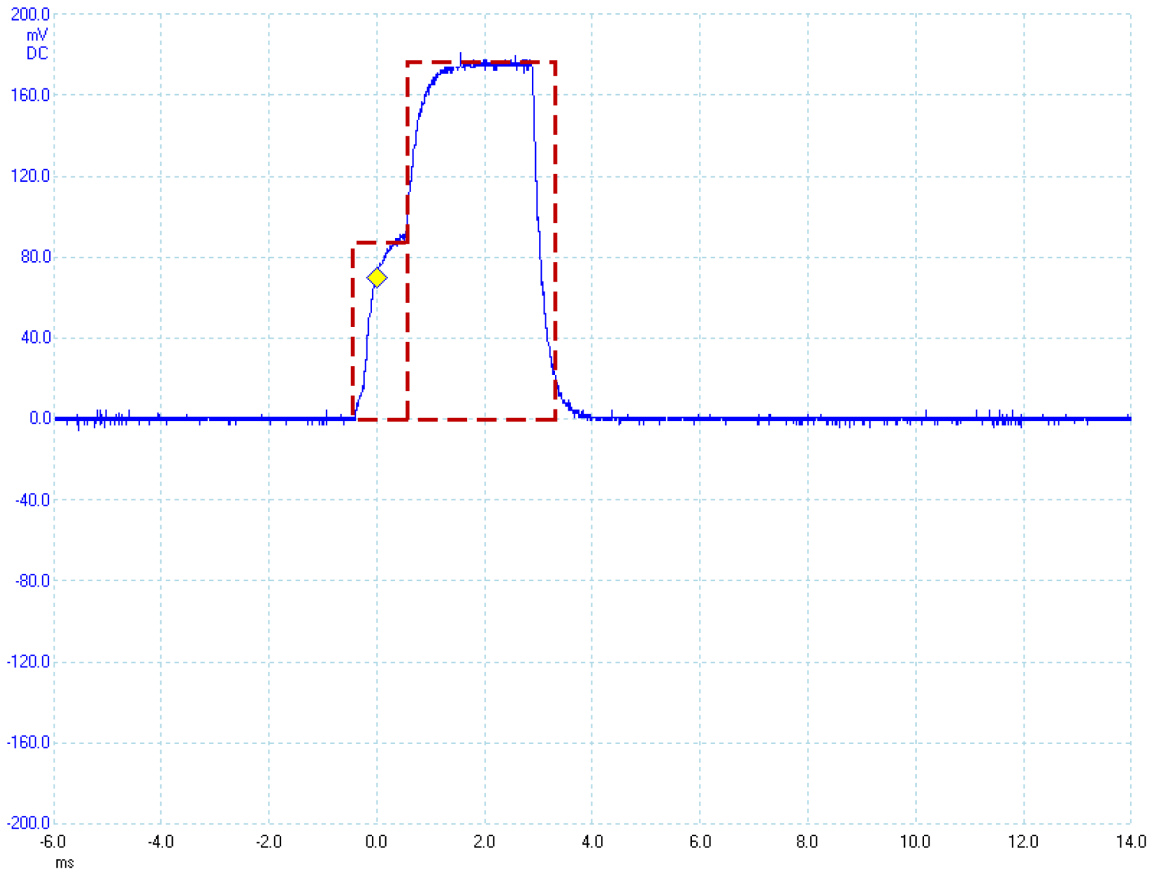



**Figure C.1: RF Transmission Geometric Power Estimation.**

Figure C.2 shows the triangular and rectangular approximations used for the breath counting algorithm. Each breath count creates a smaller triangles and each multiplication creates a larger one. The span of each rectangle is 10 seconds. The areas under several count triangles (i.e., the number depends on the RR) and one multiplication triangle with respect to the top of the rectangle were summed, divided over 10 seconds, and then added to the 39-mV height of the rectangle. The result is the average voltage over the shunt resistor by the breath counting algorithm.

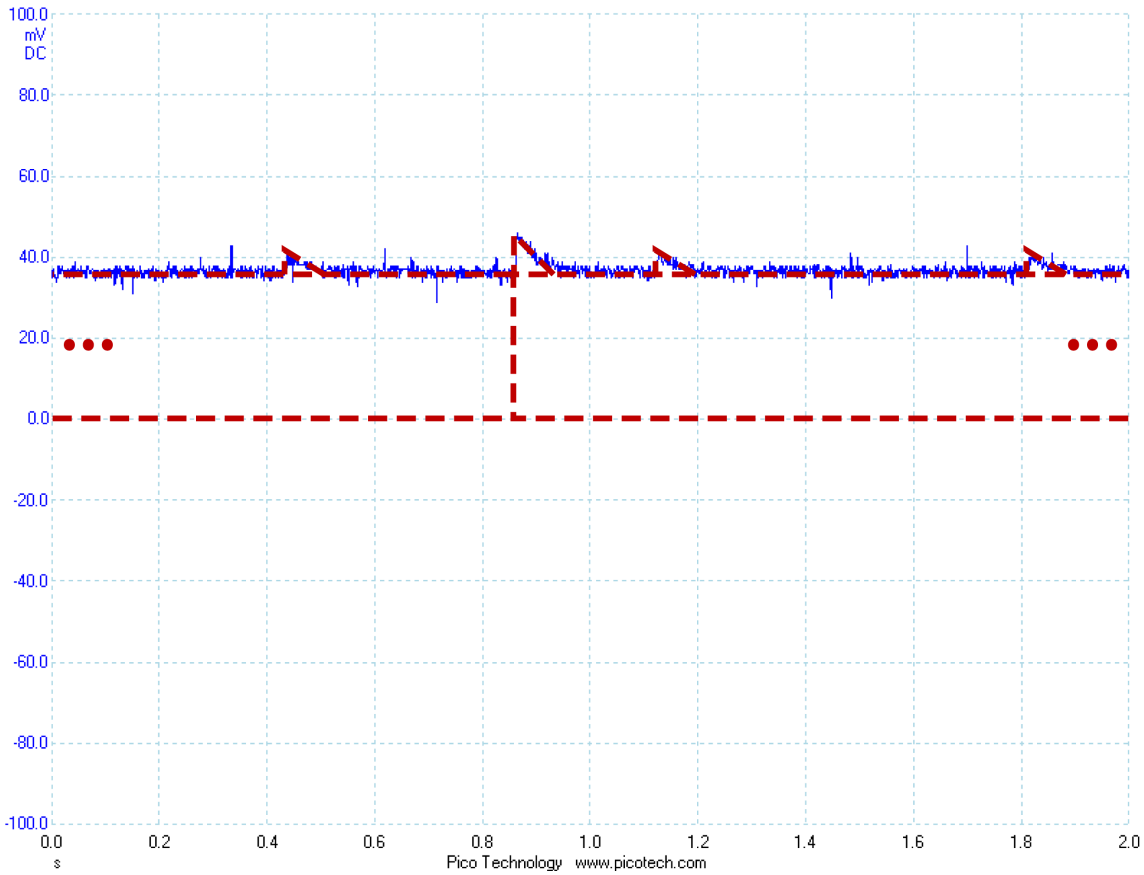



**Figure C.2: Breath Counting Algorithm (No Transmission) Geometric Power Estimation.**

Figure C.3 pictures the average voltage of the breath interval timing algorithm without RF transmission and with the comparator constantly on. The geometric estimation is similar to the breath counting algorithm; however there are only two smaller triangles versus several. These two triangles indicate when a breath time point is captured and stored, and the larger one represents the single division per 10-s time period. The areas of these triangles relative to the top of the 10-s rectangle were again summed, divided by 10 seconds, and added to the 39-mV height of the rectangle to acquire the average voltage over the shunt resistor.

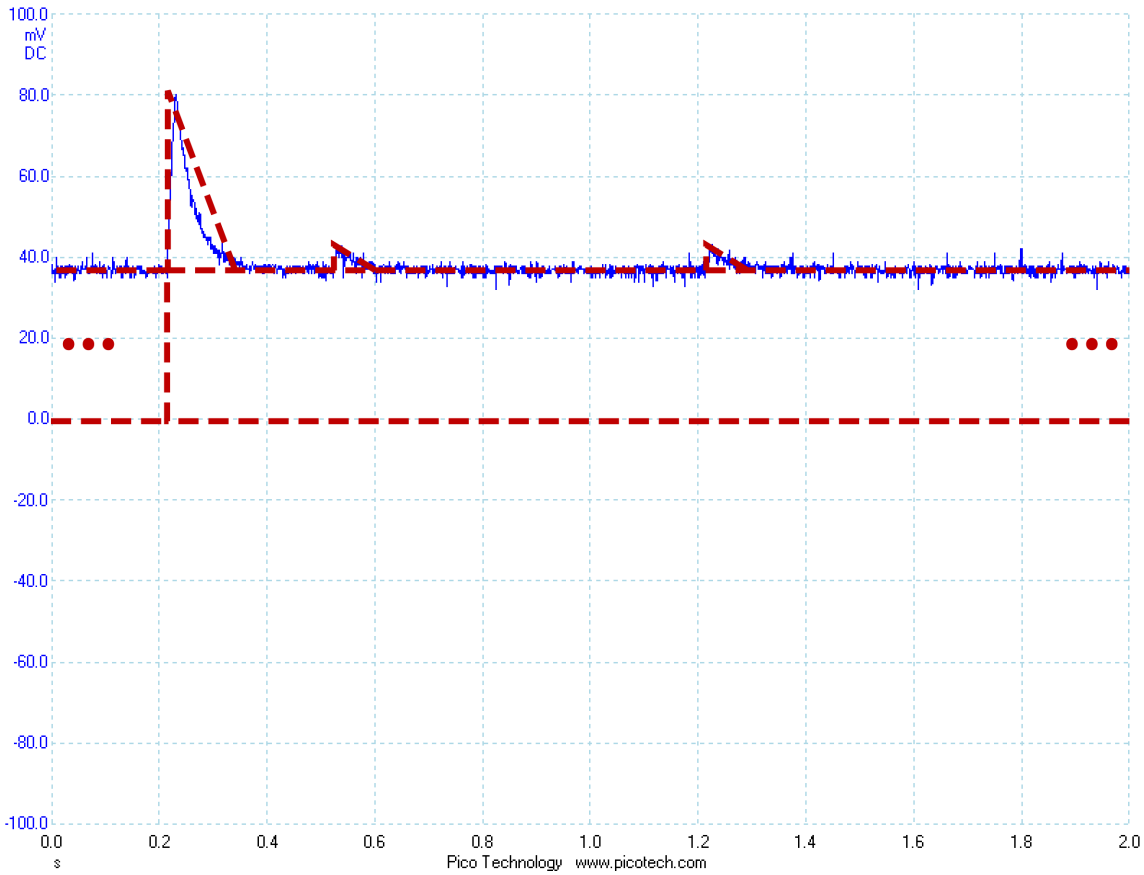



**Figure C.3: Breath Interval Timing Algorithm (No Transmission, Comparator Constantly On) Geometric Power Estimation.**

Figure C.4 exhibits when comparator toggling is enabled in the breath interval timing algorithm. The power consumption estimation becomes slightly more difficult since it is not easy to predict how long the comparator will remain. This alters the width of the smaller rectangle that indicates the time that the comparator is on. To cope with this dependence, a minimum and maximum value range was determined by assuming the worst case of 1 br/ 10 s and best case of 2 br/ 10 s. Also, the effect of the comparator turning off affects the triangle of the second capture, making it incredibly tiny relative to the voltage of when the comparator is on. Thus, instead of three triangles, there are now four: the large on for the division, the two captures triangles, and the one next to the rectangle produced when the comparator is turned on. Average voltage over the shunt resistor is evaluated by adding all the triangular areas and the comparator rectangle, dividing the result by 10 seconds, and summing it with the height of the 10-mV, 10-s rectangle.

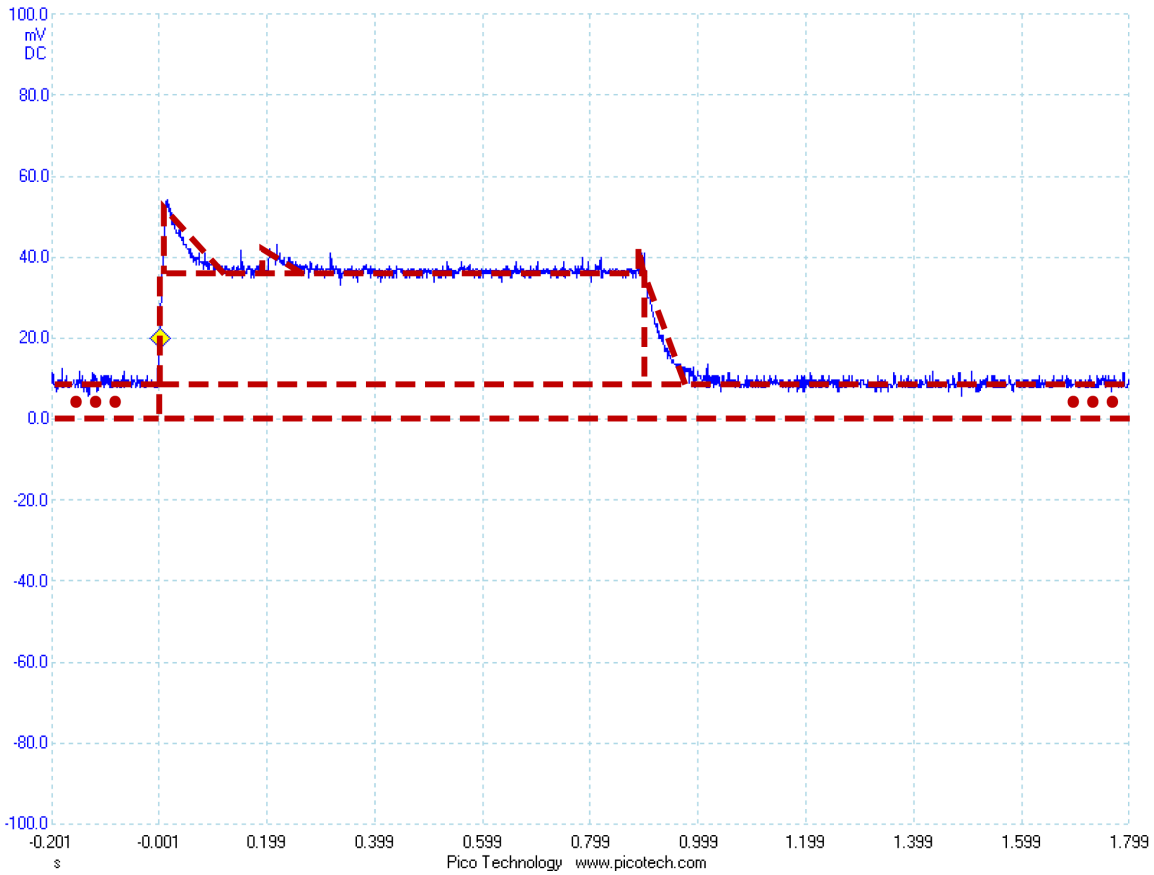



**Figure C.4: Breath Interval Timing Algorithm (No Transmission, Comparator Toggled) Geometric Power Estimation.**

Although the figures in this appendix are for the specific measurements regarding the 90-br/min, 10-s RF transmission case, the calculations for other respiratory rates and intervals can be done similarly and thus were automated by the program available in the next sub-appendix. It is only required that these feature shapes be approximated for a single case, because they only occur once per period (i.e. RF transmission), and their basic shape does not change when the period or respiratory rate changes.

## C.2. Power Calculation Program

```
%*************************************************************************
%
% Power Consumption Estimates Based on 10-s Transmission @ 1.5 Hz
%
% VERSION: powerest.m
% CREATED: October 4, 2011
% REVISED: December 6, 2011
% REQUIRED: Reference measurement figures in thesis appendix.
%
% COMMENTS:
% This program creates power estimation plots based on 10-s
% transmission interval figures measured at 1.5 Hz (i.e., 90 br/min).
% Geometrical estimations of power in each figure are used.
% The figures are of the voltage over a resistor in series with
% the EM430F6137RF900 evaluation board were recorded using
% the PicoScope 2140 PC Oscilloscope. P = IV, all units are
% standard W,A,V unless otherwise noted.
%
% There will be 8 plots and 8 calculations generated:
%   - Plots
%       - power vs. time b/w tx (assume: 14 br/min)
%           - txTimeFigure
%           - countTimeFigure
%           - intervalTimeFigure
%           - toggleTimeFigure
%           - comboTimeFigure
%       - power vs. RR (assume: 10 s)
%           - countRRFigure
%           - toggleRRFigure
%           - comboRRFigure
%   - Calculations (assume: 14 br/min; time b/w tx: 10/15/30s, 1 min)
%       - tx power only
%       - algs w/ tx
%           - breath counting
%           - breath interval timing w/ comparator constantly on
%           - breath interval timing w/ comparator toggling max
%           - breath interval timing w/ comparator toggling min
%       - algs w/o tx
%           - breath counting
%           - breath interval timing w/ comparator constantly on
%           - breath interval timing w/ comparator toggling max
%           - breath interval timing w/ comparator toggling min
%
% The RR range we will plot over depends on the algorithm's valid range.
% Ideally, we'd like to plot over 2-90 br/min. However, for a 10-s interval
% b/w transmissions, the count algorithm's current implementation is
% limited to only 6-90 br/min and the timing algorithms is limited to 12-90
% br/min. Therefore, these are the ranges that were plotted.
%
%*************************************************************************

% Clear up the screen and set number formats
clc; clear; format long;

% Set singular global constants
vcc = 3.5;                      % Volts (v) supplied to eval board
txInterval = 10;                % # of seconds (s) b/w transmissions (tx)
respRate = 14;                  % Respiratory rate (RR) in br/min
respFreq = respRate/60;         % Respiratory rate in hertz (Hz)

% Set range global constants
txIntervalRange = [10:1:60];    % Range: 10 s to 1 min for time b/w tx
respRateCountRange = ...        % Valid range: 6 to 90 br/min for RR
    [6:1:90];
respFreqCountRange = ...        % Valid range: ~0.1 to 1.5 Hz
    respRateCountRange ./ 60;
respRateIntervalRange = ...     % Valid range: 12 to 90 br/min for RR
    [12:1:90];
```

```
respFreqIntervalRange = ...     % Valid range: ~0.2 to 1.5 Hz
    respRateIntervalRange ./ 60;

%**************************************************************************
% Calculate the average power for RF transmission; Since all other
% signals look RELATIVELY negligible, then it's okay to assume they are
% and the average voltage is equal to the area under the curve divided
% by the total time. We are essentially flattening the curve.
%
%  ^       |------|             ^
%  |       |      |             |
%  |       |      |         =>  | ------------------------
%  | ------|      |-------      |
%  |----------------------      |------------------------
%
% Since, there are no dependencies on RR, we do not plot a correlating
% figure.
%**************************************************************************

% The transmission curve can be estimated by two rectangles: one small one
% and one big one. These can be added and then divided by the total time.
heightTxSmall = 90E-3;      % Height of small rect (v)
heightTxBig = 180E-3;       % Height of big rect (v)
widthTxSmall = 1.0E-3;      % Width of small rect (s)
widthTxBig = 2.65E-3;       % Width of big rect (s)
shuntTxResistor = 10;       % Shunt resistor value (ohms)

% Calculate the average transmission voltage over the chosen time interval
% over the resistor, by adding the small rectangle area to the big
% rectangle area and then dividing by the time interval (i.e., average
% voltage contributed by tx over the time interval). Then calculate average
% current and multiply by the voltage supplied to the EM430F6137RF900
% (i.e., vcc - v over the resistor) to get the average power estimation.
averageTxVoltageTimeRange = ...
    (heightTxSmall*widthTxSmall + heightTxBig*widthTxBig) ...
    ./ txIntervalRange;
averageTxCurrentTimeRange = ...
    averageTxVoltageTimeRange ./ shuntTxResistor;
averageTxPowerTimeRange = ...
    averageTxCurrentTimeRange .* ...
    (vcc - averageTxVoltageTimeRange);
averageTxPowerMicroWattsTimeRange = ...
    averageTxPowerTimeRange .* 1E6;

% Plot Power (uW) vs. Tx Interval (s)
figure;
grid on;
hold on;
title('Tx Power Consumption wrt Time b/w Tx @ 14 br/min');
xlabel('Time b/w Tx (s)');
ylabel('Power (uW)');
plot(txIntervalRange,averageTxPowerMicroWattsTimeRange,'m^-');

%**************************************************************************
% Calculate the average power consumption of the count algorithm. Each
% count action can be estimated as a small triangle, while the
% multiplication for rate projection is a slightly larger triangle. These
% triangles can be thought of as sitting on a rectangle (i.e., constant
% power consumption throughout the entire interval). This means that
% we can simply add the height of this rectangle to the distributed power
% consumption estimated for the two triangles sitting on top of it. This
% calculation is frequency dependent, since more counts per interval will
% increase the power consumption. In summary, the geometric shapes per
% interval are:
%   - several small triangles for count actions
%   - one big triangle for multiplication (assuming it is always carried
%     out; it won't be if we are doing a 1 min interval or may be lessened
%     if we do a bit shift)
%   - one rectangle for lowest voltage
%**************************************************************************
heightCountRectangle = 39E-3;       % Height of rect (v)
```

```
% Absolute heights of the triangles from 0 V
heightCountSmallAbsolute = 41E-3;   % Height of small tri (v)
heightCountBigAbsolute = 48E-3;     % Height of big tri (v)

% Relative heights of the triangles wrt top of rectangle
heightCountSmallRelative = ...      % Height of small tri (v)
    heightCountSmallAbsolute - heightCountRectangle;
heightCountBigRelative = ...        % Height of big tri (v)
    heightCountBigAbsolute - heightCountRectangle;

% Widths of triangle bases wrt top of rectangle (i.e., not 0 V)
widthCountSmall = 0.1E-3;           % Width of small tri (s)
widthCountBig = 0.14E-3;            % Width of big tri (s)
shuntCountResistor = 2.2E3;         % Shunt resistor value (ohms)

% Calculate the average count algorithm voltage over the chosen time
% interval over the resistor, by first adding several small triangle areas
% to the one big triangle area and then dividing by the time interval
% (i.e., average voltage contributed by counting and multiplying for rate
% projection over the time interval). In order to determine the number of
% small triangles, you need to know the frequency of the signal; simply
% multiply the time interval by the frequency to get the number of small
% triangles (i.e., max amount of breaths that can occur in the allotted
% time interval). We need to use the ceiling function since we are only
% tracking the start of a breath versus the whole breath (i.e., worst case
% power consumption for a given tx interval).
%
% After that, you need to take into account the rectangle the triangles are
% sitting on, so you just add the voltage height of it to the triangle
% calculations. Then calculate average current and multiply by the voltage
% supplied to the EM430F6137RF900 (i.e., vcc - v over the resistor) to get
% the average power estimation.

% First, create plot values wrt respiratory rate
numberOfCountsRRRange = ceil(respFreqCountRange .* txInterval);
averageCountVoltageRRRange = ...
    (numberOfCountsRRRange .* ...
    (heightCountSmallRelative * widthCountSmall / 2) + ...
    (heightCountBigRelative * widthCountBig / 2)) ...
    ./txInterval + ...
    heightCountRectangle;
averageCountCurrentRRRange = ...
    averageCountVoltageRRRange ./ shuntCountResistor;
averageCountPowerRRRange = ...
    averageCountCurrentRRRange .* ...
    (vcc - averageCountVoltageRRRange);
averageCountPowerMicroWattsRRRange = ...
    averageCountPowerRRRange * 1E6;

% Plot Power (uW) vs. RR (br/min)
figure;
grid on;
hold on;
title('Count Algorithm Power Consumption wrt RR @ 10-s Tx');
xlabel('Respiratory Rate (br/min)');
ylabel('Power (uW)');
plot(respRateCountRange,averageCountPowerMicroWattsRRRange,'ro-');

% Second, create plot values wrt the time b/w tx
numberOfCountsTimeRange = ceil(respFreq .* txIntervalRange);
averageCountVoltageTimeRange = ...
    (numberOfCountsTimeRange .* ...
    (heightCountSmallRelative * widthCountSmall / 2) + ...
    (heightCountBigRelative * widthCountBig / 2)) ...
    ./ txIntervalRange + ...
    heightCountRectangle;
averageCountCurrentTimeRange = ...
    averageCountVoltageTimeRange ./ shuntCountResistor;
averageCountPowerTimeRange = ...
    averageCountCurrentTimeRange .* ...
```

```
    (vcc - averageCountVoltageTimeRange);
averageCountPowerMicroWattsTimeRange = ...
    averageCountPowerTimeRange * 1E6;

% Plot Power (uW) vs. Time b/w Tx (s)
figure;
grid on;
hold on;
title('Count Algorithm Power Consumption wrt Time b/w Tx @ 14 br/min');
xlabel('Time b/w Tx (s)');
ylabel('Power (uW)');
plot(txIntervalRange,averageCountPowerMicroWattsTimeRange,'ro-');

%*************************************************************************
% Calculate the average power of the interval algorithm without toggling
% the comparator. This is similar to the count algorithm in which there are
% triangles sitting on top of a rectangle that expands across the whole
% time interval. However, instead of having several small triangles, there
% are only two to mark the capture of the start of two consecutive breaths.
% There is still a large triangle to estimate the division. In summary, we
% have:
%   - two small triangles for marking two consecutive breaths
%   - one big triangle for the division
%   - one rectangle for the lowest voltage
%*************************************************************************
heightIntervalRectangle = 39E-3;       % Height of rect (v)

% Absolute heights of the triangles from 0 V
heightIntervalSmallAbsolute = 42E-3;   % Height of small tri (v)
heightIntervalBigAbsolute = 80E-3;     % Height of big tri (v)

% Relative heights of the triangles wrt top of rectangle
heightIntervalSmallRelative = ...      % Height of small tri (v)
    heightIntervalSmallAbsolute - heightIntervalRectangle;
heightIntervalBigRelative = ...        % Height of big tri (v)
    heightIntervalBigAbsolute - heightIntervalRectangle;

% Widths of triangle bases wrt top of rectangle (i.e., not 0 V)
widthIntervalSmall = 0.1E-3;           % Width of small tri (s)
widthIntervalBig = 0.14E-3;            % Width of big tri (s)
shuntIntervalResistor = 2.2E3;         % Shunt resistor value in ohms

% Calculate the average interval algorithm voltage over the time interval
% chosen over the resistor, by first adding the two small triangle areas to
% the one big triangle area and then dividing by the time interval (i.e.,
% average voltage contributed by two captures and a division over the time
% interval). Note: Adding two triangles of the same height and base width
% equates to one rectangle of the same height and width. This is the
% calculation used below.
%
% After that, you need to take into account the rectangle the triangles are
% sitting on, so you just add the voltage height of it to the triangle
% calculations. Then calculate average current and multiply by the voltage
% supplied to the EM430F6137RF900 (i.e., vcc - v over the resistor) to get
% the average power estimation.
%
% Since, there are no dependencies on RR, we do not plot a correlating
% figure.
averageIntervalVoltageTimeRange = ...
    ((heightIntervalSmallRelative * widthIntervalSmall) + ...
    (heightIntervalBigRelative * widthIntervalBig / 2)) ...
    ./ txIntervalRange + ...
    heightIntervalRectangle;
averageIntervalCurrentTimeRange = ...
    averageIntervalVoltageTimeRange ./ shuntIntervalResistor;
averageIntervalPowerTimeRange = ...
    averageIntervalCurrentTimeRange .* ...
    (vcc - averageIntervalVoltageTimeRange);
averageIntervalPowerMicroWattsTimeRange = ...
    averageIntervalPowerTimeRange * 1E6;
```

```
% Plot Power (uW) vs. Time b/w Tx (s)
figure;
grid on;
hold on;
title(['Interval Algorithm Power Consumption w/o Toggle ' ...
    'wrt Time b/w Tx @ 14 br/min']);
xlabel('Time b/w Tx (s)');
ylabel('Power (uW)');
plot(txIntervalRange,averageIntervalPowerMicroWattsTimeRange,'bd-');

%**************************************************************************
% Calculate the average power of the interval algorithm with toggling
% the comparator; this one is trickier than the interval algorithm w/o
% toggling, since there are several features for consideration. In this
% scheme there are two rectangles and four triangles. From bottom to top,
% we have a large rectangle with a smaller rectangle plus a triangle
% sitting on of it. On the smaller rectangle are three other triangles: one
% big one and two other smaller ones. The bottom rectangle represents the
% when the comparator is completely turned off. The smaller rectangle and
% triangle on top of that represents when the comparator is on. The width
% of this rectangle-triangle combo is variable depending on the respiratory
% rate. The big triangle on the very top illustrates the major division
% that occurs, and the smaller triangles illustrate the two timer captures.
%
% In summary, the shapes used for estimation are:
%   - one big triangle for the division
%   - two smaller triangles for the two captures
%   - one small rectangle & triangle for the comparator-on time
%   - one big rectangle for the lowest voltage
%
% The tricky part is that the time the comparator is on is variable, it
% can be as long as two breaths ("first" breath happened just before the
% "previous" transmission) or as short as one breath ("first" breath
% happened just after the "previous" transmission). We account for this by
% altering the width of the comparator rectangle. Technically, we should
% also take into account the comparator triangle; however, this feature is
% constant. Thus, we don't vary this since we are only estimating the range
% of the toggling power consumption. Leaving the triangle as a permanent
% feature instead of variable should only slightly overestimate the max
% power consumption.
%**************************************************************************
heightToggleCompOffRect = 10E-3;    % Height of comp-off rect (v)
shuntToggleResistor = 2.2E3;        % Shunt resistor value (ohms)

% Absolute heights of triangular features and the comp-on rectangle wrt 0 v
heightToggleDivisionTriAbsolute = 53E-3;    % Height of division tri (v)
heightToggleCapture1TriAbsolute = 42E-3;    % Height of Capture 1 tri (v)
heightToggleCapture2TriAbsolute = 42E-3;    % Height of Capture 2 tri (v)
heightToggleCompOnTriAbsolute = 39E-3;      % Height of comp-on tri (v)
heightToggleCompOnRectAbsolute = 39E-3;     % Height of comp-on rect (v)

% Relative heights of triangular features and the comparator rectangle wrt
% to top of comp-off rectangle
heightToggleDivisionTriRelative = ...       % Height of division tri (v)
    heightToggleDivisionTriAbsolute - ...
    heightToggleCompOnRectAbsolute;
heightToggleCapture1TriRelative = ...       % Height of capture 1 tri (v)
    heightToggleCapture1TriAbsolute - ...
    heightToggleCompOnRectAbsolute;
heightToggleCapture2TriRelative = ...       % Height of capture 2 tri (v)
    heightToggleCapture2TriAbsolute - ...
    heightToggleCompOnRectAbsolute;
heightToggleCompOnTriRelative = ...         % Height of comp-on tri (v)
    heightToggleCompOnTriAbsolute  - ...
    heightToggleCompOffRect;
heightToggleCompOnRectRelative =  ...       % Height of comp-on rect (v)
    heightToggleCompOnRectAbsolute - ...
    heightToggleCompOffRect;

% Widths of triangular features and the comparator rectangle
widthToggleDivisionTri = 0.12E-3;   % Width of division tri (s)
```

```
widthToggleCapture1Tri = 0.1E-3;    % Width of capture 1 tri (s)
widthToggleCapture2Tri = 0.02E-3;   % Width of capture 2 tri (s)
widthToggleCompOnTri = 0.1E-3;      % Width of comp-on tri (s)

% Calculate the average interval algorithm (with toggling) voltage over the
% time interval chosen over the resistor, by first adding the four
% triangles and small rectangle and then dividing by the time interval
% (i.e., average voltage contributed by two captures, a division, and the
% comparator over the time interval).
%
% After that, you need to take into account the rectangle that everything
% is sitting on, so you just add the voltage height of it to the previous
% calculations. Then calculate average current and multiply by the voltage
% supplied to the EM430F6137RF900 (i.e., vcc - v over the resistor) to get
% the average power estimation.
%
% As opposed to the case when the comparator is constantly on, the power
% estimation of the toggle method varies with both the respiratory rate and
% the transmission interval. Thus, two plots were created.

% First, create plot values wrt respiratory rate
widthToggleCompOnRectMinRRRange = ...
    1 ./ respFreqIntervalRange;
widthToggleCompOnRectMaxRRRange = ...
    2 ./ respFreqIntervalRange;

% Calculate the minimum values wrt respiratory rate
averageToggleVoltageMinRRRange = ...
    ((heightToggleDivisionTriRelative * widthToggleDivisionTri / 2) + ...
    (heightToggleCapture1TriAbsolute * widthToggleCapture1Tri / 2) + ...
    (heightToggleCapture2TriAbsolute * widthToggleCapture2Tri / 2) + ...
    (heightToggleCompOnTriRelative * widthToggleCompOnTri / 2) + ...
    (heightToggleCompOnRectRelative .* ...
    widthToggleCompOnRectMinRRRange)) ...
     ./ txInterval + ...
     heightToggleCompOffRect;
averageToggleCurrentMinRRRange = ...
    averageToggleVoltageMinRRRange ./ shuntToggleResistor;
averageTogglePowerMinRRRange = ...
    averageToggleCurrentMinRRRange .* ...
    (vcc - averageToggleVoltageMinRRRange);
averageTogglePowerMicroWattsMinRRRange = ...
    averageTogglePowerMinRRRange .* 1E6;

% Calculate the maximum values wrt respiratory rate
averageToggleVoltageMaxRRRange = ...
    ((heightToggleDivisionTriRelative * widthToggleDivisionTri / 2) + ...
    (heightToggleCapture1TriAbsolute * widthToggleCapture1Tri / 2) + ...
    (heightToggleCapture2TriAbsolute * widthToggleCapture2Tri / 2) + ...
    (heightToggleCompOnTriRelative * widthToggleCompOnTri / 2) + ...
    (heightToggleCompOnRectRelative .* ...
    widthToggleCompOnRectMaxRRRange)) ...
    ./ txInterval + ...
    heightToggleCompOffRect;
averageToggleCurrentMaxRRRange = ...
    averageToggleVoltageMaxRRRange ./ shuntToggleResistor;
averageTogglePowerMaxRRRange = ...
    averageToggleCurrentMaxRRRange .* ...
    (vcc - averageToggleVoltageMaxRRRange);
averageTogglePowerMicroWattsMaxRRRange = ...
    averageTogglePowerMaxRRRange .* 1E6;

% Plot Power (uW) vs. RR (br/min)
figure;
grid on;
hold on;
title('Interval Algorithm Power Consumption w/Toggle wrt RR @ 10-s Tx');
xlabel('Respiratory Rate (br/min)');
ylabel('Power (uW)');

% We sort of need to cheat in order to get the desired area coloring
```

```
% effect. We are just overlaying that shaded max area plot with the min
% area plot in order to show the difference.
area(respRateIntervalRange,...
    averageTogglePowerMicroWattsMaxRRRange,'facecolor','y');
area(respRateIntervalRange,...
    averageTogglePowerMicroWattsMinRRRange,'facecolor','w');

% Show the grid on top
set(gca,'Layer','top');

% Plot the min and max values on top of everything else
plot(respRateIntervalRange,averageTogglePowerMicroWattsMinRRRange,'gs-');
plot(respRateIntervalRange,averageTogglePowerMicroWattsMaxRRRange,'gs-');

% Set the x-axis limit to hide the side lines from the area plots
xlim([min(respRateIntervalRange) max(respRateIntervalRange)]);

% Second, create plot values wrt the time b/w tx
widthToggleCompOnRectMinTime = ...
    1 ./ respFreq; % Comp rectangle height
widthToggleCompOnRectMaxTime = ...
    2 ./ respFreq;

% Calculate the minimum values wrt the time b/w tx
averageToggleVoltageMinTimeRange = ...
    ((heightToggleDivisionTriRelative * widthToggleDivisionTri / 2) + ...
    (heightToggleCapture1TriAbsolute * widthToggleCapture1Tri / 2) + ...
    (heightToggleCapture2TriAbsolute * widthToggleCapture2Tri / 2) + ...
    (heightToggleCompOnTriRelative * widthToggleCompOnTri / 2) + ...
    (heightToggleCompOnRectRelative .* widthToggleCompOnRectMinTime)) ...
     ./ txIntervalRange + ...
     heightToggleCompOffRect;
averageToggleCurrentMinTimeRange = ...
    averageToggleVoltageMinTimeRange ./ shuntToggleResistor;
averageTogglePowerMinTimeRange = ...
    averageToggleCurrentMinTimeRange .* ...
    (vcc - averageToggleVoltageMinTimeRange);
averageTogglePowerMicroWattsMinTimeRange = ...
    averageTogglePowerMinTimeRange .* 1E6;

% Calculate the maximum values wrt the time b/w tx
averageToggleVoltageMaxTimeRange = ...
    ((heightToggleDivisionTriRelative * widthToggleDivisionTri / 2) + ...
    (heightToggleCapture1TriAbsolute * widthToggleCapture1Tri / 2) + ...
    (heightToggleCapture2TriAbsolute * widthToggleCapture2Tri / 2) + ...
    (heightToggleCompOnTriRelative * widthToggleCompOnTri / 2) + ...
    (heightToggleCompOnRectRelative * widthToggleCompOnRectMaxTime)) ...
    ./ txIntervalRange + ...
    heightToggleCompOffRect;
averageToggleCurrentMaxTimeRange = ...
    averageToggleVoltageMaxTimeRange ./ shuntToggleResistor;
averageTogglePowerMaxTimeRange = ...
    averageToggleCurrentMaxTimeRange .* ...
    (vcc - averageToggleVoltageMaxTimeRange);
averageTogglePowerMicroWattsMaxTimeRange = ...
    averageTogglePowerMaxTimeRange .* 1E6;

% Plot Power (uW) vs. Time b/w Tx (s)
figure;
grid on;
hold on;
title(['Interval Algorithm Power Consumption' ...
    ' w/Toggle wrt Time b/w Tx @ 14 br/min']);
xlabel('Time b/w Tx (s)');
ylabel('Power (uW)');

% We sort of need to cheat in order to get the desired area coloring
% effect. We are just overlaying that shaded max area plot with the min
% area plot in order to show the difference.
area(txIntervalRange,...
    averageTogglePowerMicroWattsMaxTimeRange,'facecolor','y');
```

```
area(txIntervalRange,...
    averageTogglePowerMicroWattsMinTimeRange,'facecolor','w');

% Show the grid on top
set(gca,'Layer','top');

% Plot the min and max values on top of everything else
plot(txIntervalRange,averageTogglePowerMicroWattsMinTimeRange,'gs-');
plot(txIntervalRange,averageTogglePowerMicroWattsMaxTimeRange,'gs-');

% Set the x-axis limit to hide the side lines from the area plots
xlim([min(txIntervalRange) max(txIntervalRange)]);

%************************************************************************
% Plot combination RR and Time b/w Tx plots that will encompass all prior
% plots to allow for comparison. Note: the interval algorithm will not have
% an estimation for 6 br/min since this is outside of its operating range.
%************************************************************************

% Plot wrt to respiratory rate
figure;
grid on;
hold on;
title('Power Consumption wrt RR @ 10-s Time b/w Tx');
xlabel('Respiratory Rate (br/min)');
ylabel('Power (uW)');

% First plot the interval timing w/ toggle estimate so that it appears in
% the background
area(respRateIntervalRange,...
    averageTogglePowerMicroWattsMaxRRRange,'facecolor','y');
area(respRateIntervalRange,...
    averageTogglePowerMicroWattsMinRRRange,'facecolor','w');
set(gca,'Layer','top');
h1 = plot(respRateIntervalRange,...
    averageTogglePowerMicroWattsMinRRRange,'gs-');
plot(respRateIntervalRange,averageTogglePowerMicroWattsMaxRRRange,'gs-');
xlim([min(respRateIntervalRange) max(respRateIntervalRange)]);

% Then plot all the other curves on top of it. Note: The interval algorithm
% without toggling and the RF transmission estimates are frequency
% independent and will be plotted as a straight line.
h2 = plot(respRateIntervalRange,...
    averageIntervalPowerMicroWattsTimeRange(...
    find(txIntervalRange==10) * ...
    ones(size(respRateIntervalRange))),'bd-');
h3 = plot(respRateCountRange,...
    averageCountPowerMicroWattsRRRange,'ro-');
h4 = plot(respRateIntervalRange,...
    averageTxPowerMicroWattsTimeRange(...
    find(txIntervalRange==10)* ...
    ones(size(respRateIntervalRange))),'m^-');

% Print the legend
legend(...
    [h1, h2, h3, h4],...
    'Breath Interval Timing w/ Comp Toggle Max and Min',...
    'Breath Interval Timing w/o Comp Toggle',...
    'Breath Counting',...
    'RF Transmission Alone',...
    'Location','Best');

% Plot wrt to time b/w tx
figure;
grid on;
hold on;
title('Power Consumption wrt Time b/w Tx @ 14 br/min');
xlabel('Time b/w Tx (s)');
ylabel('Power (uW)');

% First plot the interval timing w/ toggle estimate so that it appears in
```

```
% the background
area(txIntervalRange,...
    averageTogglePowerMicroWattsMaxTimeRange,'facecolor','y');
area(txIntervalRange,...
    averageTogglePowerMicroWattsMinTimeRange,'facecolor','w');
set(gca,'Layer','top');
h1 = plot(txIntervalRange,...
    averageTogglePowerMicroWattsMinTimeRange,'gs-');
plot(txIntervalRange,averageTogglePowerMicroWattsMaxTimeRange,'gs-');
xlim([min(txIntervalRange) max(txIntervalRange)]);

% Then plot all the other curves on top of it.
h2 = plot(txIntervalRange,...
    averageIntervalPowerMicroWattsTimeRange,'bd-');
h3 = plot(txIntervalRange,...
    averageCountPowerMicroWattsTimeRange,'ro-');
h4 = plot(txIntervalRange,...
    averageTxPowerMicroWattsTimeRange,'m^-');

% Print the legend
legend(...
    [h1, h2, h3, h4],...
    'Breath Interval Timing w/ Comp Toggle Max and Min',...
    'Breath Interval Timing w/o Comp Toggle',...
    'Breath Counting',...
    'RF Transmission Alone',...
    'Location','Best');

%*************************************************************************
% Extract specific values at RR = 14 br/min and 10-s/15-s/30-s/1-min
% time b/w tx intervals.
%*************************************************************************

% Obtain the matrix indices for the specific intervals of 10/15/30/60 s
powerTimeTableIndices = [...
    find(txIntervalRange==10),...
    find(txIntervalRange==15),...
    find(txIntervalRange==30),...
    find(txIntervalRange==60)];

% Order of addition to the table will be:
%   - Tx Alone
%   - Count Alg Alone
%   - Interval Alg w/o Toggle Alone
%   - Interval Alg w/ Toggle Max Alone
%   - Interval Alg w/ Toggle Min Alone
%   - Count Alg w/ Tx
%   - Interval Alg w/ Tx
%   - Interval Alg w/ Toggle Max w/ Tx
%   - Interval Alg w/ Toggle Min w/ Tx

% Set up the table with proper column titles and empty access slots
powerTimeTable = [10,15,30,60];
powerTimeTable = [powerTimeTable;zeros(9,4)];

% Calculate the rows in which tx is off or is inapplicable
powerTimeTable(2,:) = ...
    averageTxPowerMicroWattsTimeRange(powerTimeTableIndices);
powerTimeTable(3,:) = ...
    averageCountPowerMicroWattsTimeRange(powerTimeTableIndices);
powerTimeTable(4,:) = ...
    averageIntervalPowerMicroWattsTimeRange(powerTimeTableIndices);
powerTimeTable(5,:) = ...
    averageTogglePowerMicroWattsMaxTimeRange(powerTimeTableIndices);
powerTimeTable(6,:) = ...
    averageTogglePowerMicroWattsMinTimeRange(powerTimeTableIndices);

% Calculate the rows in which tx is on
powerTimeTable(7,:) = powerTimeTable(2,:) + powerTimeTable(3,:);
powerTimeTable(8,:) = powerTimeTable(2,:) + powerTimeTable(4,:);
powerTimeTable(9,:) = powerTimeTable(2,:) + powerTimeTable(5,:);
```

```
powerTimeTable(10,:) = powerTimeTable(2,:) + powerTimeTable(6,:)


%*************************************************************************
% Extract specific values at 10-s time b/w tx intervals and 6/15/30/90
% br/min respiratory rates. Note: the interval algorithm will not have an
% estimation for 6 br/min since this is outside of its operating range.
%*************************************************************************

% Obtain the matrix indices for the specific intervals of 10/15/30/60 s
powerRRTableCountIndices = [...
    find(respRateCountRange==6),...
    find(respRateCountRange==15),...
    find(respRateCountRange==30),...
    find(respRateCountRange==90)];
powerRRTableIntervalIndices = [...
    find(respRateIntervalRange==15),...
    find(respRateIntervalRange==30),...
    find(respRateIntervalRange==90)];

% Order of addition to the table will be:
%   - Tx Alone
%   - Count Alg Alone
%   - Interval Alg w/o Toggle Alone
%   - Interval Alg w/ Toggle Max Alone
%   - Interval Alg w/ Toggle Min Alone
%   - Count Alg w/ Tx
%   - Interval Alg w/ Tx
%   - Interval Alg w/ Toggle Max w/ Tx
%   - Interval Alg w/ Toggle Min w/ Tx

% Set up the table with proper column titles and empty access slots
powerRRTable = [6,15,30,90];
powerRRTable = [powerRRTable;zeros(9,4)];

% Calculate the rows in which tx is off or is inapplicable
powerRRTable(2,:) = ...
    averageTxPowerMicroWattsTimeRange(...
    find(txIntervalRange==10)) * ...
    ones(size(powerRRTable(2,:)));
powerRRTable(3,:) = ...
    averageCountPowerMicroWattsRRRange(powerRRTableCountIndices);
powerRRTable(4,[2:4]) = ...
    averageIntervalPowerMicroWattsTimeRange(...
    find(txIntervalRange==10)) * ...
    ones(size(powerRRTable(2,[2:4])));
powerRRTable(5,[2:4]) = ...
    averageTogglePowerMicroWattsMaxRRRange(...
    powerRRTableIntervalIndices);
powerRRTable(6,[2:4]) = ...
    averageTogglePowerMicroWattsMinRRRange(...
    powerRRTableIntervalIndices);

% Calculate the rows in which tx is on
powerRRTable(7,:) = powerRRTable(2,:) + powerRRTable(3,:);
powerRRTable(8,[2:4]) = powerRRTable(2,[2:4]) + powerRRTable(4,[2:4]);
powerRRTable(9,[2:4]) = powerRRTable(2,[2:4]) + powerRRTable(5,[2:4]);
powerRRTable(10,[2:4]) = powerRRTable(2,[2:4]) + powerRRTable(6,[2:4])
```

# Appendix D: Problems and Issues

There are few experiments in which people do not encounter unforeseen problems or issues. This appendix explains those that were experienced during this study.

## D.1. Interrupt Priorities

The CC430 has fixed interrupt priorities; for example, depending on what timer module is used, an interrupt from the radio module may be prioritized higher or lower than a timer interrupt. Not understanding how the different interrupts interact with each other can cause unexpected behavior to the unwary.

## D.2. Receiver Bug

With its current code iteration, sometimes the receiving module will fail to reset itself so that the timeout LED will be constantly on. This indicates that it is no longer receiving any RF data, since new information would reset the timeout timer.

The general logic for the receiving module is sound. This notion has been established from two observations: (1) Prior to setting the receiver module up for communication via RS232, all indication was done via blinking LEDs, and no timeout was observed; (2) The current logic correctly displays data received for several RF transactions before entering a timeout "loop".

Concerning the real-world application of these transmitting and receiving devices, this sort of bug is gravely unacceptable, especially in the medical field; however, since the receiving apparatus is not the concentration of this thesis, a full and detailed debugging session of the code has been left for future implementations. The receiving module's purpose is simply to illustrate that an RF signal from the transmitter can be correctly received by the receiver; what the receiver does with this information, such as preparing the information for display, does not have an effect on this thesis's focus, which is the power consumption of the transmitter.

For the curious, the original LED iteration of the receiver code can be found in Appendix A, with an explanation of why RS232 needed to be implemented.

## D.3. Near-Field Phenomenon

During an early experiment, in which sample code from TI, explained in Appendix A, was used to test the RF functionality, it seemed like the RF signal was incredibly unreliable; the signal was only received possibly less than half of the time. However, the two modules were sitting in the close proximity of a few inches. When the two modules were separated further, maybe a meter or so, the signal seemed to be received 100% of the time for a few isolated trials.

Changing the distance between the modules was done several times with a few transmissions for each; and from the reactions of the modules, it looks like a near-field phenomenon was the cause for unreliability.

Although the near-field phenomenon affects the reliability of transmission to the point that characterizing it is very vital to the safety and reliability of the real-world application, understanding it is not necessary for this thesis's purpose. This thesis compares the power consumption for two computer algorithms; although altering the distance between the transmitter and receiver will affect the RF circuit and software settings, thus affecting the overall power consumption, the extra power consumption is not associated with the algorithm logic and execution. The contribution of power consumption by the RF module is factored out from this thesis, since the frequency of transmissions required depends on the situational application of the end devices. Despite its negligibility in this thesis, the RF power consumption remains as a very important variable for the overall system that must be studied in the future.

## D.4. Buggy IDE

The IDE used for programming the CC430 was CCS4, and like most other complicated software the package did not come without a few bugs. For example, if a user steps through the code and expects a certain variable to be updated, the IDE does not indicate this update properly in its variable displays. The author has had a similar experience in the past with CCS. Validation of correct code execution was done the "old-fashioned", more reliable way, using external indicators such as an LED.

## D.5. Power Supply Through JTAG

For debugging purposes, it is very convenient to power the evaluation board through the JTAG programming cable from a computer; however, future studies utilizing this technique should be wary that this creates a great amount of noise when measuring the voltage in the setup mentioned in Chapter 4. Therefore, it is recommended that a separate power supply be used when conducting any measurements that scrutinize the characteristics of the CC430 electrical capabilities.

# Appendix E: Terminology

This appendix provides a reference for understanding the content of this thesis. The sub-appendix titled “Disambiguation” describes everyday terms or phrases that might carry a particular significance or weight or convey a certain connotation with regard to this study. Also, sometimes, the standardization or consolidation of terms in many fields results in a collision of language and jargon. This conflict can cause documents that address all these fields to be confusing; conventions in one field might have a completely different meaning in another. In an attempt to mitigate this confusion, the sub-appendix titled “Conventions” clarifies which conventions are used in this thesis. In addition, the “Abbreviations” sub-appendix denotes the meaning of any shorthand or abbreviations used.

## E.1. Disambiguation

This thesis focuses on the subsystem of a larger system. In order to differentiate between the two systems, the term “project” refers to the scenario of the encompassing system, whereas the term “study” refers to the situation of the subsystem. The term “overall system” refers to the system of the project, and the term “subsystem” refers to the system of this study.

There are two relatively identical circuits used in this study; in order to differentiate between these circuits and since communication will be unidirectional, each will be referred to by their major operation regarding their wireless functionality. Future studies that require bidirectional communication, nodal networks, or the like should rename the circuits accordingly. The transmitting circuit, or “transmitter”, is meant to be attached to the patient being monitored and its low-power optimization is the motivation of this study. The receiving circuit, or “receiver”, is meant to be integrated into a monitoring system that displays the received data to the person overseeing the patient, typically a medical practitioner. The transmitter may also be referred to as “on-patient”, while the receiver may also be referred to as “off-patient”.

## E.2. Conventions

All respiratory rates will be given in the typical unit of breaths per minute unless otherwise stated. To avoid confusion, this study will represent breaths per minute with br/min rather than BPM, which can also stand for beats per minute when conducting a heart rate measurement. If other increments of time are preferred, similar notations will be used, such as br/s to represent breaths per second.

## E.3. Abbreviations and Shorthand

The abbreviations and shorthand utilized in this thesis are summarized in Table E.1 for easy reference.

**Table E.1: Abbreviations and Shorthand.**

| Abbreviation | Explanation |
|---|---|
| µA | microampere(s) |
| µV | microvolt(s) |
| µW | microwatt(s) |
| AC | alternating current |
| ADC | analog-to-digital converter |
| ASP | analog signal processor |
| br/min | breath(s) per minute |
| CCS4 | Code Composer Studio Version 4.x |
| $CO_2$ | carbon dioxide |
| CPU | central processing unit |
| DC | direct current |
| DMA | direct memory access |
| DSP | digital signal processor |
| DVS | dynamic voltage scaling |
| ECG | electrocardiogram |
| Hz | hertz |
| IC | integrated circuit |
| ICU | Intensive Care Unit |
| IDE | Integrated Development Environment |
| I/O | input/output |
| LCD | liquid crystal display |
| LED | light-emitting diode |
| LPM | low-power mode |
| mA | milliampere(s) |
| MHz | megahertz |
| mV | millivolt(s) |
| mW | milliwatt(s) |
| PCB | printed circuit board |
| PMM | power-management module |
| RAM | random-access memory |
| RF | radio frequency |
| RR | respiratory rate |
| RTC | real-time clock |
| SoC | system-on-chip |
| TI | Texas Instruments Incorporated |
| UCS | unified clock system |
| USCI | universal serial communication interface |
| WDT | watchdog timer |

# References


[1] D.R. Goldhill, S.A. White, and A. Sumner, "Physiological values and procedures in the 24 h before ICU admission from the ward," *Anaesthesia*, vol. 54, no. 6, pp. 529-534, June 1999.

[2] John F. Fieselmann, Michael S. Hendryx, Charles M. Helms, and Douglas S. Wakefield, "Respiratory Rate Predicts Cardiopulmonary Arrest for Internal Medicine Inpatients," *Journal of General Internal Medicine*, vol. 8, pp. 354-360, July 1993.

[3] Michelle A. Cretikos et al., "Respiratory rate: the neglected vital sign," *The Medical Journal of Australia*, 2008. [Online]. Available: http://www.mja.com.au/public/issues/188_11_020608/cre11027_fm.html. [Accessed: Sept. 14, 2011].

[4] M.R. Neuman, "Vital Signs [Tutorial]," *Pulse, IEEE*, vol. 2, no. 1, pp. 39-44, Jan.-Feb. 2011.

[5] Lynn S. Bickley, *Bates' Guide to Physical Examination and History Taking*, 10th ed.: Lippincott Williams & Wilkins; Tenth, North American Edition, 2008.

[6] U.S. Department of Health and Human Services: Food and Drug Administration: Center for Devices and Radiological Health: Anesthesiology and Respiratory Devices Branch: Division of Anesthesiology, General Hospital, Infection Control, and Dental Devices: Office of Device Evaluation, "Class II Special Controls Guidance Document: Apnea Monitors; Guidance for Industry and FDA", *U.S. Department of Health and Human Services*, Jul. 17, 2002. [Online]. Available: http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm072846.htm#9 [Accessed: Sept. 24, 2011].

[7] Nicolas André et al., "Dew-based Wireless Mini Module for Respiratory Rate Monitoring," *Sensors Journal, IEEE*, vol. PP, no. 99, pp. 1, 2011.

[8] Lynette Jones, Nikhila Deo, and Brett Lockyer, "Wireless Physiological Sensor System for Ambulatory Use," *Wearable and Implantable Body Sensor Networks, 2006. BSN 2006. International Workshop on*, pp. 4-149, April 2006.

[9] Philips Healthcare, "VitalSense-XHR Heart Rate & Respiration Sensor", *Philips Resprionics*, 2011. [Online]. Available: http://vitalsense.respironics.com/default.asp. [Accessed: Sept. 26, 2011] .

[10] Masimo, "Continuous and Noninvasive Respiration Rate with Rainbow Acoustic Monitoring", *Masimo*, 2010. [Online]. Available: http://www.masimo.com/rra/index.htm. [Accessed: Sept. 26, 2011].

[11] A. Bates, M. J. Ling, D. K. Arvind, and J. Mann, "Respiratory Rate and Flow Waveform Estimation from Tri-axial Accelerometer Data," *Body Sensor Networks (BSN), 2010 International Conference on*, pp. 144-150, June 2010.

[12] Kai Medical, "Kai Medical Products", *Kai Medical*, 2011. [Online]. Available: http://www.kaimedical.com/en/products.php. [Accessed: Sept. 26, 2011].

[13] Anders Høst-Madsen et al., "Signal Processing Methods for Doppler Radar Heart Rate Monitoring". [Online]. Available: http://www.ee.hawaii.edu/~madsen/papers/Host-Madsen.pdf. [Accessed: Sept. 26, 2011].

[14] Wansuree Massagram, Victor M. Lubecke, Anders Høst-Madsen, and Olga Boric-Lubecke, "Assessment of Heart Rate Variability and Respiratory Sinus Arrhythmia via Doppler Radar," *IEEE Transactions on Microwave Theory and Techniques*, vol. 57, no. 10, pp. 2542-2549, October 2009.

[15] Jenshan Lin and Changzhi Li, "Wireless Non-Contact Detection of Heartbeat and Respiration Using Low-Power Microwave Radar Sensor," *Asia-Pacific Microwave Conference 2007*, June 2008.

[16] Amy D. Droitcour et al., "Non-Contact Respiratory Rate Measurement Validation for Hospitalized Patients," *Engineering in Medicine and Biology Society, 2009. EMBC 2009. Annual International Conference of the IEEE*, pp. 4812-4815, September 2009.

[17] The Engineer, "Internet transceivers could test breathing rates non-invasively", *The Engineer*, Sept. 19, 2011. [Online]. Available: http://www.theengineer.co.uk/sectors/medical-and-healthcare/news/internet-transceivers-could-test-breathing-rates-non-invasively/1010109.article [Accessed: Sept. 23, 2011].

[18] Medica.de, "Surgery Patients: Catching a Breath – Wirelessly", *Medica*, Sept 22, 2011. [Online]. Available.: http://www.medica.de/cipp/md_medica/custom/pub/content,oid,34661/lang,2/ticket,g_u_e_s_t/local_lang,2/~/Catching_a_Breath_%E2%80%93_Wirelessly.html. [Accessed: Sept. 23, 2011].

[19] Ben Coxworth, "Wireless system uses off the shelf components to monitor patients' breathing", *Gizmag,* Sept. 20, 2011. [Online]. Available: http://www.gizmag.com/wireless-transceivers-monitor-breathing/19890/. [Accessed: Sept. 24, 2011].

[20] Institute of Physics, "Energy harvesting", *Institute of Physics*. [Online]. Available: http://www.iop.org/resources/energy/index.html. [Accessed: Sept. 24, 2011].

[21] Murugavel Raju, "Energy Harvesting: ULP meets energy harvesting: A game-changing combination for design engineers", *Texas Instruments*, 2008. [Online]. Available: http://www.ti.com/corp/docs/landing/cc430/graphics/slyy018_20081031.pdf. [Accessed: Sept. 24, 2011].

[22] Ehsaneh Shahhaidar, Olga Boric-Lubecke, Reza Ghorbani, and Michael Wolfe, "Electromagnetic Generator as Respiratory Effort Energy Harvester," *Power and Energy Conference at Illinois (PECI), 2011 IEEE*, Champaign, USA, pp. 1-4, February 2011.

[23] Bryson Padasdao and Olga Boric-Lubecke, "Respiratory Rate Detection Using a Wearable Electromagnetic Generator," *Engineering in Medicine and Biology Society, EMBC, 2011 Annual International Conference of the IEEE*, August 2011.

[24] ScienceDaily, "Smart Pants: Computer Engineers Develop Clothes that Sense and Interpret Movements", *ScienceDaily*, Apr. 1, 2011. Available: ttp://www.sciencedaily.com/videos/2006/0402-smart_pants.htm. [Accessed: Sept. 24, 2011].

[25] Ben Schiller, "A Smart T-Shirt That Monitors Vital Signs Without Wires | Fast Company", *Fast Company*, Sept. 27, 2011. [Online]. Available: http://www.fastcompany.com/1782989/a-smarter-t-shirt-to-monitor-your-vital-signs-and-take-all-those-wires-off-your-chest. [Accessed: Sept. 27, 2011].

[26] ScienceDaily, "Clothing to Power Personal Computers", *ScienceDaily*, Aug. 18, 2011. [Online]. Available: http://www.sciencedaily.com/releases/2010/08/100817143810.htm. [Accessed: Sept. 24, 2011].

[27] Alexandros Pantelopoulos and Nikolaos G. Bourbakis, "A Survey on Wearable Sensor-Based Systems for Health Monitoring and Prognosis," *IEEE Transactions on Systems, Man, and Cybernetics—Part C: Applications and Reviews*, vol. 40, no. 1, January 2010.

[28] ScienceDaily, "Wireless network in hospital monitors vital signs, even as patients move about", *ScienceDaily*, Aug. 5, 2011. [Online]. Available: http://www.sciencedaily.com/releases/2011/08/110804145418.htm. [Accessed: Sept. 27, 2011].

[29] Jin Huang, Lei Wang, Lian-Kang Chen, and Yuan-Ting Zhang, "A Low Power Wearable Transceiver for Human Body Communication," *31st Annual International Conference of the IEEE EMBS*, Minneapolis, USA, pp. 3802-3805, September 2009.

[30] Z. D. Nie, L. Wang, W. G. Chen, T. Zhang, and Y. T. Zhang, "A Low Power Biomedical Signal Processor ASIC Based on Hardware Software Codesign," *31st Annual International Conference of the IEEE EMBS*, Minneapolis, USA, pp. 2559-2562, September 2009.

[31] Xin Liu et al., "Multiple Functional ECG Signal is Processing for Wearable Applications of Long-Term Cardiac Monitoring," *IEEE Transaction of Biomedical Engineering*, vol. 58, no. 2, pp. 380-389, February 2011.

[32] Zulhakimi Razak, Ahmet Erdogan, and Tughrul Arslan, "ASIC Design of an Adaptive Control Unit for Reconfigurable Analog-to-Digital Converters," *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, Lixouri, Greece, pp. 179-184, July 2010.

[33] Shahin Farshchi, Aleksey Pesterev, Paul H. Nuyujukian, Istvan Mody, and Jack W. Judy, "Bi-Fi: An Embedded Sensor/System Architecture for Remote Biological Monitoring," *IEEE Transactions on Information Technology In Biomedicine*, vol. 11, no. 6, pp. 611-618, November 2007.

[34] Ferenc Kovács, Miklós Török, and István Habermajer, "A Rule-Based Phonocardiographic Method for Long-Term Fetal Heart Rate Monitoring," *IEEE Transactions on Biomedical Engineering*, vol. 47, no. 1, pp. 124-130, January 2000.

[35] Mel M. S. Ho, Tor S. Lande, and Christopher Toumazou, "Efficient Computation of the LF/HF Ratio in Heart Rate Variability Analysis Based on Bitstream Filtering," *Biomedical Circuits and Systems Conference, 2007. BIOCAS 2007. IEEE*, Montreal, Canada, pp. 17-20, November 2007.

[36] Yindar Chuo et al., "Mechanically Flexible Wireless Multisensor Platform for Human Physical Activity and Vitals Monitoring," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 4, no. 5, pp. 281-294, October 2010.

[37] Fei Hu, Shruti Lakdawala, Qi Hao, and Meikang Qiu, "Low-Power, Intelligent Sensor Hardware Interface for Medical Data Preprocessing," *Information Technology in Biomedicine, IEEE Transactions on*, vol. 13, no. 4, pp. 656-663, July 2009.

[38] B.Y. Huang et al., "A Pilot Study on Low Power Pulse Rate Detection Based on Compressive Sampling," *31st Annual International Conference of the IEEE EMBS*, Minneapolis, USA, pp. 753-756, September 2009.

[39] Urs Anliker et al., "AMON: A Wearable Multiparameter Medical Monitoring and Alert System," *IEEE Transactions on Information Technology In Biomedicine*, vol. 8, no. 4, pp. 415-427, December 2004.

[40] Robert Rieger and John T. Tayor, "An Adaptive Sampling System for Sensor Nodes in Body Area Networks," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 17, no. 2, pp. 183-189, April 2009.

[41] C. Gonzalez, J. Jimenez-Leube, and J. Sanz-Maudes, "Wireless Sensor Node: monitoring low-rate phenomena using a low-power RF-enabled microcontroller," *Intelligent Signal Processing, 2007. WISP 2007. IEEE International Symposium on*, Alcala de Henares, Spain, pp. 1-6, October 2007.

[42] Texas Instruments, "Bringing personal and industrial wireless networking to the mass market", *Texas Instruments*, 2011. Available: http://www.ti.com/corp/docs/landing/cc430/index.htm. [Accessed: Sept. 24, 2011].

[43] Texas Instruments, "MSP430 USB Debugging Interface", *Texas Instruments*, 2011. [Online]. Avaiable: http://www.ti.com/tool/msp-fet430uif. [Accessed: Sept. 24, 2011].

[44] Texas Instruments, "MSP430™16-bit Ultra-Low Power MCUs – CC430 RF SoC Series – CC430F6137 – TI.com", *Texas Instruments*, 2011. [Online]. Avaiable: http://www.ti.com/product/cc430f6137. [Accessed: Sept. 26, 2011].

[45] Texas Instruments, "CC430 Wireless Development Tool – EM430F6137RF900 – TI Tool Folder", *Texas Instruments*, 2011. [Online]. Avaiable: http://www.ti.com/tool/em430f6137rf900. [Accessed: Sept. 26, 2011].

[46] I-Jan Wang et al., "A Wearable Mobile Electrocardiogram Measurement Device with Novel Dry Polymer-Based Electrodes," *TENCON 2010 – 2010 IEEE Region 10 Conference*, pp. 379-384, November 2010.

[47] B. Tovar Corona, J. E. Gonzalez Villarruel, H. Becerra Esquivel, A. Juárez Carrasco, and A. Espíritu Santo Rincón, "Prototype of a Portable Platform for ECG Monitoring and Diagnostic Applications," *Electrical Engineering, Computing Science and Automatic Control, 2008. CCE 2008. 5th International Conference on*, pp. 223-227, November 2008.

[48] Darwin S. David, Pradeep K. Gopalakrishnan, and T. Hui Teo, "Low-Power Low-Voltage Digital System for Wireless Heart Rate Monitoring Application," *Integrated Circuits, 2007. ISIC '07. International Symposium on*, pp. 212-215, September 2007.

[49] Irina Hossain and Zahra Moussavi, "Respiratory Airflow Estimation by Acoustical Means ," *Engineering in Medicine and Biology, 2002. 24th Annual Conference and the Annual Fall Meeting of the Biomedical Engineering Society EMBS/BMES Conference, 2002. Proceedings of the Second Joint*, vol. 2, pp. 1476-1477, 2002.

[50] Saiful Huq, Azadeh Yadollahi, and Zahra Moussavi, "Breath Analysis of Respiratory Flow Using Tracheal Sounds," *Signal Processing and Information Technology, 2007 IEEE International Symposium on*, pp. 414-418, December 2007.

[51] Azadeh Yadollahi and Zahra M. K. Moussavi, "The Effect of Anthropometric Variations on Acoustical Flow Estimation: Proposing a Novel Approach for Flow Estimation Without the Need for Individual Calibration," *Biomedical Engineering, IEEE Transactions on*, vol. 58, no. 6, pp. 1663-1670, June 2011.

[52] Mohammed Habul, Richard Hallmark, Aleksandr Kotlyanskiy, Pooja Parekh, and Ridha Kamoua, "Sleep Apnea Diagnostic Device," *Systems, Applications and Technology Conference (LISAT), 2011 IEEE Long Island*, pp. 1-5, May 2011.

[53] H. R. Silva et al., "Wireless Hydrotherapy Smart-Suit Network for Posture Monitoring," *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*, pp. 2713-2717, June 2007.

[54] Emilio Sardini, Mauro Serpelloni, and Marco Ometto, "Multi-parameters Wireless Shirt for Physiological Monitoring," *Medical Measurements and Applications Proceedings (MeMeA), 2011 IEEE International Workshop on*, pp. 316-321, May 2011.

[55] Jun Ohnishi, Yoshiharu Yonezawa, and Ishio Ninomiya, "A Compact, Electrode-Mounted, ECG R-R Interval Recording System," *[Engineering in Medicine and Biology, 1999. 21st Annual Conf. and the 1999 Annual Fall Meeting of the Biomedical Engineering Soc.] BMES/EMBS Conference, 1999. Proceedings of the First Joint*, vol. 1, no. 1, pp. 293, October 1999.

[56] Soumyajit Mandal, Lorenzo Turicchia, and Rahul Sarpeshkar, "A Battery-Free Tag for Wireless Monitoring of Heart Sounds," *Wearable and Implantable Body Sensor Networks, 2009. BSN 2009. Sixth International Workshop on*, pp. 201-206, June 2009.

[57] Phil Corbishley and Esther Rodríguez-Villegas, "Breathing Detection: Towards a Miniaturized, Wearable, Battery-Operated Monitoring System," *Biomedical Engineering, IEEE Transactions on*, vol. 55, no. 1, pp. 196-204, January 2008.

[58] Michael D. Scott, Bernhard E. Boser, and Kristofer S. J. Pister, "An Ultra-Low Power ADC for Distributed Sensor Networks," *Solid-State Circuits Conference, 2002. ESSCIRC 2002. Proceedings of the 28th European*, pp. 255-258, September 2002.

[59] B. Otis et al., "Low Power IC Design for Energy Harvesting Wireless Biosensors," *Radio and Wireless Symposium, 2009. RWS '09. IEEE*, pp. 5-8, January 2009.

[60] Lennart Yseboodt et al, "Ultra-low-power DSP design", *EE Times*, Aug. 30, 2007. [Online]. Available: http://www.eetimes.com/design/signal-processing-dsp/4017529/Ultra-low-power-DSP-design?pageNumber=0. [Accessed: Sept. 18, 2011].

[61] Darlene Storm, "Black Hat: Lethal Hack and wireless attack on insulin pumps to kill people – Computerworld Blogs", *Computerworld*, Aug. 4, 2011. [Online]. Available: http://blogs.computerworld.com/18744/black_hat_lethal_hack_and_wireless_attack_on_insulin_pumps_to_kill_people. [Accessed: Sept. 27, 2011].

[62] Morgan Hunter HealthSearch, "Hacking Risk of Wireless Medical Devices | Morgan Hunter HealthSearch Blog", *Morgan Hunter HealthSearch*, Sept. 15, 2011. [Online]. Available: http://blog.mhhealthsearch.com/2011/09/can-wireless-medical-devices-be-hacked/. [Accessed: Sept. 27, 2011].

[63] Dan Goodin, "Wireless Health Group News | LinkedIn", *The Register*, Aug. 25, 2011. [Online]. Available: http://www.linkedin.com/news?viewArticle=&articleID=728987442&gid=2181454&type=member&item=67896096&articleURL=http%3A%2F%2Fwww.theregister.co.uk%2F2011%2F08%2F25%2Fmedtronic_insulin_pump_hacking%2F&urlhash=WbzU&goback=.gde_2181454_member_67896096. [Accessed: Sept. 27, 2011].

[64] Associated Press, “Jay Radcliffe Discovers Way to Hack Insulin Pumps, Monitors: Diabetes”, *Newser*, Aug. 5, 2011. [Online]. Available: http://www.newser.com/story/125174/jay-radcliffe-discovers-way-to-hack-insulin-pumps-monitors-diabetes.html. [Accessed: Sept. 27, 2011].

[65] Jr., John W. Clark et al., *Medical Instrumentation Application and Design*, 4th ed., John G. Webster, Ed. Hoboken, New Jersey, United States of America: John Wiley & Sons, Inc., 2010.

[66] U.S. Department of Health and Human Services: Food and Drug Administration: Center for Devices and Radiological Health: Electrophysics Branch: Division of Physical Science: Office of Science and Engineering Laboratories, “Radio-Frequency Wireless Technology in Medical Devices”, *U.S. Department of Health and Human Services*, Jan. 3, 2007. [Online]. Available: http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm077210.htm#5. [Accessed: Sept. 30, 2011].

[67] Federal Communications Commission, “OET – Radio Frequency Safety”, *Federal Communications Commission*, Apr. 20, 2011. [Online]. Available: http://transition.fcc.gov/oet/rfsafety/. [Accessed: Sept. 30, 2011].

[68] Federal Communications Commission, “2007 CFR Title 47, Volume 1”, *National Archives and Records Administration*, Oct. 1, 2007. [Online]. Available: http://www.access.gpo.gov/nara/cfr/waisidx_07/47cfr15_07.html. [Accessed: Sept. 30, 2011].

[69] Texas Instruments, “SimpliciTI™ - RF Made Easy: Network Protocol for Sub-1 GHz, 2.4 GHz and IEEE 802.15.4 RF ICs”, *Texas Instruments*, 2008. [Online]. Avaiable: http://www.ti.com/corp/docs/landing/simpliciTI/index.htm. [Accessed: Nov. 29, 2011].

[70] Texas Instruments, “Temperature, Wireless, USB Data Logger | Watch Development Tool”, *Texas Instruments*, 2010. [Online]. Avaiable: http://www.ti.com/tool/ez430-chronos&DCMP=Chronos&HQS=Other+OT+chronos. [Accessed: Nov. 28, 2011].