Temperature-Compensated BLE Transmission from a Crystal-Free Mote

by Titan Yuan

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor Kristofer S.J. Pister
Research Advisor

5/27/2020

(Date)

* * * * * * *

Prabal K Dutta

Professor Prabal Dutta
Second Reader

5/29/2020

(Date)

Abstract

Temperature-Compensated BLE Transmission from a Crystal-Free Mote

by

Titan Yuan

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kristofer S.J. Pister, Chair

As wireless IoT nodes become increasingly smaller, one solution is to build a wireless sensor node without a crystal, reducing the size, cost, and power consumption of the node. This led to the development of a crystal-free wireless system-on-chip with a standards-compatible radio that can communicate with off-the-shelf IEEE802.15.4 or Bluetooth Low Energy (BLE) devices. However, the lack of an external frequency reference makes radio operation over supply and temperature variations challenging. In particular, the frequency drift of the mote's local oscillator over temperature exceeds the frequency error requirement specified in the 802.15.4 and BLE standards.

In this work, we first present a method to calibrate a crystal-free wireless system-on-chip for use as an IoT temperature sensor between 0 °C and 100 °C. Using the frequency ratio of two clocks on the mote, the 2 MHz chipping clock for the chip's transmitter and the 32 kHz sleep timer, both of which would already be running during normal radio operation, we can linearly estimate the ambient temperature with an error of less than 2 °C.

We then describe how we can configure the chip and tune the local oscillator (LO) using a 15-bit frequency tuning code to transmit BLE advertising packets on a specified BLE channel. If we simply sweep the lowest five bits of the LO frequency tuning code, we show that we can transmit BLE packets over a range of around 20 °C. To be more power-efficient, though, we discuss two possible approaches to compensate the LO frequency for temperature changes. Instead of sweeping the tuning code, we can calculate the correct frequency setting for the LO given a temperature estimate. Alternatively, we demonstrate that we can use an external OpenMote that constantly transmits 802.15.4 packets on a specified frequency channel as a frequency reference by adjusting the crystal-free mote's LO frequency based on the intermediate frequency (IF) of the received OpenMote packets. Tuning only the last five bits of the tuning code using this approach allows the crystal-free mote to operate as a BLE beacon or 802.15.4-to-BLE translator over a temperature range of around 20 °C. Finally, while the

crystal-free mote's BLE packets can be received by commercial BLE devices, receiving BLE packets transmitted by off-the-shelf BLE devices does not work on the mote.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, there has been a rapid growth in the field of internet of things (IoT). Low-power wireless technology has enabled many exciting applications, including a wide variety of sensors, MEMS actuators, and microrobots [5, 6]. There have also been numerous standards developed for IoT mesh networks, such as 6TiSCH [4] that is based on the Time Synchronized Channel Hopping (TSCH) standard of IEEE802.15.4, where IoT nodes synchronize with each other and use channel hopping to reduce power consumption and improve network reliability [3, 4]. All of these applications require an IoT node that is small, lightweight, low-cost, low-power, and compatible with off-the-shelf network devices.

The Single Chip Micro Mote (SCµM) was developed to replicate all of the functionalities of a wireless sensor node on a single chip. It was created specifically for microrobotic applications and features an ARM Cortex M0 microprocessor, an optical receiver, and a radio compliant with the 802.15.4 and Bluetooth Low Energy (BLE) standards [9] on a $2 \times 3 \times 0.3\,\mathrm{mm}^3$ chip.

Most commercial wireless nodes feature a crystal that functions as an external frequency reference whose frequency is insensitive to voltage and temperature variations. However, in order to reduce the cost and power consumption of the chip, SCµM does not have a crystal. Instead, it relies on CMOS oscillators as the clocks for the microprocessor and the radio. One benefit of this is that SCµM requires just an antenna and a battery in order to function as a wireless sensor node. Therefore, it can be used as a wireless controller for microrobots, attached within a bandage to function as a small body temperature sensor, or placed on top of insects as a tracker.

As one can imagine, though, building a small, low-cost, and crystal-free mote that can operate its radio over supply and temperature variations is challenging. In this thesis, we will primarily focus on SCµM's radio operation over temperature. While crystal oscillators vary by around 40 ppm within the entire temperature range, the local oscillator on SCµM, whose frequency dictates the channel on which the radio is transmitting and receiving, features a temperature coefficient of around $-40\,\mathrm{ppm}/^\circ\mathrm{C}$ [9]. However, the 802.15.4 specifications require the local oscillator to have a frequency error of no more than 40 ppm [9], and the BLE specifications require a frequency error of no more than 50 ppm [1] although the BLE carrier drift specification is not definitive.

In this thesis, we first introduce the SCµM chip (Chapter 2). We describe how it is programmed via its optical receiver and how the on-chip oscillators are calibrated after bootloading using optical start frame delimiter (SFD) interrupts. Afterwards, we discuss SCµM's on-chip oscillators and their corresponding frequency stability. In particular, we are interested in compensating the frequency of the 2.4 GHz local oscillator over temperature variations. While the LO frequency can be tuned using a 15-bit LC frequency tuning code, the LO frequency is not monotonic with respect to this tuning code. Thus, we outline two approaches at creating a monotonic function in order to facilitate LO frequency tuning.

We then present some work allowing SCµM to operate its radio over temperature variations (Chapter 3). The on-chip oscillator frequencies are calibrated once after bootloading, but a one-time calibration is insufficient for crystal-free radio operation across temperature. Previous work relies on periodic network compensation to adjust the local oscillator frequency by listening for other packets in the environment, but we show how we can use the ratio of the free-running 2 MHz and 32 kHz RC oscillator frequencies to develop a linear model that estimates the ambient temperature. The 2 MHz oscillator is used as the chipping clock for the chip's transmitter, and the 32 kHz can be used as a sleep timer for SCµM between periodic radio operation, so using these two oscillators to estimate the temperature does not consume extra power. SCµM could then function as an IoT temperature sensor.

Since SCµM was originally designed to transmit and receive 802.15.4 packets, we also describe how we configure SCµM to transmit BLE packets that can be received by commercial devices, such as smartphones, so that it can interface between 802.15.4 mesh networks and other commercial devices (Chapter 4). After calibrating the local oscillator frequency using optical start frame delimiter (SFD) interrupts, we show that we can transmit BLE-compliant packets over a range of 20 °C simply by sweeping the fine LO frequency tuning code. To reduce power consumption, though, we discuss two possible approaches. One option is to use the temperature estimate based on the ratio of the 2 MHz and 32 kHz oscillator frequencies to adjust the LO frequency. Another option, if SCµM is functioning as a 802.15.4-to-BLE translator, is to keep track of two frequency settings, one for transmitting the BLE packets and one for receiving the 802.15.4 packets. We adjust the 802.15.4 RX frequency setting based on the intermediate frequency (IF) of the received 802.15.4 packets, and we adjust the BLE TX frequency setting by the same amount.

Finally, we describe some results regarding receiving BLE packets on SCµM (Chapter 5). We show that transmitting BLE packets from a SCµM to another SCµM works as expected because SCµM's BLE packets are modulated using frequency-shift keying (FSK). However, receiving BLE packets from off-the-shelf devices, which are modulated using Gaussian frequency-shift keying (GFSK), does not work on SCµM.

In the appendix, we describe the software developed for SCµM for the aforementioned applications. We describe how to perform temperature calibration on SCµM to find the linear model for the temperature estimate based on the 2 MHz and 32 kHz frequency ratio. We also show how to configure SCµM and find the LO frequency tuning codes to transmit BLE packets over temperature variations.

In this thesis, all work described uses a SCµM3C board.

Chapter 2

Single-Chip Micro Mote (SCµM)

2.1 Overview

SCµM is a $2 \times 3 \times 0.3\,\mathrm{mm}^3$ crystal-free single-chip micro mote that features an ARM Cortex M0 microprocessor, a standards-compatible IEEE802.15.4 or Bluetooth Low-Energy (BLE) radio, and an optical receiver for bootloading [8]. SCµM was developed to operate in OpenWSN networks implementing the 802.15.4 time synchronized channel hopping (TSCH) standard [4] and for use in microrobotic applications because of its low size and cost. SCµM features 64 kB of SRAM program memory and 64 kB of SRAM data memory.

Notably, SCµM does not require any external components to operate except a power supply and an antenna for its radio. It does not have any external frequency reference for its radio and instead relies on multiple free-running oscillating circuits for timekeeping.

Using the optical receiver, we can program the firmware on the mote. Additionally, the optical receiver is used during initial bootloading to receive multiple optical start frame delimiters (SFD) timed 100 ms apart to calibrate SCµM's free-running oscillating circuits. After the initial programming of the SCµM chip and the subsequent oscillator frequency calibration, the mote can then transmit packets compliant with the 802.15.4 or BLE standards.

For development, we usually wirebond the SCµM chip onto a development PCB, as shown in Figure 2.1. This allows us to attach an antenna via an SMA connection, access the GPIO pins for input and output, and measure the voltages at various points on SCµM. The development PCB also features an FTDI RS232/UART chip, so that SCµM can transmit to and receive bytes from a connected computer over UART. This is extremely useful for debugging since we can then log SCµM's register values and state variables from the firmware using `printf` statements.

We have also developed smaller PCBs, as shown in Figure 2.2, on which to wirebond SCµM chips, such that the only inputs are power and ground. They have a wirebonded antenna and are programmed the same way as a SCµM on a development board via the optical receiver on the chip. These boards lack any debugging unless we probe the pads and are primarily used as wireless transceivers.

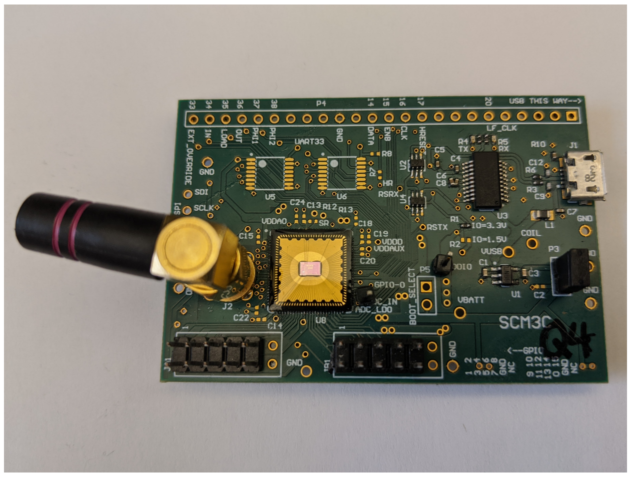


Figure 2.1: Development board Q4 with a wirebonded SCμM chip that I used for my experiments.

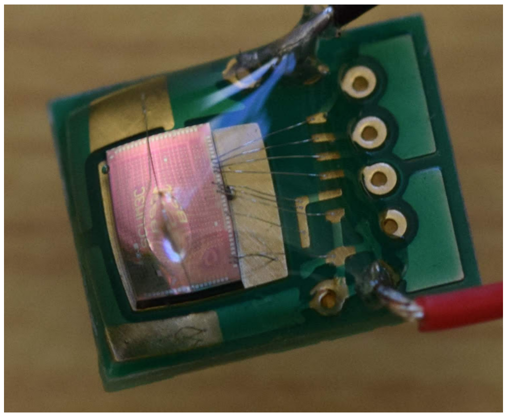


Figure 2.2: SCμM wirebonded onto a smaller PCB with just the power and ground wires.

2.2 Programming

Programming SCµM is accomplished using an external infrared LED connected to a Teensy 3.6 microcontroller. We send optical pulses, where a `1` bit is represented by a long pulse and a `0` bit is represented by a short pulse, to the optical receiver on the SCµM chip in order to transmit the executable binary and to calibrate its oscillators [14]. For best performance, it is recommended to position the LED around 2 cm above the corner of the SCµM chip over where the optical receiver is located.

The entire start-up sequence consists of two stages: bootloading and optical frequency calibration. First, we transmit the binary to SCµM that then performs a cyclic redundancy check (CRC) on the payload. After booting, since SCµM does not have an external frequency reference, we then have the LED send a magic byte sequence every 100 ms. These start frame delimiters (SFDs) trigger an interrupt on SCµM, so that it can tune its on-board oscillators. After SCµM has received 25 SFD interrupts, it starts executing the program.

Bootloading

When bootloading SCµM, the optical programmer[1] first sends 100 bytes of `0b01010101` as a preamble before the start symbol. If we wish to hard reset SCµM and re-program its flash, the start symbol is $[169, 176, 167, 50]$. The programmer then sends 200 bytes of `0b01010101` to allow SCµM to reset itself followed by the actual binary payload encoded in 4B/5B. Finally, the programmer sends 600 `1` bits to clock through all of the received bits on SCµM. If we wish to skip the hard reset, the start symbol is $[184, 84, 89, 40]$. Note that all of these bytes are transmitted in little-endian format.

The received optical data is self-clocked, so the clock to sample the received optical pulses is a delayed replica of the data signal. As shown in Figure 2.3, a `1` bit is thus a long pulse because when the delayed clock signal goes high, the data signal is still high. A `0` bit is a short pulse because when the delayed clock signal goes high, the data signal is already low. We can modify the pulse widths p_1, p_2, p_3, and p_4 by changing the number of `NOP`s on the Teensy microcontroller during each of these intervals[2]. The nominal values are $p_1 = 80$, $p_2 = 80$, $p_3 = 2$ or 3, and $p_4 = 80$. The trade-off regarding the length of p_3 is that a longer pulse is more likely to be received by SCµM's optical receiver, but the length of p_3 cannot exceed the delay between the optical data and clock signals, which is set by the hardware.

During bootloading, these two signals, `OPTICAL_CLK_RAW` and `OPTICAL_DATA_RAW`, are available as GPIO outputs[3]. More information on the optical bootloader can be found in [16].

[1] `https://github.com/tryuan99/scum-test-code/blob/79104269eef314236c36ccf035edfcc826b24d33/scm_v3c/teensy_uC_programmer/teensy_uC_programmer.ino`

[2] `https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/bootload/bootload.py#L91`

[3] `https://docs.google.com/spreadsheets/d/1aphqlyBsOSbV8ofCgYJlZNo2-GQ3CV6BmYxwak9xiTg/edit?usp=sharing`

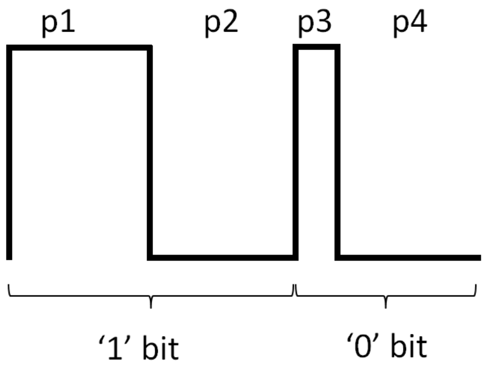


Figure 2.3: Pulse widths of a `1` bit and a `0` bit as transmitted by the IR LED and received by the optical receiver.

If SCµM does not boot or the CRC check fails after bootloading, this is usually an indication that the alignment between the IR LED and SCµM's optical receiver is incorrect. Re-positioning the IR LED by trial and error is the usual debug path. Another possibility is to measure the current consumption of SCµM during the bootloading process. If the hard reset is triggered, then SCµM's power consumption should be significantly lower. Afterwards, SCµM turns on some of its LDOs to prepare for the frequency calibration, which should raise the current back up to around 2 mA or more, especially if the LC divider is on for LC frequency calibration.

Optical Frequency Calibration

After bootloading, SCµM performs a CRC check on the payload and commences frequency calibration. Here, the optical bootloader now transmits many interrupt frames until SCµM has calibrated its oscillators. Every 100 ms, the IR LED first sends preamble bytes followed by the sequence $[221, 176, 231, 47]$ and finally more `1` bits to clock through the data. This four-byte sequence triggers the optical start frame delimiter (SFD) interrupt[4] on SCµM, allowing us to calibrate the frequencies of SCµM's on-board oscillators.

During frequency calibration, we measure the frequencies of the on-board oscillators using counters and tune their frequencies by adjusting their DACs. We tune the 2 MHz RC oscillator, the 64 MHz IF RC oscillator divided down to 16 MHz, and the 20 MHz HCLK RC oscillator. The 32 kHz RC oscillator does not have a tunable resistor DAC. We describe how to calibrate the 2.4 GHz LC oscillator in Section 4.3.

If these SFD interrupts are not triggered consistently or not triggered at all, this is usually indicative of a misalignment between the IR LED and SCµM's optical receiver. Other times,

[4]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/optical.c#L122`

if SCµM never exits frequency calibration, this could indicate that SCµM has not received 25 start frames yet before the IR LED finished sending start frames. To correct this issue, we can increase the number of start frames transmitted by the IR LED[5].

2.3 Oscillators

In this section, we give a brief overview of the on-chip oscillators on SCµM and describe how their frequencies can be tuned. We also describe the default use cases of these oscillators and characterize their frequency stability.

Most of the configurations on SCµM are set by software immediately after bootloading prior to optical frequency calibration. These configurations are stored in the analog scan chain (ASC), a 1,200 bit sequence, that is written into a shift register using `analog_scan_chain_write` and then loaded using `analog_scan_load`[6]. Examples of these static configurations are the clock divide ratios and the clock sources.

During execution of the application, dynamic configurations can be changed by modifying the memory-mapped `ANALOG_CFG_REG__i` registers. Examples of these dynamic configurations include the frequency tuning codes and the LDOs. These `ANALOG_CFG_REG__i` registers actually point to different registers when reading and writing. When reading from the `ANALOG_CFG_REG__i` registers, we read from the `analog_rdata` bus, and when writing to these registers, we write to the `analog_cfg` bus[7].

The frequencies of the oscillators can be estimated using internal memory-mapped counter registers in the Cortex M0 microprocessor that increment on the positive edge of the respective clocks. After each of the 100 ms SFD interrupts during optical frequency calibration, we tune the bits controlling the resistor (or capacitor for the LC oscillator) DACs depending on the number of counts within the last 100 ms. For the software described here, 25 optical SFDs must be received by the mote before calibration is complete.

The oscillator frequencies are sensitive to supply voltage, which, while locally regulated, varies from chip to chip. They also have a high temperature coefficient as they were not designed to be temperature-independent, e.g., the 2 MHz has a temperature coefficient of around 160 ppm/°C [16] and the 2.4 GHz oscillator has a temperature coefficient of around −40 ppm/°C [9]. All of these variations over voltage and temperature require each SCµM chip to be calibrated individually during programming.

[5]https://github.com/tryuan99/scum-test-code/blob/79104269eef314236c36ccf035edfcc826b24d33/scm_v3c/teensy_uC_programmer/teensy_uC_programmer.ino#L1399

[6]https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/scm3c_hw_interface.c#L1297

[7]https://docs.google.com/spreadsheets/d/1aphqlyBsOSbV8ofCgYJlZNo2-GQ3CV6BmYxwak9xiTg/edit?usp=sharing

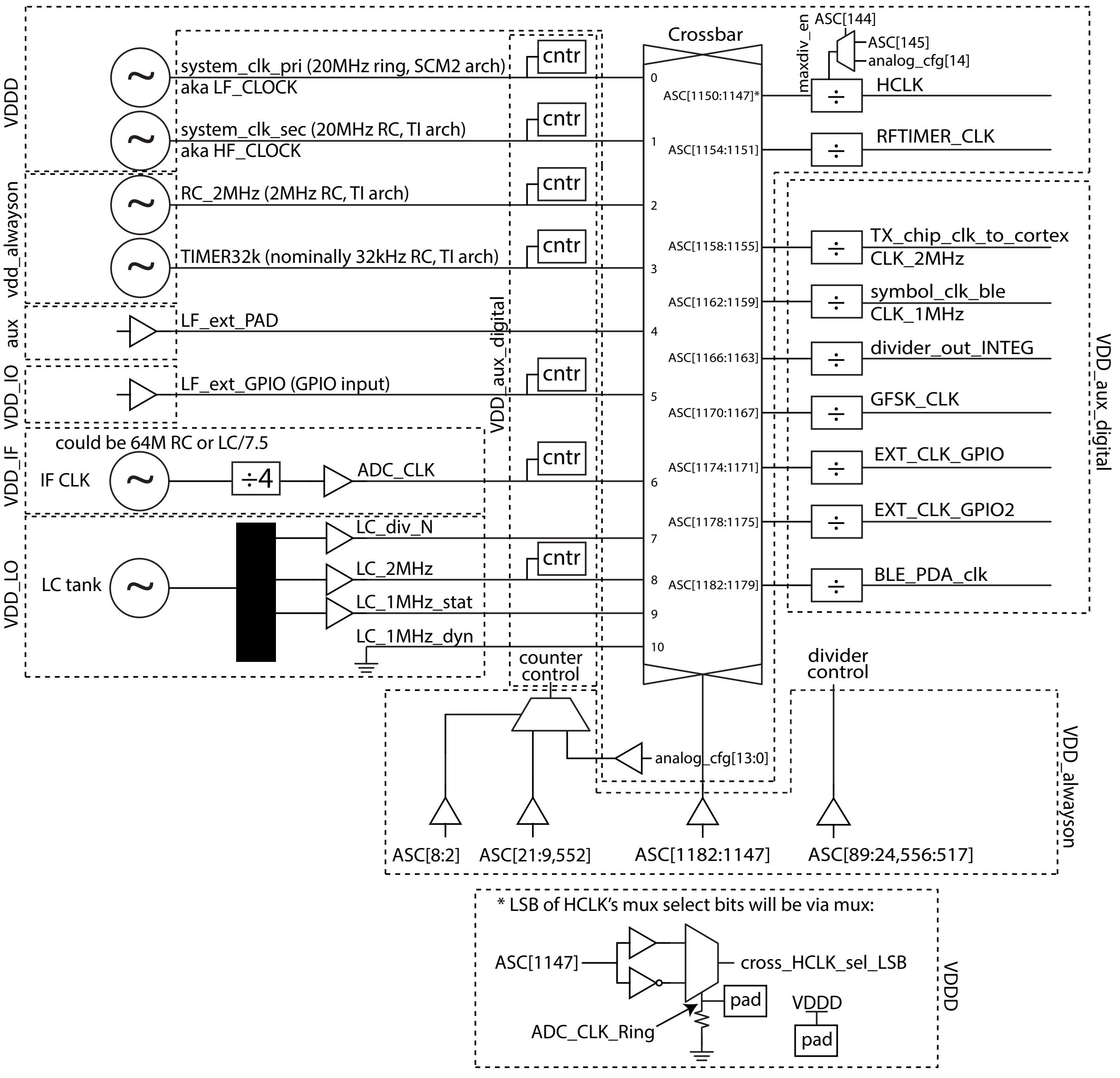


Figure 2.4: Clock diagram of SCµM3C.

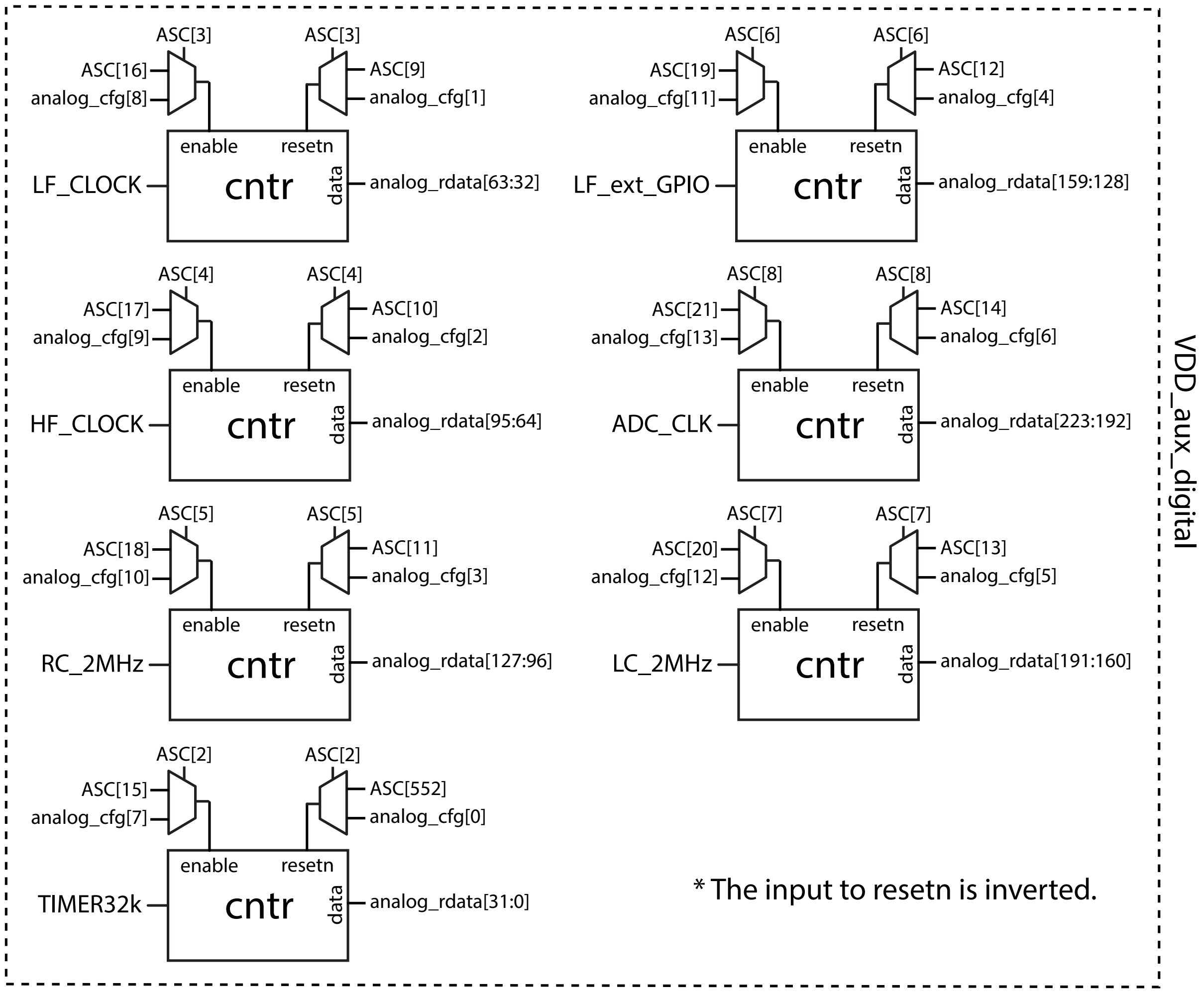


Figure 2.5: On-chip oscillator counters on SCµM3C.

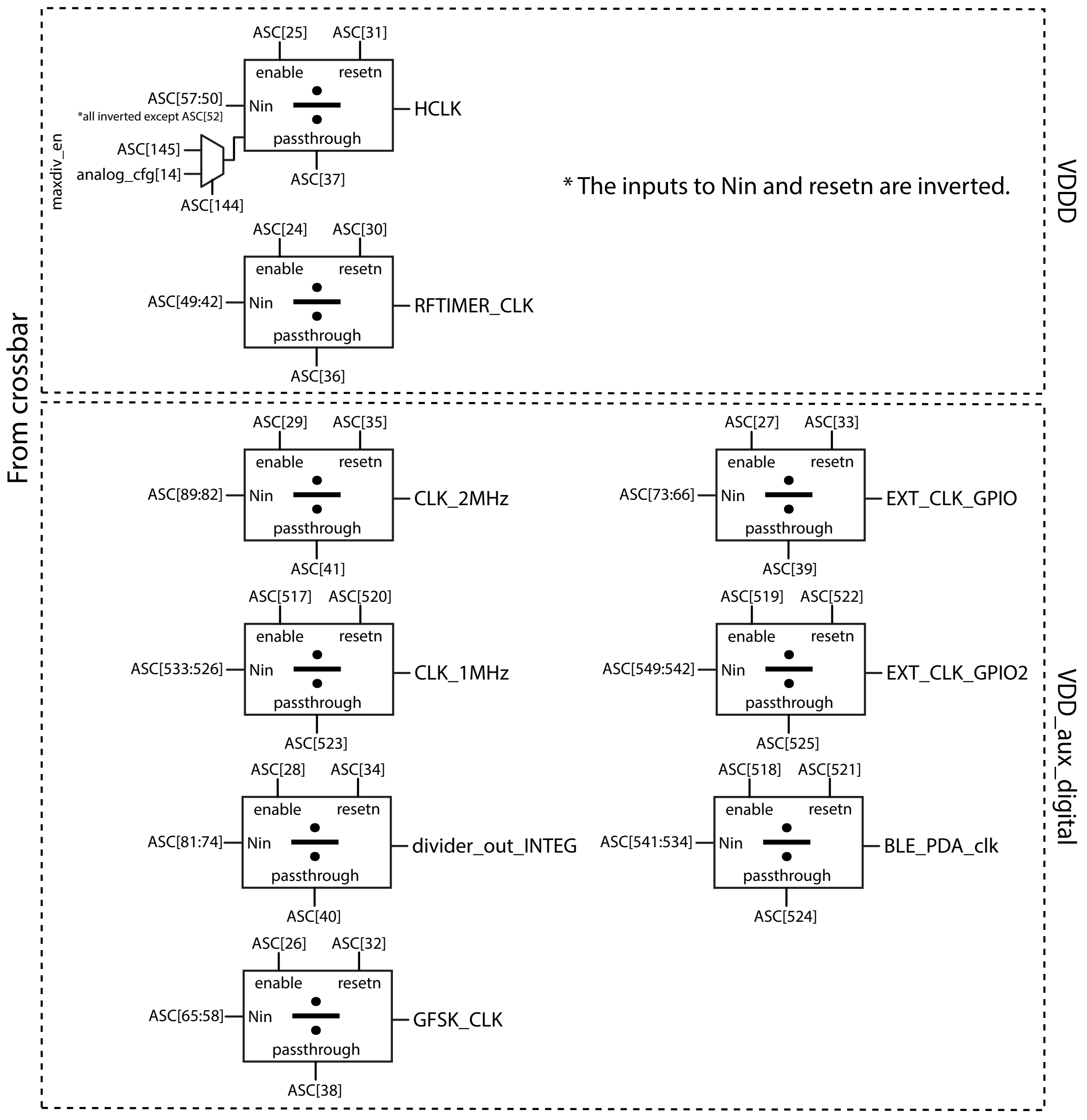

From crossbar
VDDD
VDD_aux_digital
maxdiv_en
ASC[145]
analog_cfg[14]
ASC[144]
ASC[57:50]
*all inverted except ASC[52]
ASC[25] ASC[31] enable resetn Nin passthrough ASC[37] HCLK
* The inputs to Nin and resetn are inverted.
ASC[24] ASC[30] ASC[49:42] ASC[36] RFTIMER_CLK
ASC[29] ASC[35] ASC[89:82] ASC[41] CLK_2MHz
ASC[27] ASC[33] ASC[73:66] ASC[39] EXT_CLK_GPIO
ASC[517] ASC[520] ASC[533:526] ASC[523] CLK_1MHz
ASC[519] ASC[522] ASC[549:542] ASC[525] EXT_CLK_GPIO2
ASC[28] ASC[34] ASC[81:74] ASC[40] divider_out_INTEG
ASC[518] ASC[521] ASC[541:534] ASC[524] BLE_PDA_clk
ASC[26] ASC[32] ASC[65:58] ASC[38] GFSK_CLK

Figure 2.6: Clock dividers on SCμM3C.

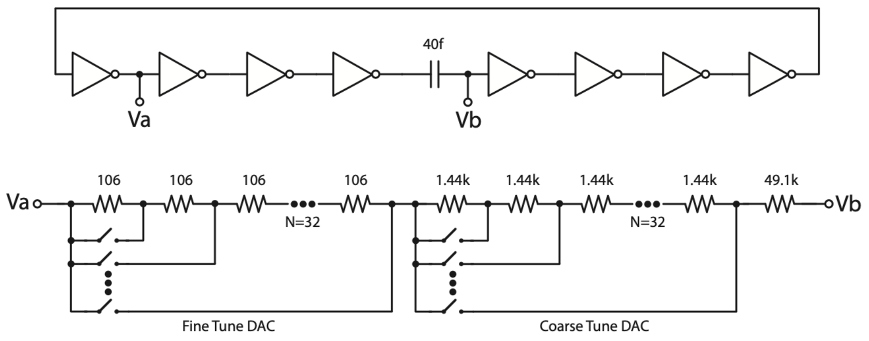


Figure 2.7: The 2 MHz RC oscillator circuit schematic as presented in [16]. The 32 kHz RC oscillator circuit does not have a tunable resistor DAC.

20 MHz **HF_CLOCK RC Oscillator**

The 20 MHz HF_CLOCK oscillator is an RC oscillator primarily used as the source for HCLK and the RFTimer clock discussed below. It uses a circuit topology similar to the circuit in [11], as shown in Figure 2.7. Notably, it has a 10-bit tunable resistor DAC, split into a 5-bit coarse and a 5-bit fine tune DAC, so that its frequency can be adjusted during the optical frequency calibration[8]. Setting `ASC[1147] = 1b'1`[9] sets HF_CLOCK to be the source for HCLK, and setting `ASC[1151] = 1b'1`[10] sets HF_CLOCK to be the source for RFTimer.

HCLK is used to clock the Cortex M0 microprocessor and is divided down from the HF_CLOCK oscillator by setting `ASC[57:50]` to the appropriate divide ratio. Note that `ASC[52]` is inverted, so in order to achieve a divide ratio of 2, such that the HCLK frequency is 10 MHz, we would set `ASC[57:50] = 8b'00000110`. By default, `ASC[57:50] = 8b'0`, so the divide ratio is 4, and the nominal frequency of HCLK is 5 MHz at boot.

The RFTimer, described in more detail in Sections 3.35 and 4.3.1 of [10], is a clock divided down from HF_CLOCK in the software described. Setting `ASC[36] = 1b'1` sets HF_CLOCK to pass through to the RFTimer, so that the RFTimer has a frequency of 20 MHz. Otherwise, the divide ratio between HF_CLOCK and the RFTimer can be set using `ASC[49:42]`. Note that all eight bits of `ASC[49:42]` are inverted. Nominally, `ASC[49:42] = 8b'11010111`[11], so the divide ratio is set to 40, meaning that the RFTimer frequency is 500 kHz.

[8]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/optical.c#L186`

[9]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/scm3c_hw_interface.c#L1166`

[10]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/scm3c_hw_interface.c#L1176`

[11]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/scm3c_hw_interface.c#L1182`

The RFTimer is usually used to trigger recurring interrupts within the code, e.g., to transmit a BLE packet every 500 ms. However, since RFTimer is divided down from HF_CLOCK, the frequency stability of the HF_CLOCK oscillator affects the accuracy of the RFTimer, as described below.

2 MHz **RC Oscillator**

The 2 MHz RC oscillator also uses a circuit topology similar to the circuit in [7]. Notably, the 2 MHz circuit, as described in [1] and shown in Figure 2.7, has a finely tunable resistor as it is the chipping clock for the chip's transmitter, so its frequency needs to be accurate to within ±50 ppm for BLE [1]. It also does not include the additional supply rejection circuitry from [7]. The frequency is tuned by adjusting a 10-bit resistor DAC containing 5 bits of coarse and 5 bits of fine adjustment during optical frequency calibration[12]. According to Section 5.13 of [16], it has a temperature coefficient of around 160 ppm/°C.

32 kHz **RC Oscillator**

The 32 kHz is similar to the 2 MHz RC oscillator, but it is intended to be a low-frequency timer and can be used as the on-chip Cortex M0 clock to conserve energy between higher-power radio operations [1]. Moreover, it does not have a tunable resistor DAC and was originally designed to have a temperature coefficient of zero.

64 MHz **IF RC Oscillator**

The 64 MHz IF RC oscillator is used to generate the 16 MHz IF sampling clock. The chip rate of an 802.15.4 packet is 2 MHz, so we sample each chip 8 times. 802.15.4 uses direct sequence spread spectrum (DSSS) to spread its signals, so that 8 chips correspond to one bit. The IF sampling clock's frequency is calibrated during optical frequency calibration[13].

2.4 GHz **LC Oscillator**

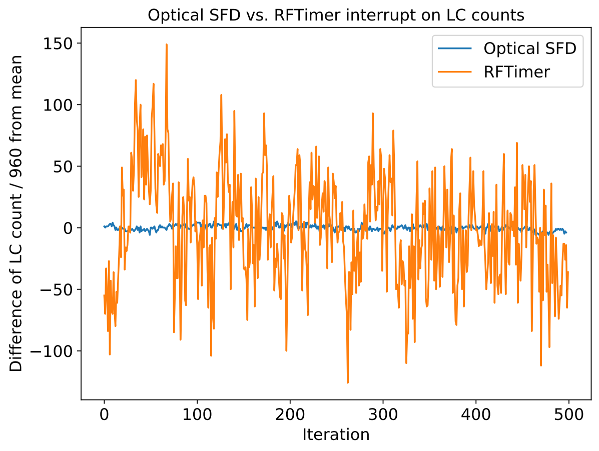
The 2.4 GHz LC tank is used as the local oscillator (LO) and is described in Section 2.4.

Frequency Stability

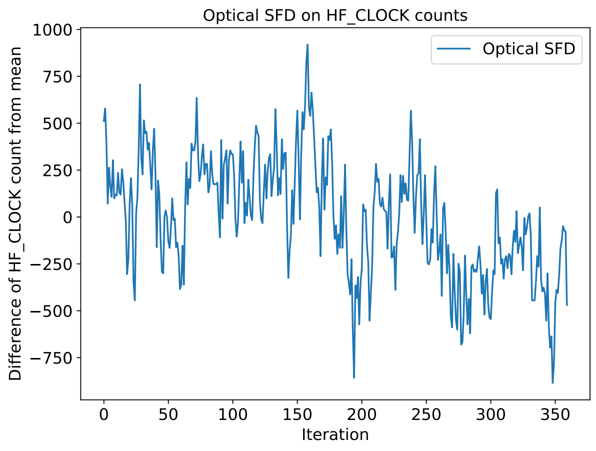
We characterized the frequency stability of the aforementioned oscillators at a fixed temperature by using the counter registers for each of the oscillators. We first used the IR LED with the Teensy 3.6 microcontroller to transmit many 100 ms start frames over a period of between half a minute to a minute. Each of these start frames triggers an SFD interrupt

[12] `https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/optical.c#L236`

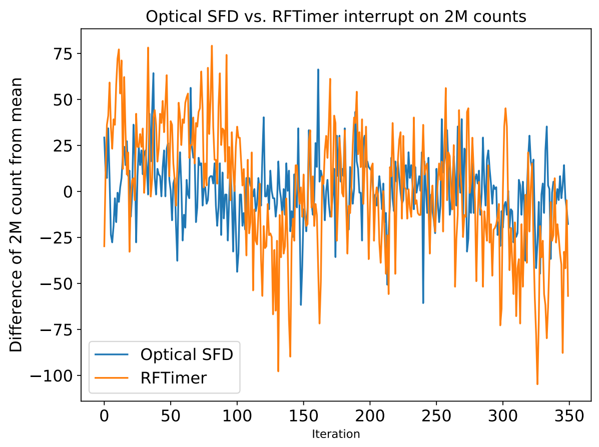
[13] `https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/optical.c#L269`



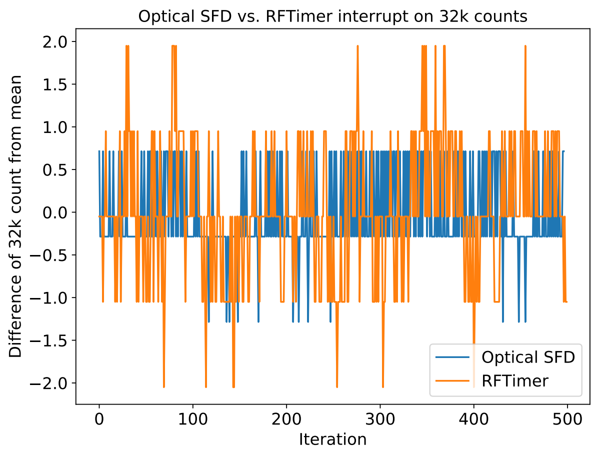
(a) 2.4 GHz LC oscillator.



(b) 20 MHz HF_CLOCK oscillator.



(c) 2 MHz oscillator.



(d) 32 kHz oscillator.

Figure 2.8: The frequency stability of some of the clocks on SCμM measured at room temperature. All frequency counts were recorded every 100 ms, either during an optical SFD interrupt or during an RFTimer interrupt. The optical SFD frames originating from the Teensy 3.6 microcontroller are fairly accurate. In contrast, since the RFTimer is divided down from the HF_CLOCK oscillator, RFTimer itself is noisy. We observe more jitter in the 100 ms period for the 2 MHz and 32 kHz oscillators than described in [1].

in SCμM, and in the interrupt handler, we recorded the number of counts for each of the oscillators. We also used the RFTimer to trigger 100 ms interrupts, in which we also recorded the number of counts for each of the oscillators. The results for the frequency stability of the oscillators at room temperature are shown in Figure 2.8.

In Figure 2.8a, note that the frequency counts using the RFTimer interrupts are much noisier than the frequency counts measured using the optical SFD interrupts. This has two implications. First, at a fixed temperature, the frequency of the 2.4 GHz LC oscillator does not drift much. Second, the 500 kHz RFTimer, derived from the 20 MHz HF_CLOCK oscillator, has around 200 ppm of RMS noise. This is substantiated by Figure 2.8b, where HF_CLOCK has an RMS noise of around 200 ppm as well. It is important to note that during normal operation without a Teensy 3.6 microcontroller, the RFTimer is the most commonly used timer. However, it is subject to a frequency stability of 200 ppm, which should be considered when developing SCµM applications requiring an accurate on-chip timer.

In Figures 2.8c and 2.8d, both of the frequency counts show some noise regardless of the source of the 100 ms interrupts. The 2 MHz oscillator has an RMS noise of around 100 ppm, which is higher than the 20 ppm of frequency stability observed in Figure 5 of [1]. Since the 32 kHz frequency counts seem to be dominated by quantization noise, we do not characterize its frequency stability, but in Figure 6 of [1], the 32 kHz oscillator has a frequency stability of around 20 ppm.

2.4 2.4 GHz LC Oscillator

The local oscillator (LO) consists of an LC tank whose frequency can be adjusted using a 15-bit tuning code, split into 3 5-bit values called the coarse, mid, and fine frequency tuning codes. Each of the coarse, mid, and fine codes is used to tune a 5-bit capacitive DAC, as shown in Figure 2.9. Each fine code corresponds to a change of around 100 kHz in the LO frequency, each mid code corresponds to a change of around 700 kHz, and each coarse code corresponds to a change of around 11 MHz [9]. The coarse, mid, and fine codes are set in software by calling `LC_FREQCHANGE`[14].

Figure 2.10 shows the 802.15.4 modulation logic schematic that is used to modulate the 802.15.4 modulation capacitor, where `{b3, b2, b1, b0} = ASC[996:999]`. Figure 2.11 shows the BLE modulation schematic that is used for BLE transmission. Note that for this project, we use FSK modulation for BLE packets.

To measure the frequency of the LC oscillator, we use the local oscillator divider, a top-level diagram of which is shown in Figure 2.12 and described in more detail in Section 5.5 of [9]. We divide the local oscillator frequency by 960 and measure a counter every 100 ms as determined by the on-chip RFTimer interrupts. The results are shown in Figure 2.13 as we sweep through all possible 2^{15} tuning codes along with a plot of the LC frequency over coarse codes 23 and 24 only. Notably, the LC frequency is not monotonic with respect to the 15-bit frequency tuning code. Instead, when the fine code carries over to the mid code, there is a small drop in the frequency of around 1.6 MHz, and when the mid code carries over to the coarse code, there is a larger drop in the frequency of 7.7 MHz to 11.5 MHz, as shown in Figure 2.14. These frequency drops at the edges of the fine and mid codes vary across

[14]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/scm3c_hw_interface.c#L1489`

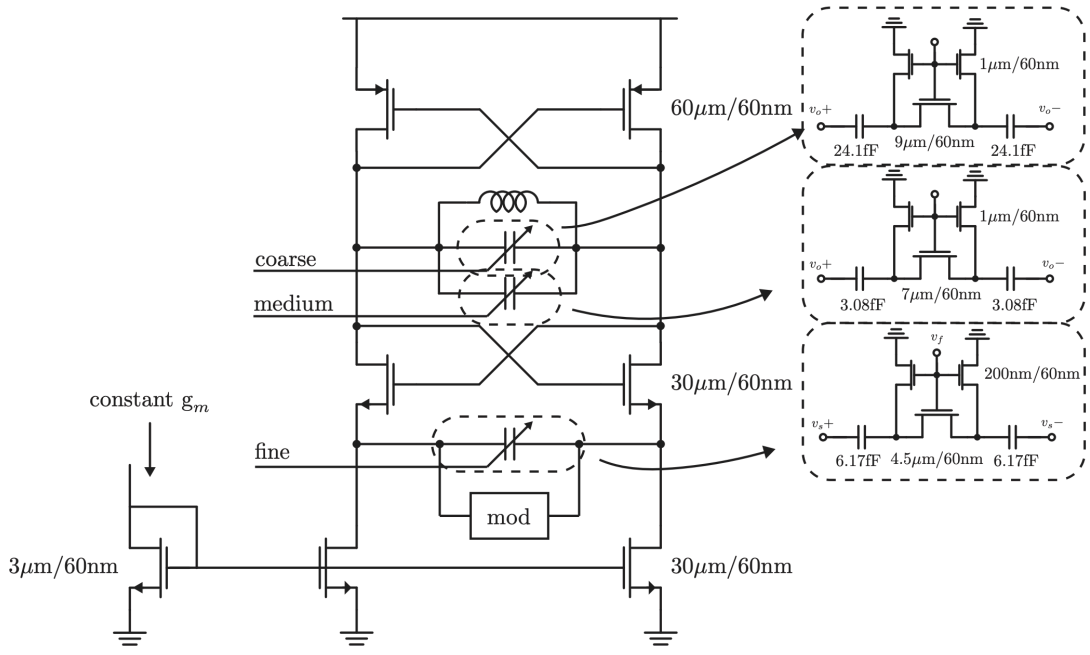


Figure 2.9: Local oscillator schematic with tuning and modulation as presented in [9].

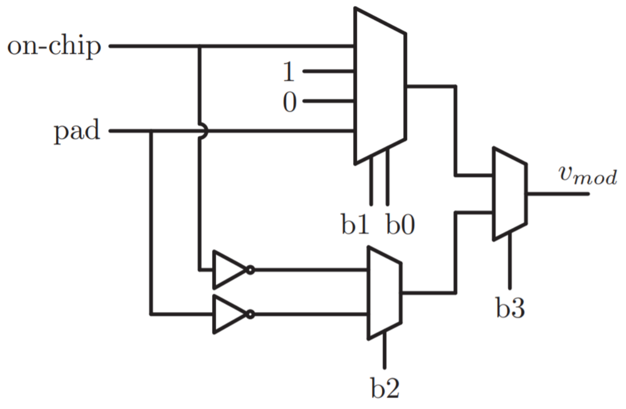


Figure 2.10: 802.15.4 modulation logic schematic as presented in [9].

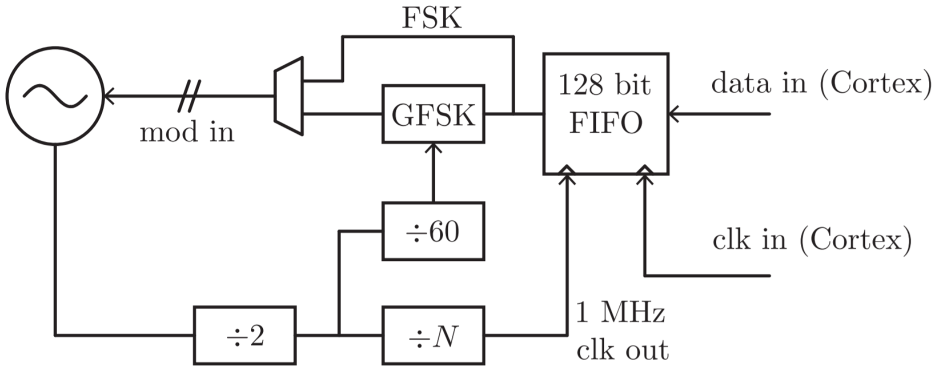


Figure 2.11: BLE modulation schematic as presented in [9].

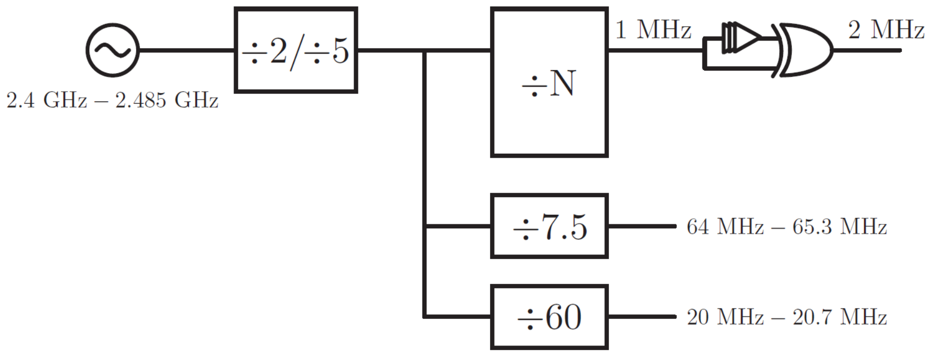


Figure 2.12: Top-level local oscillator divider block diagram as presented in [9].

the tuning code and are difficult to characterize, changing across current settings, between chips, across temperature, and from one end of the band to the other.

Since the LC frequency as a function of the 15-bit frequency tuning code is non-monotonic and there are 2^{15} possible frequency tuning codes, it is difficulty to calibrate the LC frequency with only 25 SFD interrupts unless we have a good initial estimate of the frequency tuning code. Instead, we can tune the LC frequency by trial and error by using a spectrum analyzer, or we can use optical LC calibration as described in Section 4.3. Once the frequency tuning code of a particular SCµM chip for a particular frequency at a particular temperature has been found, it can be used as a good initial frequency tuning code for subsequent calibrations.

Note that the LC frequency varies with temperature and is chip-dependent. The frequency also drifts if the antenna load on SCµM is changed or if some LDOs, especially the one controlling the LC divider that draws a significant amount of current, are turned on or off.

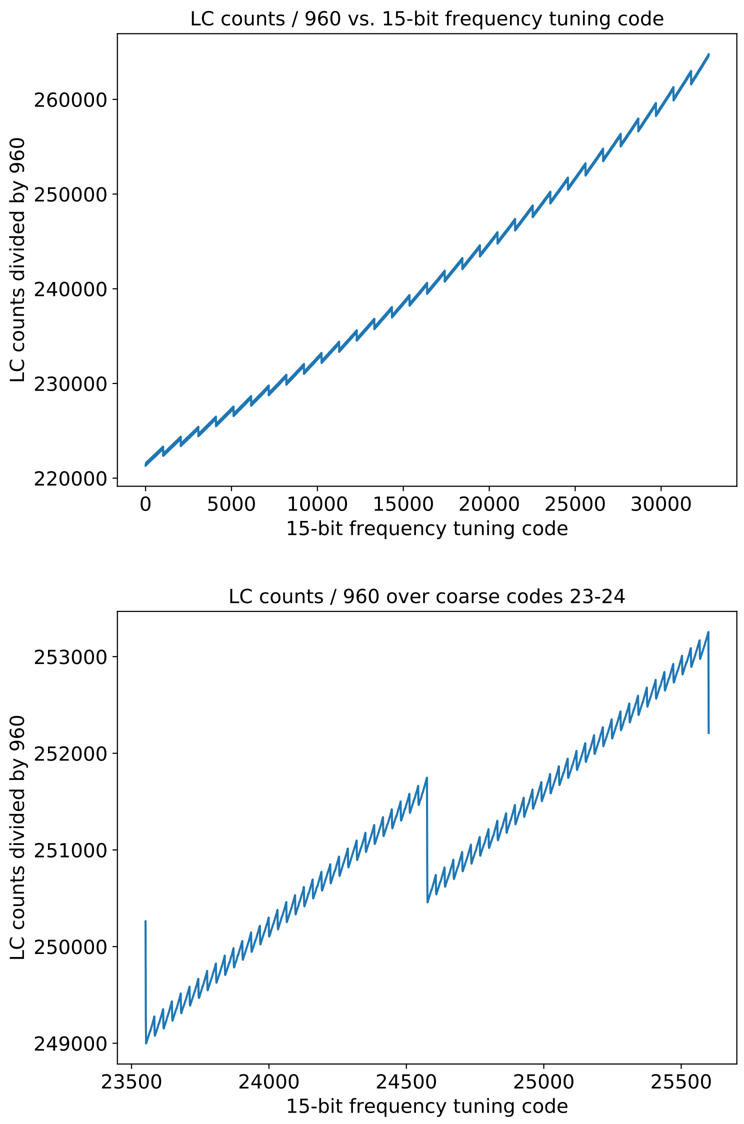


Figure 2.13: SCμM's LC frequency counts divided by 960 as a function of the 15-bit frequency tuning code. Note that the LC frequency is not monotonic with respect to the tuning code. There is a drop in the LC frequency when the fine code carries over to the mid code and a larger drop in the LC frequency when the mid code carries over to the coarse code. The LC counts were recorded every 100 ms.

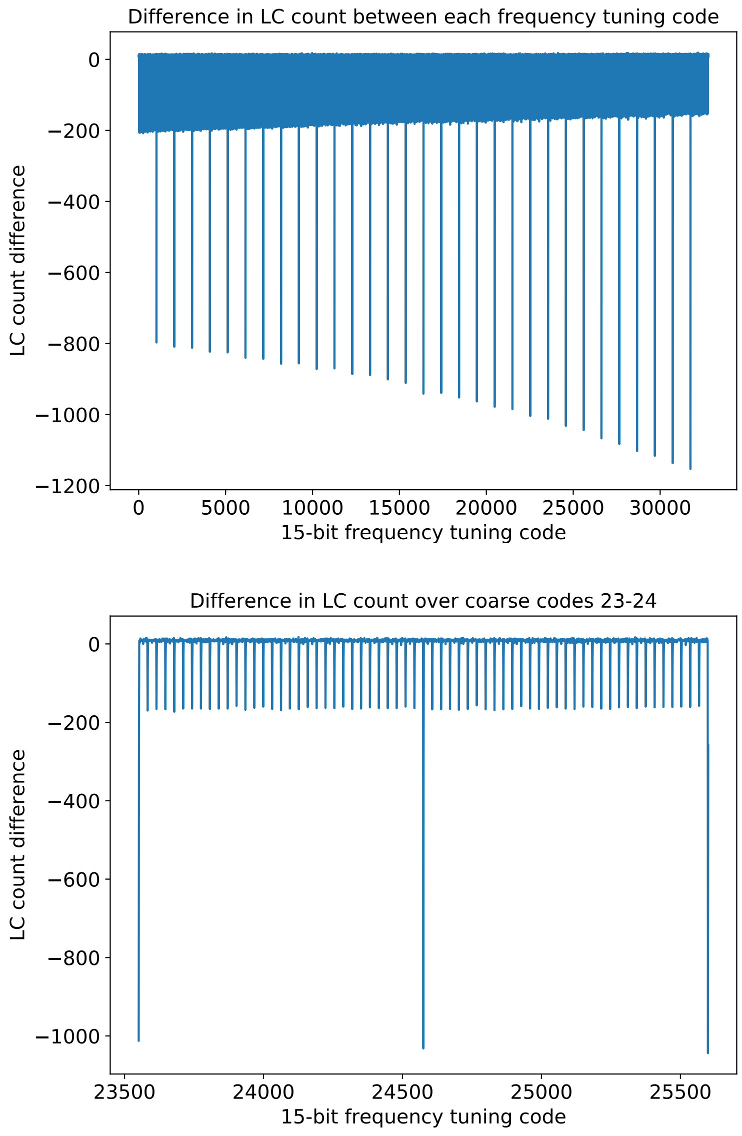


Figure 2.14: Difference in LC counts between successive 15-bit LC frequency tuning codes as shown in Figure 2.13. When the fine code rolls over to the mid code, there is a frequency drop of around 170 counts, which corresponds to a frequency drop of around 1.6 MHz. When the mid code rolls over the coarse code, there is a frequency drop of around 800 to 1,200 counts, which corresponds to a frequency drop of around 7.7 MHz to 11.5 MHz.

LC Monotonic Function

In order to facilitate LC frequency calibration, we attempted to create a monotonic function that maps an `LC_code` to a 15-bit frequency tuning code. There are two different approaches we attempted to create a monotonic function. First, the naive solution is to hard-code all of the 15-bit frequency tuning codes to use to cover the entire 2.4 GHz ISM band, such that their corresponding LC frequencies are monotonically increasing. The other approach, as described in Section 5.6 of [9], is that since there is some frequency overlap between consecutive mid codes and coarse codes, we need to find the fine code and the mid code, both less than 32, at which we would roll over earlier instead of at 32. Both of these approaches were first simulated using the frequency count data similar to the one presented in Figure 2.13 but with a shorter 40 ms RFTimer interrupt period.

Hard-coding Frequency Tuning Codes

The first approach to creating an LC monotonic function is by simply hard-coding all of the 15-bit frequency tuning codes into an array on SCµM's firmware and indexing into this array to change LC frequencies. After characterizing the LC frequency as a function of all 2^{15} frequency codes, we greedily selected the monotonic frequency codes, such that the frequency difference between successive monotonic frequency codes was at least 8 LC counts within a 40 ms interval, which corresponds to around 190 kHz. This gave us a total of 1,133 LC frequency tuning codes to use.

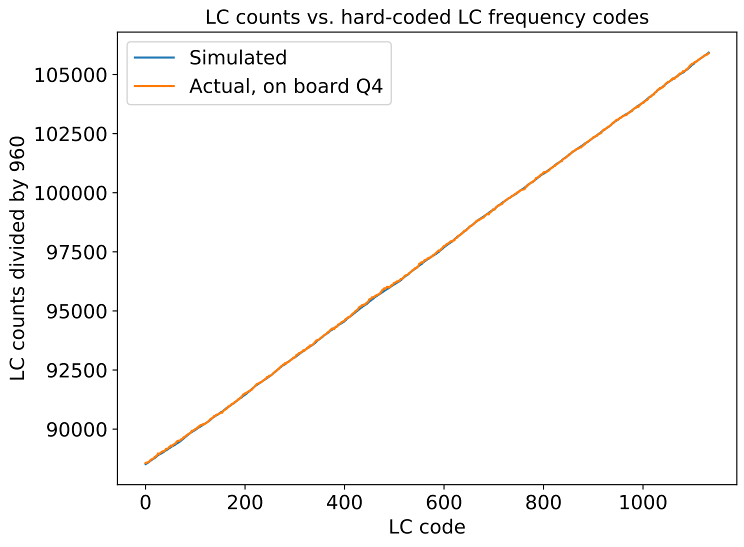
We then coded these 1,133 LC frequency tuning codes into an array on SCµM's firmware and swept through all of these 1,133 tuning codes in the array, measuring the corresponding LC counts. We set the RFTimer to trigger an interrupt every 40 ms, during which we recorded the LC counts and switched to the next monotonic frequency tuning code in the array. The resulting LC counts, both in simulation and on the actual SCµM development board Q4, are shown in Figure 2.15.

The corresponding LC count difference between successive hard-coded frequency codes is shown in Figure 2.16. In simulation, this hard-coding solution creates a strictly monotonic function of the LC frequency with respect to the 1,133 selected frequency tuning codes to use. However, when sweeping through all hard-coded frequency codes on board Q4, the LC frequency does not increase monotonically.

It is important to note that the 40 ms interrupts were triggered by the RFTimer. Comparing Figures 2.16 and 2.8a, this non-monotonicity could be caused by the frequency stability of the RFTimer clock. One disadvantage of using this approach to build a LC monotonic function is that storing 1,000 hard-coded frequency codes in the firmware requires around 4 kB, a considerable amount when SCµM only features 64 kB of program memory.

Rollover Fine and Mid Codes

Another approach is to observe that there is some frequency overlap between consecutive mid and coarse codes. Therefore, in order to build the LC monotonic function, we need to



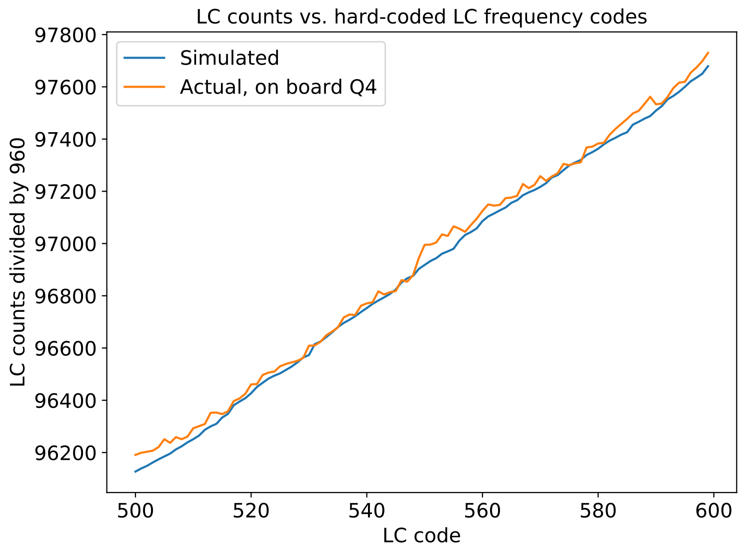


Figure 2.15: LC frequency counts as a function of the 1,133 hard-coded LC monotonic frequency codes, such that the difference between successive codes is at least 8 counts in simulation. The LC counts were recorded once every 40 ms for every LC code.

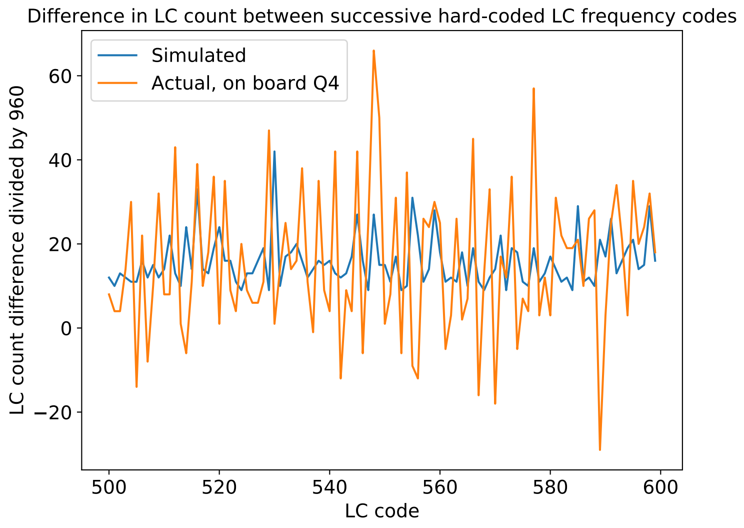


Figure 2.16: LC frequency count difference between successive hard-coded LC monotonic frequency codes within 40 ms.

find where to roll over from the fine or mid codes to the next mid or coarse code, respectively.

To implement this in simulation, we first found the range of coarse codes, such that the LC frequency is between 2.3 GHz and 2.5 GHz, thus covering the entire 2.4 GHz ISM band. For SCμM development board Q4, this corresponded to coarse codes 20 through 30.

Afterwards, for each of the coarse codes, we found the fine code, at which we would roll over to the next mid code. In other words, we found how many fine codes we would use for every mid code. For board Q4, we found the rollover fine codes to be `mid0_codes = [9, 8, 9, 9, 9, 9, 9, 10, 9, 10, 10]`.

Finally, we found how many frequency codes we would use for each coarse code before rolling over to the next coarse code. For board Q4, we round the rollover mid codes to be `coarse0_codes = [166, 148, 166, 166, 166, 166, 166, 185, 168, 185, 185]`. In other words, for the coarse code 20, we would use the following coarse, mid, and fine code triplets:

$$(20, 0, 0), (20, 0, 1), \ldots, (20, 0, 8), (20, 1, 0), \ldots, (20, 1, 8), \ldots, (20, 18, 0), \ldots (20, 18, 3)$$

to have a total of 166 frequency codes for coarse code 20.

This algorithm is described in Section 5.6 of [9]. After implementing this algorithm on SCμM development board Q4's firmware, we swept through all rollover LC monotonic codes and measured the LC counts for every LC monotonic code every 40 ms as determined by the RFTimer interrupts. The results are shown in Figure 2.17 with the corresponding LC count difference between successive LC codes in Figure 2.18.

This approach does get us a more monotonic LC frequency tuning function, even with the presence of noise in the RFTimer clock. Another benefit of this approach compared to

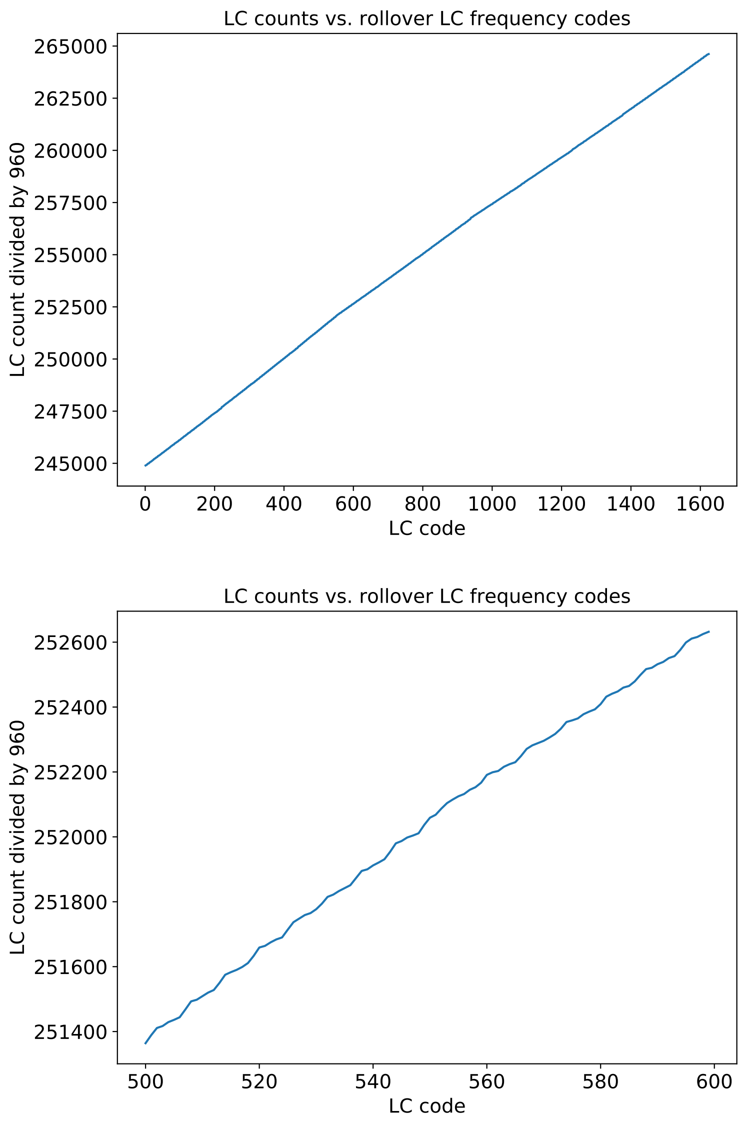


Figure 2.17: LC frequency counts as a function of the rollover LC monotonic codes. The LC counts were recorded once every 40 ms for every LC code.

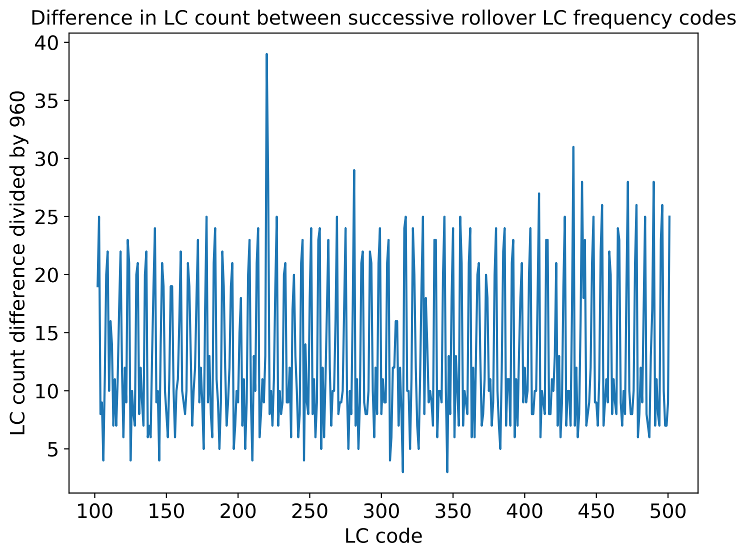


Figure 2.18: LC frequency count difference between successive rollover LC monotonic codes within 40 ms.

the hard-coding approach is that this only requires us to store two integers, one `mid0_code` and one `coarse0_code`, for every coarse code. However, the LC monotonic function is still not completely linear.

In this section, we showed two approaches at creating an LC monotonic function. Future work will involve characterizing the LC frequency using a more accurate timer than RFTimer and applying these approaches to the more accurate LC counts. In Sections 4.4 and 4.5, where we compensate the LC frequency using temperature and RX IF feedback, we resort to compensating the frequency only across the 32 fine codes, keeping the mid and coarse codes fixed, in order to not run into issues resulting from the non-monotonicity of the LC frequency tuning codes.

LC Frequency Over Temperature

Another important characteristic to note is the frequency drift of the LC oscillator as the temperature changes. In the absence of temperature changes, the LC tank has a frequency stability better than ±40 ppm [15].

In Figure 2.19, we measured the number of LC counts divided by 960 over a 500 ms interval as the ambient temperature was varied. We used the RFTimer, which has around 200 ppm of RMS noise, to generate the 500 ms intervals. SCµM was wirebonded onto a development board and placed in a TestEquity Model 107 temperature chamber. Starting from room temperature, we decreased the temperature down to 5 °C, then increased it to 80 °C, and finally decreased it back to room temperature. We measured the temperature in

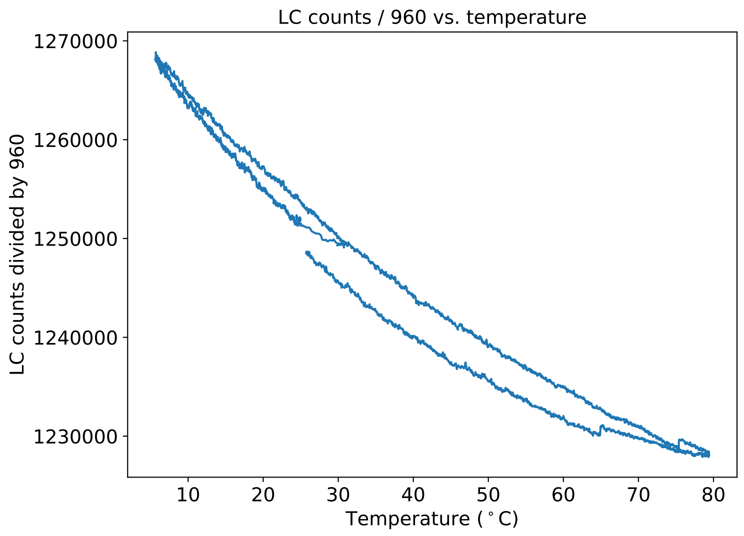


Figure 2.19: The LC frequency counts divided by 960 over temperature at a fixed frequency tuning code. The counts were recorded every 500 ms using the RFTimer interrupts. Note the hysteresis caused by the thermal mass of the development board to which the SCμM chip was wirebonded.

the chamber using a SparkFun TMP102 digital temperature sensor connected to a Teensy 3.6 microcontroller placed next to SCμM. Since the development board has a large metal ground plane, it has a large thermal mass. Therefore, due to the high temperature ramp rate relative to the board's thermal mass, we observe some hysteresis in the measurements.

According to Figure 5.30 of [9], the LC oscillator has a temperature coefficient of around -40 ppm/°C, much larger than the very small temperature coefficient seen in crystal oscillators that vary within ± 40 ppm across the entire temperature range. In order to compensate the LC frequency to allow SCμM to transceive 802.15.4 and BLE packets over temperature variations, we describe two possible methods in Chapter 4.

The temperature coefficient seen in Figure 2.19 seems much larger than the one observed in [9] because the RFTimer, sourced from the HF_CLOCK oscillator, was used to generate the 500 ms interrupts. As described in Section 2.3, the HF_CLOCK oscillator has around 200 ppm of RMS noise and an unspecified temperature coefficient of its own, which affects the precision of the LC frequency counts.

2.5 Radio

With a standards-compatible radio, SCμM is able to transmit and receive 802.15.4 and transmit BLE packets from off-the-shelf devices. In this section, we describe how to configure SCμM to transmit and receive 802.15.4 or transmit BLE packets and how the LDOs are turned on and off. We also characterize the radiation profile of a SCµM chip wirebonded onto a small PCB as shown in Figure 2.2 but without an antenna.

As described in Sections 3.25 and 4.3.2 of [10], the RFcontroller is the interface between the Cortex M0 microprocessor and the radio. The RFcontroller implements the finite state machines, the spreader and despreader, and the FIFOs for both TX and RX. However, as the radio was originally designed as a 802.15.4 transceiver, it does not fully support BLE packets directly.

The chip rate for 802.15.4 is 2 MHz, so during 802.15.4 transmission, we use the 2 MHz chipping clock to convert the packet data into bits for transmission. To load a packet into the TX FIFO, we call `radio_loadPacket`[15]. We then turn on the LDOs using the function `radio_txEnable`[16] and transmit the 802.15.4 packet by calling `radio_txNow`[17]. Furthermore, for 802.15.4, we tune the LO frequency to be 500 kHz above the intended channel frequency since it has an FM modulation frequency spacing of 1 MHz. For example, to transmit on channel 11 (2.405 GHz), we need to tune the radio to 2.4055 GHz.

In contrast, the chip rate for BLE is 1 MHz $\pm$ 40 ppm, so we cannot use the FIFO in the RFcontroller. Instead, we use a separate asynchronous FIFO, whose bits are clocked out at 1 MHz using a divided version of the 2 MHz chipping clock. We also set the LO frequency to be 250 kHz below the intended channel frequency because the FM modulation frequency spacing for BLE is 500 kHz, so for channel 37 (2.402 GHz), the target LO frequency is 2.401 75 GHz. Lastly, unlike 802.15.4 that uses minimum-shift keying (MSK), BLE standards require Gaussian frequency-shift keying (GFSK). However, on SCµM, we are only transmitting BLE packets with FSK instead of GFSK, but this seems to be sufficient for off-the-shelf BLE devices, such as phones, to receive BLE packets from SCµM. More details on BLE transmission on SCµM can be found in Section 4.2.

During RX, we tune the LO frequency to be 2.5 MHz below the target frequency. To receive 802.15.4 packets on channel 11 (2.405 GHz), we thus tune the LO frequency to 2.4025 GHz. The incoming signal is first down-converted to an intermediate frequency (IF) of 2.5 MHz. We use the 64 MHz IF clock divided down to 16 MHz to sample the down-converted signal. 802.15.4 uses direct sequence spread spectrum (DSSS) to spread its signals with a chip rate of 2 MHz, so we sample each chip eight times. Groups of 32 chips correspond to 4 bit symbols in the packet data.

For 802.15.4 RX, we use a matched filter initialized with the function `radio_init_rx_MF`[18]. To receive a packet, we first call `radio_rxEnable`[19] to turn on the appropriate LDOs fol-

[15]https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/radio.c#L179

[16]https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/radio.c#L192

[17]https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/radio.c#L208

[18]https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/scm3c_hw_interface.c#L709

[19]https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/radio.c#L215

lowed by `radio_rxNow`[20] to start the RX FSM. When a packet has been received, we can call `radio_getReceivedFrame`[21] to read the packet contents, the packet length, the RSSI, and the LQI error rate from the digital baseband [16].

The RFcontroller's RX finite state machine is only compatible with 802.15.4 packets. For BLE packets, we can use the zero crossing counter (ZCC) block to recover the data and clock of the BLE packet. We still tune the LO frequency to be 2.5 MHz below the channel frequency, so to receive BLE packets on channel 37 (2.402 GHz), we set the LO frequency to 2.3995 GHz. SCµM could receive BLE packets transmitted by other SCµMs, but it could not receive BLE packets transmitted by a smartphone. This is most likely because SCµM's BLE packets are only modulated with FSK instead of GFSK. More details on BLE RX are described in Chapter 5.

LDOs

The LDOs are controlled by the memory-mapped register `ANALOG_CFG_REG__10` and are usually turned on by calling `radio_txEnable` and `radio_rxEnable`. The most important LDOs for radio operation are the local oscillator (LO) LDO, the intermediate frequency (IF) LDO, the power amplifier (PA) LDO, and the LC divider (DIV) LDO.

For radio transmission, the LO and PA LDOs should be turned on. The DIV LDO can be optionally turned on as well if the LC frequency needs to be counted. For radio reception, the LO and IF LDOs should be turned on with the DIV LDO optionally on.

Note that turning on any one of the LDOs changes the current and will affect the LO frequency. For example, if LC frequency calibration is performed during optical calibration with the LC divider on, turning the LC divider off during normal radio operation to conserve power will induce a frequency shift in the LO.

The current settings of these LDOs can be set by calling `set_PA_supply`, `set_LO_supply`, and `set_DIV_supply`, respectively. In previous version of the SCµM's firmware, there was a bug in these functions that caused them to modify the incorrect ASC bits[22].

Radiation Characterization

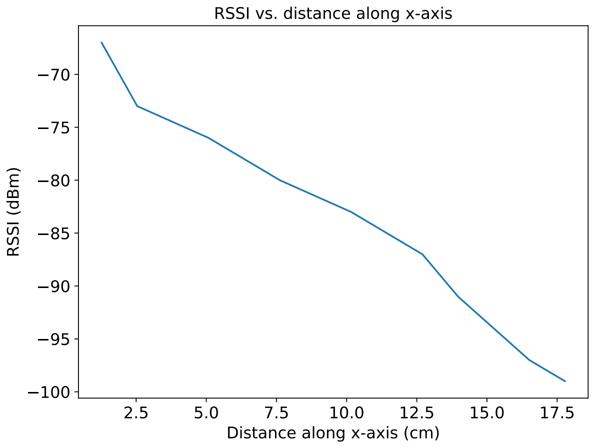
We now characterize the radiation profile of SCµM's radio without an antenna. For this experiment, we wirebonded SCµM onto a small PCB as shown in Figure 2.2. Unlike as shown in the figure, we did not wirebond an antenna onto SCµM, and we did not cover SCµM with epoxy.

We programmed SCµM to transmit 802.15.4 packets, and we used an OpenMote at variable distance away from SCµM to read the RSSI of the received packets. Afterwards,

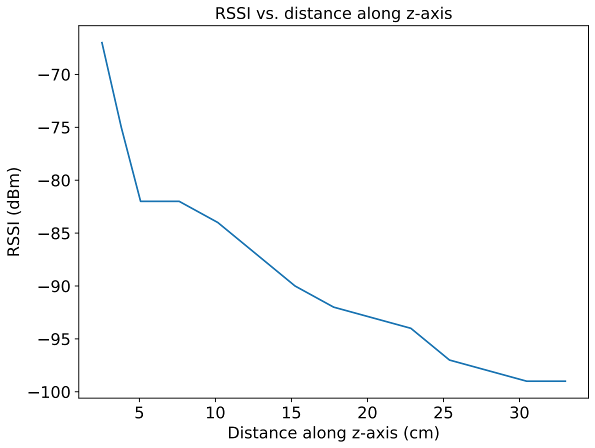
[20] `https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/radio.c#L243`

[21] `https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/radio.c#L253`

[22] `https://github.com/PisterLab/scum-test-code/pull/19`



(a) Distance along the x-axis.



(b) Distance along the z-axis.

Figure 2.20: RSSI of 802.15.4 packets transmitted from a SCµM chip wirebonded to a PCB without an antenna. The x-axis denotes the axis parallel to the PCB plane, and the z-axis denotes the axis orthogonal to the PCB plane.

we measured the RSSI along the x-axis, i.e., the axis parallel to the PCB plane, and along the z-axis, i.e., the axis pointing orthogonally out of the PCB plane. The results are shown Figure 2.20.

The range at which the OpenMote could receive SCµM's 802.15.4 packets depends on the distance and on the direction from SCµM. For example, we found that at an angle of approximately 20° from the vertical z-axis, the OpenMote could receive 802.15.4 packets with an RSSI of $-97\,\text{dBm}$ at a distance of around $122\,\text{cm}$, much greater than the range in either the x-axis or z-axis.

For comparison, Figure 5.38 of [9] shows the RSSI of BLE packets transmitted by SCµM with a wirebonded and a rubber ducky antenna. As expected, due to the presence of an antenna, the RSSI is higher at similar distance. However, radiation measurements are very imprecise due to external factors. For example, in the aforementioned experiment, tilting the antenna of the OpenMote changes the RSSI value, and the OpenMote can pick up packets from the radiating power cables as well.

Chapter 3

Temperature Estimation

Obtaining reliable temperature data is important for many IoT applications because of the dependence of many devices on the ambient temperature. For SCµM specifically, even a rough temperature estimate allows for temperature compensation, thus permitting the mote to be deployed in various environments. Most importantly, this allows SCµM to operate as a a small, low-cost IoT temperature sensor, enabling many exciting applications, such as a small body wearable to track the temperature on different parts of the body. However, it is challenging to keep both the size and the cost of such temperature sensors low.

In [13], previous temperature compensation was done using network-based calibration that tracks the intermediate frequency (IF). An external OpenMote beacon was programmed to periodically transmit 802.15.4 frames every 125 ms, and when SCµM received each frame, the IF frequency offset was determined in the clock and data recovery module and was then used to finely tune the RF frequency.

Another method described in [12] to allow channel hopping over varying temperature was by performing an offline characterization and an online network-based calibration of the LO frequency. In the offline stage, the authors swept through all of the LC frequency tuning settings and mapped these settings to the measured LC frequency. They then developed two linearization models, a recursive least-squares (RLS) model and a moving average (MA) model, that predict the tuning codes based on the current environment's temperature and on the 802.15.4 channel as demanded by the channel hopping schedule. In the online stage, initial frequency calibration for SCµM was done by using the IF frequency offset to ensure that the radio frequency accuracy was within 40 ppm per the 802.15.4 standards. The linearization models were then used to update the frequency tuning codes when the channel or the environment temperature changed and were themselves updated using network-based calibration from an OpenMote. However, when there is no external beacon providing a stream of 802.15.4 frames, such a network-based calibration is infeasible, so we propose a new method to estimate the ambient temperature using on-chip components in order to tune the LO frequency.

In this chapter, we present a method to calibrate SCµM for use as an IoT temperature sensor between 0 °C and 100 °C. The original purpose of finding a temperature estimate on

SCµM is to compensate the radio's LO frequency that is largely temperature-dependent, so that SCµM can operate over temperature variations. A one-time frequency calibration at room temperature is insufficient for crystal-free radio operation across temperature, so one possibility is to resort to periodic network compensation to calibrate the LO frequency over temperature.

Another solution is to estimate the ambient temperature using two existing running oscillators on the chip during radio operation similar to the method proposed in [11] and thus eliminate the need for periodic compensation after calibration. To find this temperature estimate, we use two clocks on SCµM that would already be running during radio operation anyway—the 32 kHz RC free-running oscillator similar to a crystal and the 2 MHz RC oscillator that is used as the chipping clock for transmitting 802.15.4 and BLE packets. We create a temperature sensor by finding a linear relationship between the ambient temperature and the ratio of these two clock frequencies. We show that a simple two-point temperature calibration is sufficient to find this linear model, and by averaging over a few temperature measurements, we observe a temperature error of less than 2 °C. We can then use SCµM as a tiny wireless temperature sensor by using the temperature estimate to calibrate the radio frequency oscillator.

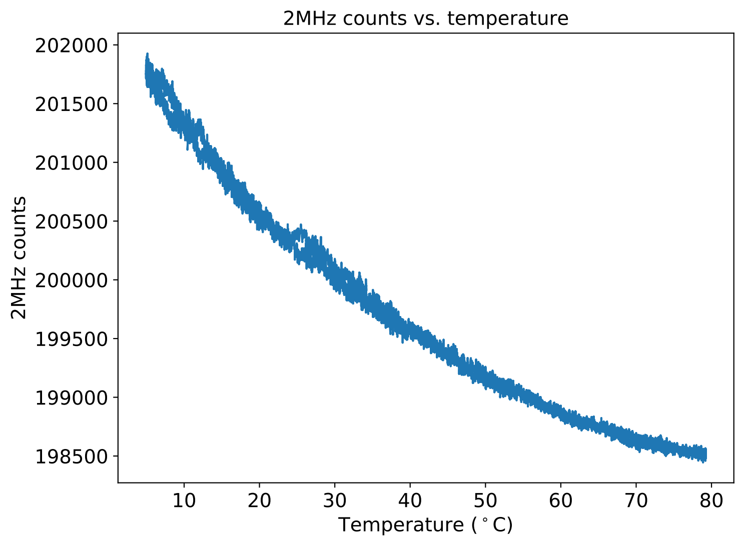
3.1 2 MHz / 32 kHz Frequencies Over Temperature

We use the 2 MHz and the 32 kHz RC oscillators on SCµM to calibrate it with respect to temperature. The 2 MHz oscillator is used as the chipping clock for transmitting 802.15.4 and BLE packets while the 32 kHz oscillator, similar to a crystal, can be used as a sleep timer for when SCµM is operating in networks implementing the 802.15.4 time synchronized channel hopping (TSCH) standards [4].

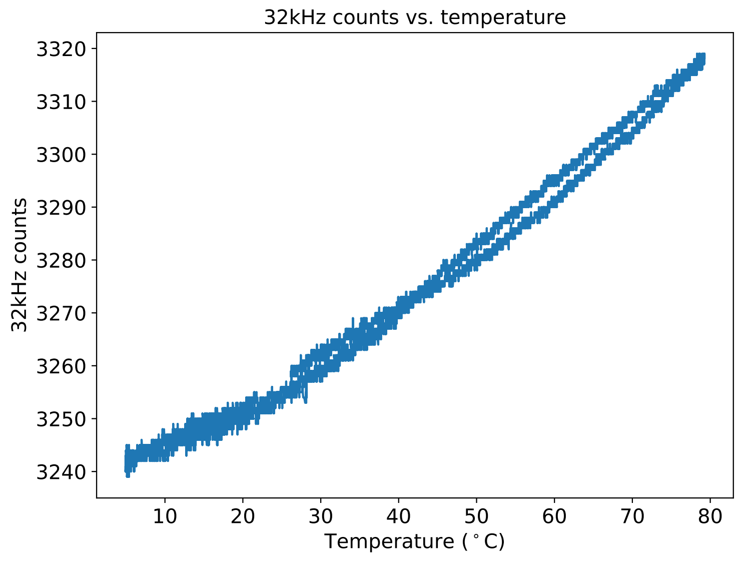
The frequencies of both of these RC oscillators have a non-linear relationship with temperature as plotted in Figure 3.1 on SCµM development board Q4. According to Section 5.13 of [16], the 2 MHz oscillator has a temperature coefficient of around 160 ppm/°C. Figure 3.1a shows a much larger temperature coefficient because the RFTimer, sourced from the HF_CLOCK oscillator, was used to generate the 100 ms interrupts. As described in Section 2.3, the HF_CLOCK oscillator has around 200 ppm of RMS noise and an unspecified temperature coefficient of its own, both of which affect the 2 MHz and 32 kHz frequency counts.

We measured these values by placing SCµM into a TestEquity Model 107 temperature chamber along with a reference temperature sensor that consisted of a SparkFun TMP102 digital temperature sensor connected to a Teensy 3.6 microcontroller as the reference temperature sensor.

While sweeping the temperature between 5 °C and 85 °C, we used the RFTimer, a 500 kHz clock divided down from the on-board 20 MHz HF_CLOCK oscillator, to trigger an interrupt approximately every 100 ms on SCµM. This is similar to optical calibration, but the interrupts originate from a noisier on-chip oscillator instead of a Teensy 3.6 microcontroller.



(a) 2 MHz frequency counts vs. RFTimer.



(b) 32 kHz frequency counts vs. RFTimer.

Figure 3.1: SCµM's 2 MHz and 32 kHz frequency counts vs. temperature. Every 100 ms, the RFTimer triggered an interrupt, and in the interrupt handler, we read the frequency counters of the two oscillators and logged them over UART.

Every 50,000 cycles of RFTimer, which corresponds to roughly 100 ms, we read the frequency counts of both the 2 MHz and the 32 kHz oscillators and reset the counters. We then logged the frequency counts as well as the reference temperature to a computer over UART.

Starting from an initial temperature of 25 °C, we programmed the temperature chamber to linearly decrease the temperature to 5 °C, then increase it to 80 °C, and finally decrease it back down to room temperature at a rate of ±1.5 °C/min. Since the SCµM chip was placed on a development PCB with a large metal ground plane, the board's large thermal mass caused some hysteresis in the frequency count measurements if the temperature ramp rate was too high.

When we divided the 2 MHz frequency count by the 32 kHz frequency count, we observed that the frequency ratio was approximately linear to temperature, as shown in Figure 3.2a overlaid with a linear regression line. The ratio values used in Figure 3.2a were actually calculated on the SCµM chip and then printed to a computer over UART along with the 2 MHz and 32 kHz frequency counts. Since SCµM has no floating point unit, a fixed point library[1] was implemented on the chip for floating point operations, especially the ratio calculation.

As mentioned previously, the 100 ms interrupts were generated by RFTimer, which is sourced from HF_CLOCK with an RMS noise of 200 ppm and some unspecified temperature coefficient. However, since we are calculating the ratio of the 2 MHz and the 32 kHz frequency counts, this removes the dependence of the ratio calculation on the HF_CLOCK's frequency stability and temperature coefficient. However, the hysteresis caused by the thermal mass of SCµM's underlying development board is still visible in Figure 3.2.

Nevertheless, we propose a linear model to predict the temperature based on the frequency ratio of the two oscillators. Calibrating SCµM to its ambient temperature is thus equivalent to determining this linear relationship.

Running this temperature sweep on multiple SCµM chips, we confirmed that, as expected, the coefficients of the linear model are chip-dependent. For the particular SCµM we used, SCµM development board Q4, we found using a linear least-squares regression:

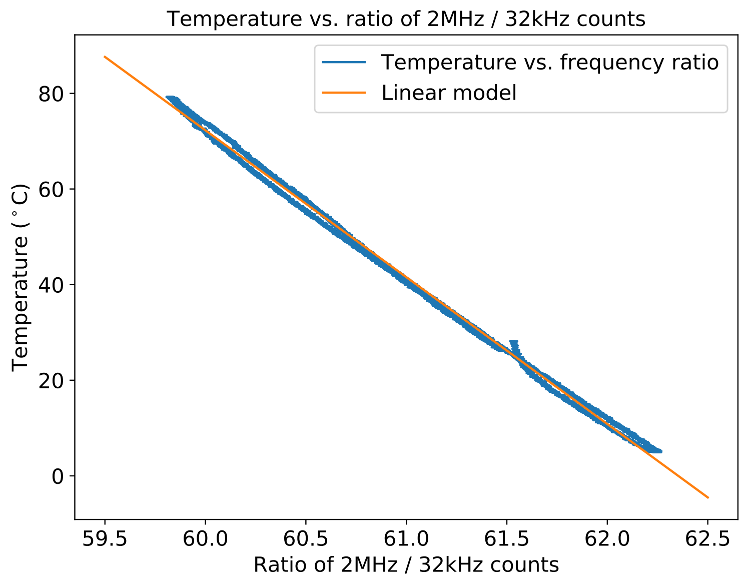
$$\text{temperature} = -30.7 \cdot \text{ratio} + 1920 \tag{3.1}$$

The maximum error between the linear model and the actual temperature was less than 3 °C, as shown in Figure 3.2b, but there are some measurement inaccuracies due to the hysteresis caused by the board's thermal mass.

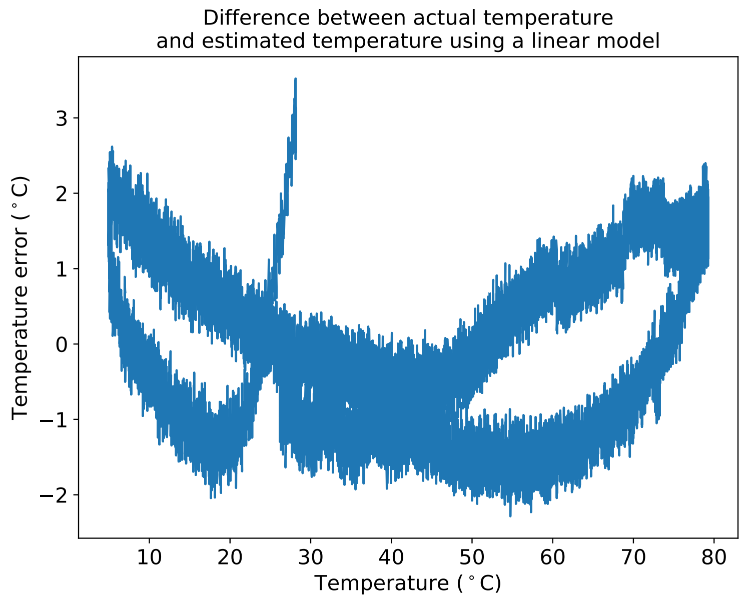
3.2 Temperature Averaging

After the initial temperature calibration described above, we observed that SCµM's estimated temperature at a given fixed temperature would vary by up to 1 °C. Compared to the Allan deviation plots in [1] for the 2 MHz and 32 kHz oscillators, the data we observed had more jitter in the 100 ms range.

[1] `https://github.com/tryuan99/scum-test-code/blob/cf82f44c9b990296e95a0af586c22b51c2e766da/scm_v3c_BLE/fixed-point.h`



(a) Temperature vs. ratio of the 2 MHz and 32 kHz counts and the linear model given by linear least-squares regression.



(b) Difference between actual temperature and estimated temperature using a linear model.

Figure 3.2: Relationship between temperature and the ratio of the 2 MHz and 32 kHz frequency counts. The temperature was varied at a rate of $\pm 1.5\,^\circ\mathrm{C/min}$, and ratio measurements were taken every 100 ms as determined by the RFTimer interrupts. The linear model given by linear least-squares regression and the corresponding differences between the actual temperature and the temperature estimated by the linear model are shown.

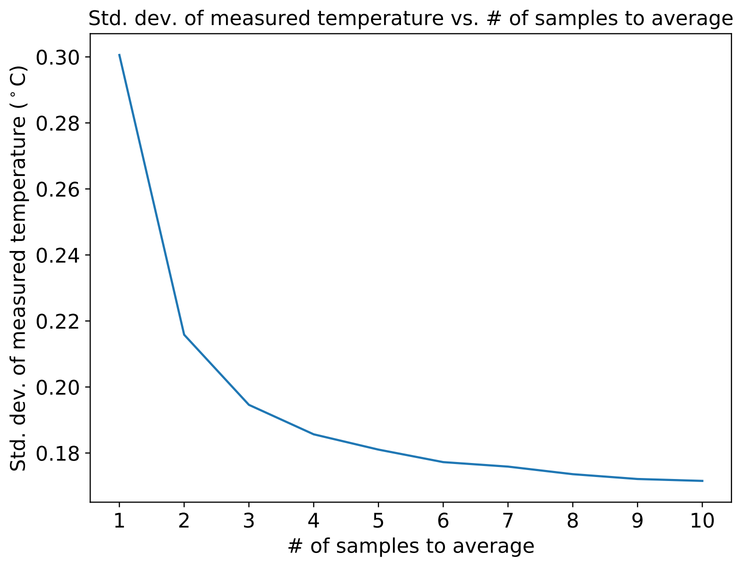


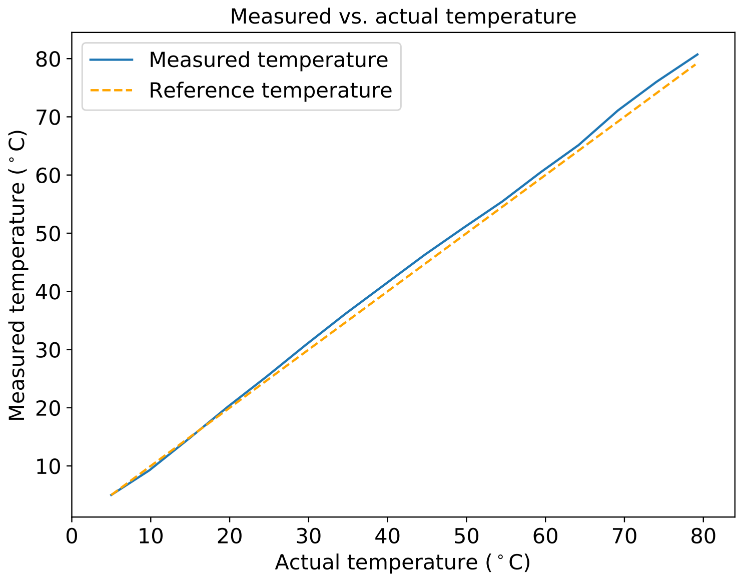
Figure 3.3: Standard deviation of the measured temperature vs. the number of samples to average over.

To mitigate variance in the temperature estimate, we implemented a simple averaging method. We recorded the 2 MHz and the 32 kHz frequency ratios at room temperature over an hour and calculated the standard deviations of the estimated temperature as a function of how many samples we average. The results are shown in Figure 3.3.

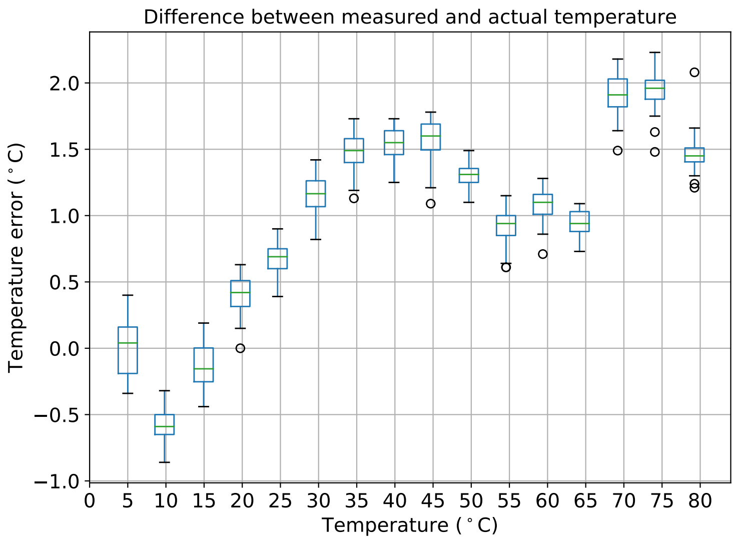
We found that averaging over any more than five temperature samples did not significantly decrease the variance of the temperature estimates, so we chose to average over five samples. This results in a duration of around 500 ms for each temperature measurement.

Having each temperature update last 500 ms with averaging instead of 100 ms corresponds to moving to a different position on the Allan variation curves in Figures 5 and 6 of [1]. From 100 ms to 500 ms, the variation of the 2 MHz oscillator stays roughly constant while the variation of the 32 kHz oscillator decreases by approximately a factor of 2 before it flattens out around 500 ms. We observe a similar trend in Figure 3.3, where the variation in the measured temperature decreases by around a factor of 2 from averaging over 1 sample to averaging over 5 samples. Afterwards, the variation in the measured temperature flattens out.

Using the coefficients given in Equation (3.1) and averaging over 5 samples, we verified the accuracy of SCµM's estimated temperature at intervals of 5 °C between 5 °C and 80 °C. At each temperature, we waited for SCµM's estimated temperature to stabilize and then recorded around 50 consecutive temperature measurements before increasing the temperature by another 5 °C. In Figure 3.4a, we used the means of these estimated temperatures at each 5 °C interval to plot SCµM's estimated temperature between 5 °C and 80 °C, and in Figure 3.4b, we show the corresponding temperature errors and their distributions. While

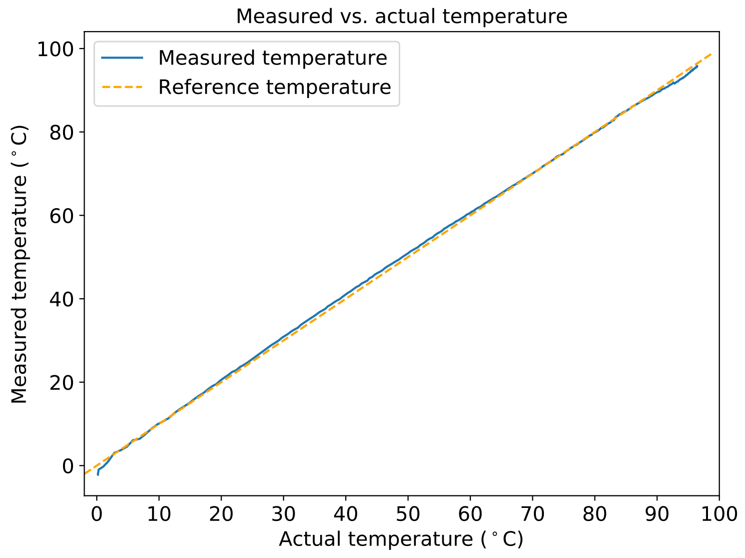


(a) SCµM's measured temperature vs. the reference temperature.

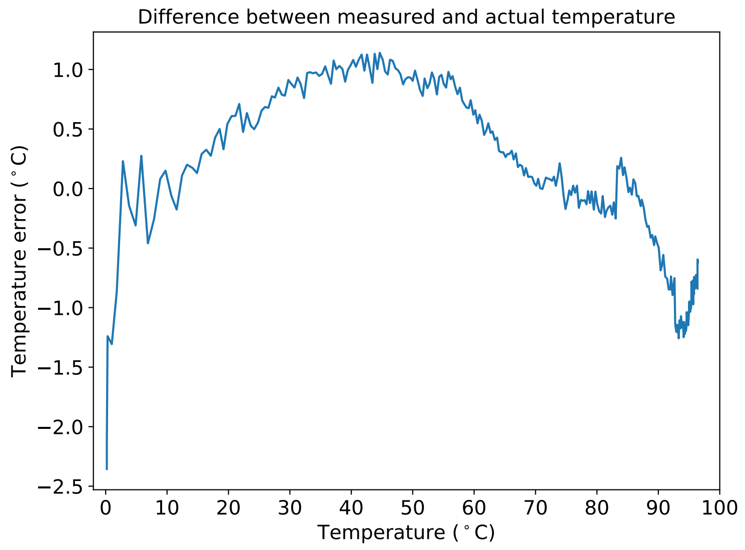


(b) Distribution of SCµM's temperature error.

Figure 3.4: SCµM's measured temperature vs. the reference temperature after a temperature sweep at a ramp rate of 1.5 °C/min between 5 °C and 80 °C.



(a) SCµM's measured temperature vs. the reference temperature.



(b) Temperature error between the measured and the actual temperature.

Figure 3.5: SCµM's measured temperature vs. the reference temperature between 0 °C and 100 °C after a two-point calibration.

the measured temperature is fairly accurate around room temperature (20 °C), the maximum error of around 2 °C occurs at higher temperatures.

3.3 Two-Point Calibration

Due to the nearly linear relationship between the temperature and the ratio of the 2 MHz and the 32 kHz frequencies, a lengthy temperature sweep to find this relationship is rather unnecessary. Instead, we propose a two-point calibration at two different temperatures. Using the same SCμM chip, we performed the temperature calibration as described above again, but we measured the ratio of the two frequencies at only 20 °C and at 30 °C, both after the temperature in the chamber stabilized. Notably, we did not choose to calibrate at the extremes of the temperature range in order to reduce the cost and time of the two-point calibration.

After calculating a linear model for the measured temperature and adjusting the bias term to account for observed hysteresis, we reprogrammed SCμM and verified the accuracy of this model by sweeping the chamber temperature from 0 °C to 100 °C while recording SCμM's estimated temperature and using the TMP102 temperature sensor as the ground truth. The measured temperature and its corresponding error between 0 °C and 100 °C are shown in Figure 3.5. At low temperatures, we noticed sporadic overflow errors occurring in the embedded software due to the implemented fixed point division algorithm. Ignoring the chip's erroneous estimated temperatures, we found that the difference between the measured and the actual temperature was within 1 °C.

3.4 Conclusion

We showed a method to find a linear relationship between the ambient temperature and the frequency ratio of the 2 MHz chipping clock for the chip's transmitter and the 32 kHz timer. Although these two RC oscillators have different temperature coefficients, their frequency ratio is roughly linear over temperature. Using a two-point calibration, the coefficients of the linear model can be easily determined, and after averaging over 5 temperature samples, we showed that the error of SCμM's measured temperature is less than 2 °C between 0 °C and 100 °C and less than 1 °C between 5 °C and 85 °C. This allows SCμM to generate fairly accurate temperature estimates and perform temperature compensation using these estimates, which eliminates the need for network-based compensation involving an external beacon. The next step is to tune the radio frequency oscillator using this temperature estimate, so that it can operate within the 802.15.4 or BLE standards over varying temperature without an external frequency reference, as described in Section 4.4. We can also transmit this temperature directly, so that SCμM can operate as a tiny wireless temperature sensor, e.g., for medical applications.

Details on the software for the aforementioned temperature calibration can be found in Appendix A.

Chapter 4

BLE TX

IEEE 802.15.4 is a communication standard commonly found in networks for internet-of-things devices. However, most commercial devices, such as phones and computers, do not support the 802.15.4 standard, making it difficult to interface with IoT devices. One possibility to allow mesh networks implementing the 802.15.4 standard to communicate with other devices is to have some IoT nodes transmit and receive data over Bluetooth Low Energy (BLE), a widely supported standard. This enables many new applications, such as controlling microrobots and communicating with wireless sensors using commercially available phones and computers.

To realize this, we would need an IoT node that is small and low-cost, so that it can be placed ubiquitously on microrobots or as wireless sensors, and that can transceive BLE-standards compliant packets. Since SCμM as presented in [8] has a standards-compatible 802.15.4 transceiver and Bluetooth Low-Energy (BLE) transmitter, we can use it as a 802.15.4-to-BLE translator for this purpose. It can communicate with other nodes in the 802.15.4 mesh network as a regular IoT node, but it can then transmit to Bluetooth-enabled devices as well.

In this chapter, we first give a brief description about the structure of the BLE packets that SCμM will transmit. Afterwards, we describe how we can tune the free-running radio frequency (RF) oscillator to transmit BLE packets on a specified BLE channel using SCμM's optical receiver.

The BLE specifications prescribe a data bitrate of 1 Mbit/s and a maximum frequency drift of ±40 ppm. However, SCμM does not have an external frequency reference. While this is great for keeping its package size small, the mote's communication frequency varies with temperature, so much that the frequency drift over temperature exceeds the limit specified in the BLE standards. Thus, we then describe how we can overcome this hurdle.

One possibility is to simply sweep the LC frequency tuning code to transmit BLE packets over a range of frequencies close to the target frequency in the hopes that at least one of the packets meets the BLE specifications. We show that this is sufficient for SCμM to operate as a BLE beacon over a range of around 20 °C. However, transmitting a BLE packet for each frequency setting is inefficient, so we present two methods to compensate the local oscillator

(LO) frequency more efficiently over temperature.

First, as described in Chaper 3, we estimate the ambient temperature by finding a linear model based on the ratio of the frequencies of two separate clocks that would be running during radio operation, a 32 kHz oscillator similar to a crystal and another 2 MHz chipping clock for the radio transmitter. We then use this temperature estimate to calculate the correct frequency setting for the LO.

Alternatively, we use an external OpenMote with no frequency drift over temperature that constantly transmits 802.15.4 packets on a specified channel as a frequency reference. As the temperature changes, we adjust SCµM's RX tuning code, such that we continue to receive the OpenMote's 802.15.4 packets, by either measuring the intermediate frequency (IF) offset or by sweeping the RX tuning code and taking the average of the tuning codes of all received packets. In the meantime, we adjust the TX tuning code for transmitting BLE packets as well. We show that these methods allow us to operate SCµM as a BLE beacon over a range of around 20 °C, limited by the monotonicity of the tuning codes.

4.1 BLE Overview

The BLE standards compatible radio on SCµM allows it to communicate with commercial off-the-shelf devices that support BLE. However, since SCµM cannot receive BLE packets (see Chapter 5), we resort to simply using SCµM as a BLE advertiser that does not accept any connections [2].

A BLE-enabled device can communicate data to other BLE-enabled devices via two possible methods:

1. It can act as a broadcaster and broadcast BLE packets to all listening BLE-enabled devices, called observers, in the vicinity. In the BLE specifications, an observer can request scan response data from the broadcaster, but since SCµM does not have a working BLE receiver, we only broadcast BLE packets from SCµM. In other words, we are limited to a one-way data transfer.

2. It can establish a permanent connection with another BLE device and exchange BLE packets. Since SCµM does not have a working BLE receiver, SCµM cannot establish these BLE connections.

Notably, BLE packets are sent on 40 different channels on the 2.4 GHz frequency band. The first 37 channels are used for connections only, and the last 3 channels—channel 37 (2.402 GHz), channel 38 (2.426 GHz), and channel 39 (2.480 GHz)–are used for advertising. Unless stated otherwise, we will solely focus on having SCµM broadcast on channel 37 (2.402 GHz).

The Generic Access Profile (GAP) layer specifies how BLE advertising is accomplished. Since SCµM can only be used as a broadcaster, GAP specifies the broadcaster and observer roles, whether the device is discoverable or connectable, and how the advertising data is communicated to the observer.

	Component	Length	Value (in little-endian format unless stated otherwise)
BLE packet	Preamble	1 byte	`0x55`
	Access address	4 bytes	`0x6B7D9171`
	PDU header	2 bytes	`0x40A4`
	Advertiser address	6 bytes	`0x0002723280C6` in big-endian format
	Payload	0-31 bytes	Application-dependent. SCµM uses all 31 bytes and zero-pads the unused bits.
	CRC	3 bytes	Calculated during packet assembly.

Table 4.1: BLE advertising packet structure. The bold components constitute the Protocol Data Unit (PDU).

BLE Advertising Packet Structure

The BLE advertising packet is structured as shown in Table 4.1[1]. The bold components, i.e., the PDU header, the advertiser address, and the payload, constitute the Protocol Data Unit (PDU). When assembling the BLE packet, the CRC is first calculated based on the PDU contents, and both the PDU and the CRC are then whitened using a linear feedback shift register (LFSR). The BLE packet generation code for SCµM can be found in the function `ble_gen_packet`[2].

We now break down each component of a BLE advertising packet[3]. Note that BLE packets are transmitted little-endian, so the order of the bits in each byte of the packet must be flipped prior to transmission.

1. The preamble of a BLE advertising packet is `0b10101010`, or `0x55` in little-endian format.

2. The access address of a BLE advertising packet is always `0x8E89BED6`, or `0x6B7D9171` in little-endian format.

3. The PDU header consists of a 4-bit PDU type specifying the type of advertising followed by 2 bits for an `RFU` field (reserved for future use), 1 bit each for `TxAdd` and `RxAdd`, and 8 bits for the length of the ensuing PDU. In little-endian format, we set the value of

[1]See `https://web.archive.org/web/20200523001311/https://microchipdeveloper.com/wireless:ble-link-layer-packet-types` for more details on the BLE packet structure. Archived from `https://microchipdeveloper.com/wireless:ble-link-layer-packet-types` on May 22, 2020.

[2]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/ble.c#L70`

[3]`https://web.archive.org/web/20200523001538/http://j2abro.blogspot.com/2014/06/understanding-bluetooth-advertising.html`. Archived from `http://j2abro.blogspot.com/2014/06/understanding-bluetooth-advertising.html` on May 22, 2020.

Length	GAP code	Data
1 byte	1 byte	n bytes

Table 4.2: The structure of each data chunk in the BLE advertising packet payload. The length byte is equal to $n + 1$, i.e., it is the length of the GAP code and the data.

the PDU header to `0x40A4`. The first four bits (`0b0010`, or `0x4` in little-endian format) signifies an `ADV_NONCONN_IND` event that is non-connectable and undirected. `TxAdd` is set to `0` because we have a fixed public advertiser address. The second byte (`0d37`, or `0xA4` in little-endian format) specifies the total length of the advertiser address and the payload in the BLE packet. We set it equal to the maximum possible value of 37 bytes because we will simply zero-pad the unused bits in the payload.

4. The advertiser address is set to `0x0002723280C6` in big-endian format for SCµM. In general, the advertiser address can be arbitrary, but it is usually unique to the advertiser, so most smartphone apps, including the BLE sniffer app described below, will only show the first BLE packet received from each unique advertiser address during each scan for BLE packets.

5. The payload can have a length up to 31 bytes. Its structure is described below.

6. The CRC is calculated based on the PDU header, the advertiser address, and the payload prior to whitening of the PDU and CRC fields with an LFSR. The LFSR is initialized to `0b1, channel`, where `channel` is the TX channel (channel 37 for SCµM).

According to the GAP guidelines, the payload consists of multiple chunks of data, each consisting of the length of the GAP code and data followed by the GAP code and the data itself as shown in Table 4.2.

Since the GAP code is always one byte in length, the length field is always equal to the data length+1. Most of the GAP codes are pre-defined[4]. In particular, we use the GAP code `0x08` to advertise the short name of SCµM, which is set to `SCUM3`. However, it is possible to define custom GAP codes for other types of data specific to the application, which are listed in Table 4.3. Note that all of the bytes, including the GAP code, the length, and the actual data, have to be in little-endian order.

BLE Sniffer App

In the BLE experiments, we used SCµM as the BLE broadcaster, and we used a Google Pixel 2 XL as the observer. Successful reception of SCµM's BLE packets indicated that SCµM's

[4]`https://web.archive.org/web/20200523001739/https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile/`. Archived from `https://www.bluetooth.com/specifications/assigned-numbers/generic-access-profile/` on May 22, 2020.

Data	Data length (excluding GAP code length)	GAP code	Notes
Short name	5 bytes	`0x08`	Assigned GAP code
LC frequency tuning code	2 bytes	`0xC0`	15-bit LC frequency tuning code
2 MHz and 32 kHz frequency counters	8 bytes	`0xC2`	4 bytes for 2 MHz counter and 4 bytes for 32 kHz counter
Temperature	2 bytes	`0xC1`	$100 \cdot$ `temp` truncated to the nearest integer
Custom data	4 bytes	`0xC3`	Used to re-broadcast 802.15.4 packets received from OpenMotes as BLE packets

Table 4.3: List of GAP codes defined by the software described here and their data lengths.

BLE compatible radio was functional and tuned correctly.

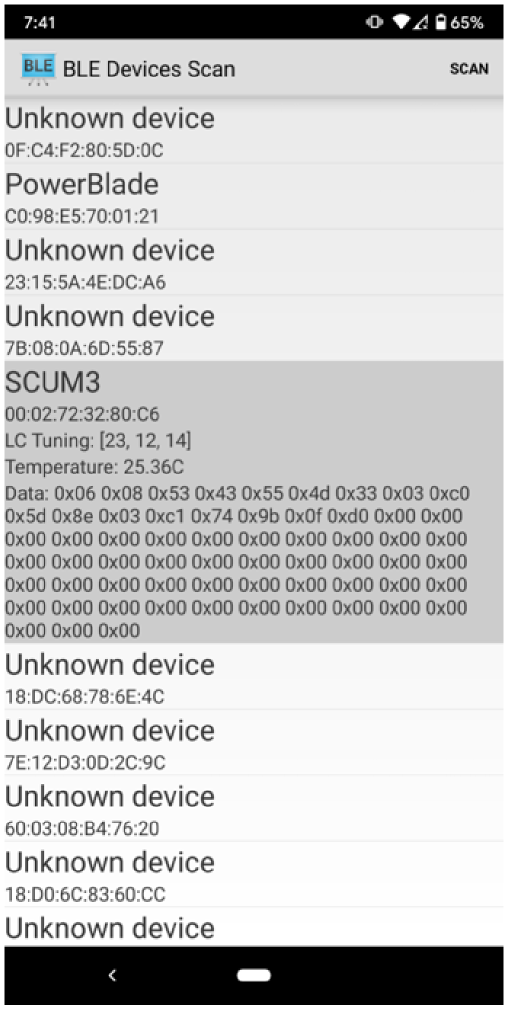
In order to correctly parse the received BLE packets, including the custom GAP codes defined in Table 4.3, we developed an Android app based on a sample app provided by Google to read GATT attributes from BLE devices. This Android app, which can found at `https://github.com/tryuan99/android-ble-scum`, scans the environment for BLE advertising packets, and if it finds packets with an advertiser address of `0x0002723208C6`, it parses the payload and displays the relevant information on the screen. An example of the app is shown in Figure 4.1.

4.2 BLE TX On SCµM

SCµM is usually configured to transmit 802.15.4 packets. In order for SCµM to transmit BLE packets, we initialize the analog scan chain (ASC) with different configuration bits, as done in the function `ble_init_tx`[5]. We no longer need the 802.15.4 modulation DAC and instead enable the BLE modulation DAC. Most importantly, `analog_cfg[183]` must be set to `0` to select BLE modulation (a `1` will select 802.15.4 modulation otherwise), which can be accomplished by setting `ANALOG_CFG_REG__11 = 0x0000` instead of `0x0080`.

After initializing the radio to use BLE modulation, when transmitting a BLE packet on SCµM, we first load the BLE packet into an asynchronous FIFO whose contents are clocked out at 1 MHz divided down from the 2 MHz chipping clock. In software, this is done in

[5]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/ble.c#L265`



Unknown device
0F:C4:F2:80:5D:0C
PowerBlade
C0:98:E5:70:01:21
Unknown device
23:15:5A:4E:DC:A6
Unknown device
7B:08:0A:6D:55:87
SCUM3
00:02:72:32:80:C6
LC Tuning: [23, 12, 14]
Temperature: 25.36C

```
Data: 0x06 0x08 0x53 0x43 0x55 0x4d 0x33 0x03 0xc0
0x5d 0x8e 0x03 0xc1 0x74 0x9b 0x0f 0xd0 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00
```

Unknown device
18:DC:68:78:6E:4C
Unknown device
7E:12:D3:0D:2C:9C
Unknown device
60:03:08:B4:76:20
Unknown device
18:D0:6C:83:60:CC
Unknown device

Figure 4.1: BLE sniffer app running on a Google Pixel 2 XL after receiving a BLE packet from SCµM. The payload includes the short name of SCµM, the LC frequency tuning codes, and the temperature. The raw payload is displayed as well.

the function `ble_transmit`[6]. We first call `load_tx_arb_fifo`, then turn on the LDOs, and finally transmit the BLE packet using `transmit_tx_arb_fifo`. The bits are then used to modulate the LO. Note that SCµM's BLE packets are modulated with FSK instead of GFSK as specified by the BLE standards. However, off-the-shelf devices, such as phones, are still able to receive FSK BLE packets from SCµM.

There was one issue that I discovered while debugging why SCµM was not modulating when I initialized the radio to transmit BLE packets. The expected current draw on SCµM board Q4 with the LO, PA, and DIV on is around 2.2 mA, but the current I was measuring was considerably lower. After further debugging, I found that adding a delay in the orders of tens of microseconds after turning on the LDOs solved this issue[7]. The issue is most likely caused by transients after the LDOs are turned on, and immediately transmitting the BLE packet from the FIFO after turning the LDOs on will not work as expected.

To tune the LO frequency, we adjust the 15-bit LC frequency tuning code that controls the 3 5-bit capacitive DACs of the 2.4 GHz LC oscillator. However, the LO frequency is usually application-specific, e.g., the desired BLE channel to transmit on may vary, and the LC frequency tuning code depends on the ambient temperature, so we cannot easily hard-code the frequency tuning code. The tuning code is also chip-dependent primarily due to effective reference mismatch between voltage regulators and the DC supply sensitivity of the oscillator. Furthermore, since there are 2^{15} possible LC frequency tuning codes, sweeping all settings to find the correct tuning code takes unfeasibly long. We thus use the LC divider, shown in Figure 2.12, to estimate the LC frequency and use the divider to calibrate the LO frequency during bootloading of SCµM.

Thus, since the LC frequency is calibrated with the divider on, we turn it on as well when transmitting BLE packets. The reason for this is simple: the divider consumes quite some power (see Figure 5.21 of [9]), and turning it off will shift the LC frequency and invalidate the tuning code found during LC frequency calibration. To conserve power, it is possible to turn off the LC divider as it is not necessary for BLE TX, but the LC frequency tuning code will need to be found using an alternative method, e.g., by sweeping all tuning codes.

Lastly, since the modulated BLE signal has a frequency spacing of 500 kHz, in order to transmit at 2.402 GHz (channel 37), we set the target frequency of the LO to be 250 kHz below the channel frequency, i.e., at 2.401 75 GHz, so that the modulated BLE signal is centered at 2.402 GHz.

4.3 Optical LC Frequency Calibration

The simplest method to calibrate the LC frequency of SCµM without the use of the LC divider is to use a spectrum analyzer to measure the frequency of SCµM's radio output.

[6]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/ble.c#L373`

[7]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/ble.c#L381`

We can adjust the LC frequency by changing the 15-bit LC frequency tuning code until the desired LC frequency is achieved. However, this method is infeasible in the absence of a spectrum analyzer, so we describe another approach using the LC divider to calibrate the LC frequency during bootloading.

After the initial 25 SFD interrupts to calibrate the other free-running oscillators, as described in Section 2.2, we begin LC calibration, which proceeds similarly. After every SFD interrupt, we measure the number of LC counts within the last 100 ms and find the 15-bit frequency tuning code that gets us closest to the target frequency of 2.401 75 GHz. However, since 2^{15} 100 ms SFD interrupts would take unfeasibly long, we observed that for many SCµM chips we tested, 2.401 75 GHz corresponded to a coarse code between 21 and 25. Furthermore, we fix the fine code to be 15, so that we would only be sweeping the coarse code between 21 and 25 and the mid code between 0 and 31, totaling around 20 s for LC calibration. If the ambient temperature does not vary much, this calibration to find the optimal coarse and mid codes for BLE transmission is a one-time operation.

However, since we fixed the fine code at 15, the LO frequency might be incorrect. Furthermore, as described in Section 2.4, the frequency error over temperature of the LO is greater than that specified in the BLE standard. Thus, even after the initial LO frequency tuning, we are limited to BLE transmission within a small range of temperature. An initial working solution to this is to simply sweep the fine code from 0 to 31 while keeping the coarse and mid codes as calibrated. We do not want the fine code to roll over because as described in Section 2.4, the LC frequency is not monotonic when the fine code rolls over, making LC frequency compensation much harder.

To characterize how well this fine code sweep works over temperature variations, we placed the SCµM development board Q4 in a temperature chamber. Inside, we connected an SMA cable to SCµM, which we ran out of the temperature chamber, and attached an antenna onto the other end of the SMA cable. This antenna was then placed next to a phone to pick up any BLE packets transmitted from SCµM.

Prior to the temperature sweep, we ran LC frequency calibration on SCµM in order to determine the optimal coarse and mid codes to use at room temperature with the additional SMA cable at the antenna output. We also programmed SCµM, such that the last byte of its advertiser address would be equal to the fine code, on which it was transmitting. This way, on the BLE sniffer app, when we received a BLE packet from SCµM, we could determine the fine code on which it was transmitted.

Finally, we selected four temperatures between 5 °C and 35 °C to measure the range of fine codes the phone could receive, as shown by the blue dots in Figure 4.2. At each of these four temperatures, we waited around two minutes for the on-board temperature on SCµM to settle in order to mitigate any hysteresis caused by the thermal mass of the development board. Note that we did not allow the fine code to roll over, so we only swept the fine code between 0 and 31. As shown in Figure 4.2, a one-time LC frequency calibration at room temperature allows for BLE operation over a range of around 20 °C.

Note that since we attached a long SMA cable with the antenna to SCµM, this changes the load at the antenna output and affects the BLE TX frequency. We can approximate

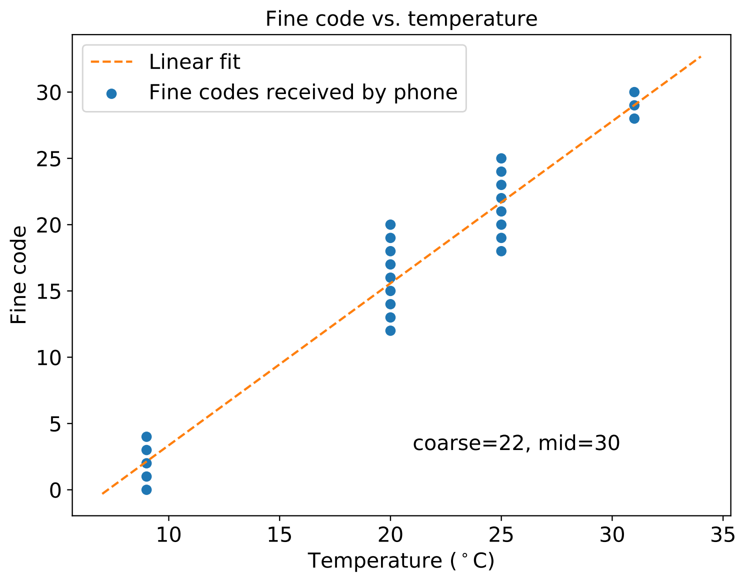


Figure 4.2: The fine codes of BLE packets received by a smartphone at four select temperatures and the corresponding linear fit. These values were recorded in a temperature chamber by running a long SMA cable with an antenna out of the chamber. The presence of the SMA cable causes a frequency shift, so we add a bias term of around −10 fine codes to the linear fit when we remove the SMA cable.

its effect as a constant bias term on the fine code, so after removing the SMA cable after the temperature sweep, we simply determine the range of fine codes that can be received at room temperature and calculate this bias term. Experimentally, on SCμM development board Q4, we found this bias term to be around −10 fine codes.

Since each fine code corresponds to around a 100 kHz change in the LO frequency but we could pick up packets over a range of nine fine codes at 20 °C, for example, this means that the phone could receive packets over a range of 900 kHz, considerably larger than the ±40 ppm frequency drift specified by the BLE specifications. This implies that the phone has some frequency tolerance.

This temperature sweep also lets us hypothesize that in order for SCμM to operate in a larger temperature range, we could perform LC calibration at different temperatures and sweep the fine codes for every coarse and mid code pair found for each temperature. For example, for radio operation between 0 °C and 45 °C, we could calibrate SCμM at 5 °C, 20 °C, and 30 °C, find their corresponding coarse and mid codes, and sweep the fine code from 0 to 31 with every of the three coarse and mid code pair to ensure that at least one of these settings meets the BLE frequency specifications.

While sweeping the fine code is a simple brute force solution, it is not efficient in terms of both power and time as SCμM transmits multiple packets for each data. In the next two sections, we present two methods for improved frequency compensation over temperature.

4.4 LC Frequency Compensation Using A Temperature Estimate

In Figure 4.2, we observe that instead of sweeping through all fine codes, we can find a linear model that determines which fine code to use at a given temperature. We thus only need to transmit one packet with this frequency setting instead of 32 identical packets over a range of frequencies, conserving power by a factor of 32. Furthermore, as described previously, since the phone has some frequency tolerance, as long as the predicted fine code is close enough to the channel frequency, then the packet will be received.

As described in Chapter 3, we can obtain a fairly accurate temperature estimate on SCµM by measuring the ratio of the 2 MHz and the 32 kHz RC oscillator frequencies. Therefore, we need to perform two temperature sweeps to allow LC frequency compensation using a temperature estimate:

1. We perform one temperature sweep in order to find the relationship between the SCµM's 2 MHz and 32 kHz oscillator frequency ratio and the ambient temperature. This can be a simple two-point calibration.

2. We perform another temperature sweep in order to determine which fine codes are received by the phone at various temperatures. Using the mean fine code received at each temperature, we then perform a least-squares linear regression to find the best linear model relating the fine code to the measured temperature. After the temperature sweep, if the antenna load was changed during the sweep, e.g., with the presence of an SMA cable, the linear model needs to be corrected with a constant bias term.

For the SCµM development board Q4, we found the relationship between the fine code and the temperature to be:

$$\text{fine code} = 1.2 \cdot \text{temperature} - 18.4 \tag{4.1}$$

We then verified that this linear model is feasible by placing SCµM into the temperature chamber, sweeping the ambient temperature, and checking that the phone still receives BLE packets across temperature. The primary disadvantage of this frequency compensation method is that it requires two temperature sweeps: one to find the linear model for the temperature estimate and one to find the linear model for the fine code depending on the temperature.

4.5 LC Frequency Compensation By Tracking The 802.15.4 RX Frequency

Since SCµM was designed to operate in 802.15.4 networks with other IoT devices, we can use the 802.15.4 packets in the environment as a frequency reference for SCµM's LO frequency, thus enabling real-time frequency compensation. Our setup consisted of an OpenMote CC2538 placed about 15 cm away from SCµM that transmitted an 802.15.4 packet with the same packet contents approximately every 62.5 ms on channel 11 (2.405 GHz).

SCµM would act as a "translator," receiving 802.15.4 packets and re-transmitting the contents of the most recently received 802.15.4 packet as a BLE packet every 400 ms. We kept track of the 802.15.4 RX frequency tuning code, so that we could continue receiving the 802.15.4 packets from the OpenMote, and the BLE TX frequency tuning code, so that we could transmit BLE packets on channel 37 (2.402 GHz).

Every 800 ms, SCµM would adjust its RX and TX tuning codes. Before starting the RX and TX frequency compensation, we pre-calibrated SCµM, so that we knew the RX and TX frequency tuning codes at room temperature. For the SCµM board Q4 used in this experiment, the 802.15.4 RX coarse and mid codes were $(23, 11)$, and the TX coarse and mid codes were $(23, 15)$. In the future, we can determine the initial RX and TX frequency tuning codes using the calibration box described in [3], where the authors use many OpenMotes to find the correct RX and TX tuning codes for each of the sixteen 802.15.4 channels. This can be augmented to find the BLE TX tuning code.

Due to the non-monotonicity of the LC frequency over the 15-bit frequency tuning code, as shown in Figure 2.13, we focused on frequency compensation over one coarse and one mid code to ensure that the frequency is strictly monotonically increasing with the fine codes. From the aforementioned fine code sweep in Section 4.4, we know that 32 fine codes allow for BLE operation over a range of around 20 °C.

In [12], the authors characterized the difference between the RX and TX tuning codes for 802.15.4 channels at 10 °C and 50 °C. For 802.15.4 channel 11, the difference was 50 tuning codes at 10 °C and 28 tuning codes at 50 °C. However, since we focused on frequency compensation over 20 °C and the phone has some frequency tolerance, we assumed that a change in the RX tuning code corresponded to an equal change in the TX tuning code. This one-to-one correspondence turned out to be an acceptable approximation as will be shown in the next two subsections.

Both of these RX feedback methods could be used for online network-based frequency compensation after an initial calibration. For example, a crystal-free mote operating in a mesh network with other 802.15.4 devices could first find the initial frequency tuning codes for 802.15.4 TX and RX using the QuickCal algorithm presented in [3]. Afterwards, during normal operation, it can use the 802.15.4 packets transmitted by other devices to calibrate its own 802.15.4 TX and RX tuning codes.

RX IF Frequency Compensation

As described in [13, 16], we can perform network compensation by measuring the IF frequency as SCµM receives 802.15.4 packets. In RX mode, we set the LO frequency to be 2.5 MHz below the target channel frequency, i.e., 2.4025 GHz for 802.15.4 channel 11. When we receive a packet, the received RF signal is down-converted to an intermediate frequency (IF) of 2.5 MHz before demodulation.

We measure the IF frequency by counting the number of zero-crossings within 100 µs. If the LO frequency is correctly tuned, the IF frequency should be 2.5 MHz. In a 100 µs interval, we thus expect 500 zero-crossings; otherwise, there is some non-zero IF offset from the expected value of 500, which indicates that the LO frequency is not tuned to exactly 2.4025 GHz. The IF frequency count is performed in hardware and is written to a register, which we can then read in software[8]. Additionally, we can read the received signal strength indicator (RSSI) and the link quality indicator (LQI) error rate generated by the digital baseband from two hardware registers whenever we receive a packet.

As described in [16], to reduce noise from spuriously received packets, we only use the IF offsets of packets that have a sufficiently low LQI error rate as packets with a large LQI error rate likely encountered interference. Furthermore, we convolve the IF offsets of all received packets with a pre-defined FIR Gaussian filter in the firmware and use the filtered IF offset to tune the LO. The FIR Gaussian filter[9] has 10 taps (length chosen arbitrarily) and was designed to have a corner frequency of 0.5 Hz. For this experiment, we used this filter to smooth out the IF offsets of the received packets before adjusting the LO frequency, but any filter that does some averaging on the history of IF offsets would probably work as well.

Every 800 ms, we then check if we have received more correct 802.15.4 packets than the number of taps in the FIR Gaussian filter. If so, we convolve the IF offsets of the most recently received packets with the FIR Gaussian filter to find the filtered IF offset. Each fine code corresponds to around 100 kHz for the LO frequency, which then corresponds to an IF offset of 20, so if the filtered IF offset is greater than ± 20, then we know that the RX and TX tuning codes are tuned incorrectly by at least one fine code. Since the frequency difference between two successive tuning codes corresponds to an IF offset of 20, we decided to adjust the LO frequency if the filtered IF offset has an absolute value greater than 12, which corresponds to a frequency offset of 60 kHz, a little more than halfway to the next frequency tuning code.

We placed SCµM with an OpenMote CC2538 inside the temperature chamber and increased the chamber temperature from 16 °C to 35 °C at a rate of 1.5 °C/min. To reduce any temperature lag on SCµM caused by the thermal mass of the development board to which the SCµM chip was wirebonded, we used a Nubee NUB8500H infrared thermometer

[8]https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/scm3c_hw_interface.c#L1039

[9]https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/radio.c#L23

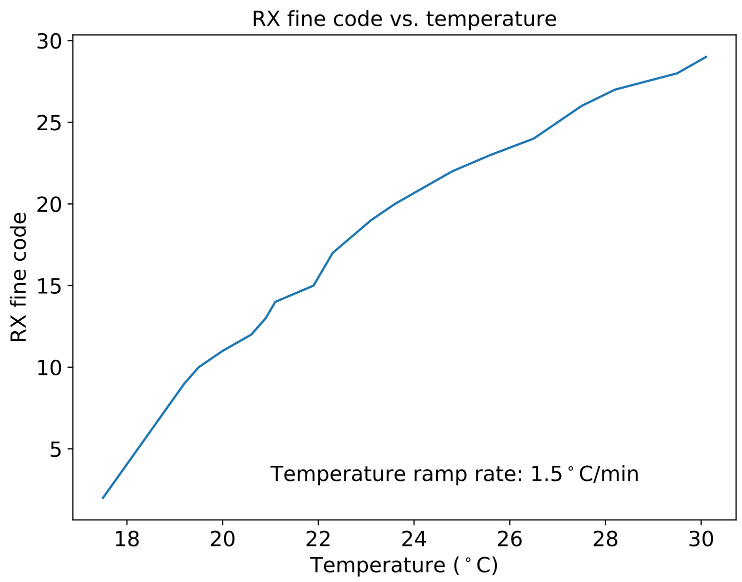


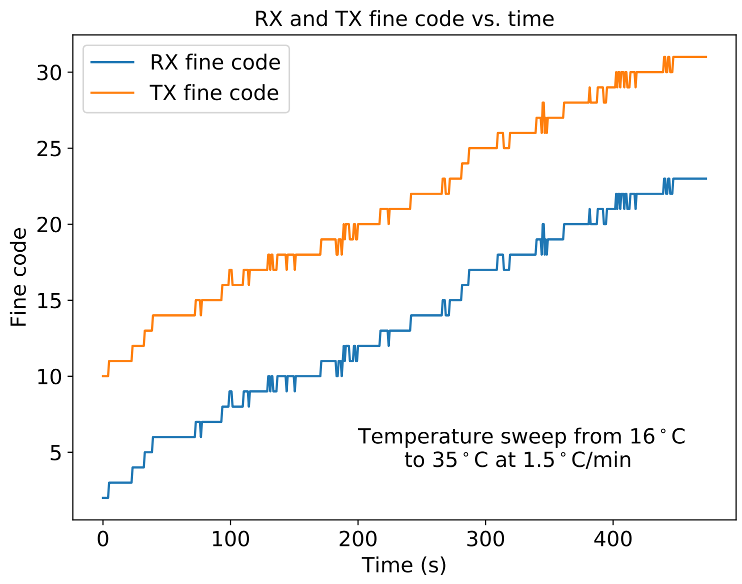
Figure 4.3: The 802.15.4 RX fine code over temperature after IF compensation as the temperature was increased from 16 °C to 35 °C at a ramp rate of 1.5 °C/min.

to measure the on-board temperature. Using this thermometer, we measured the on-board temperature whenever the RX fine code was incremented, which is shown in Figure 4.3.

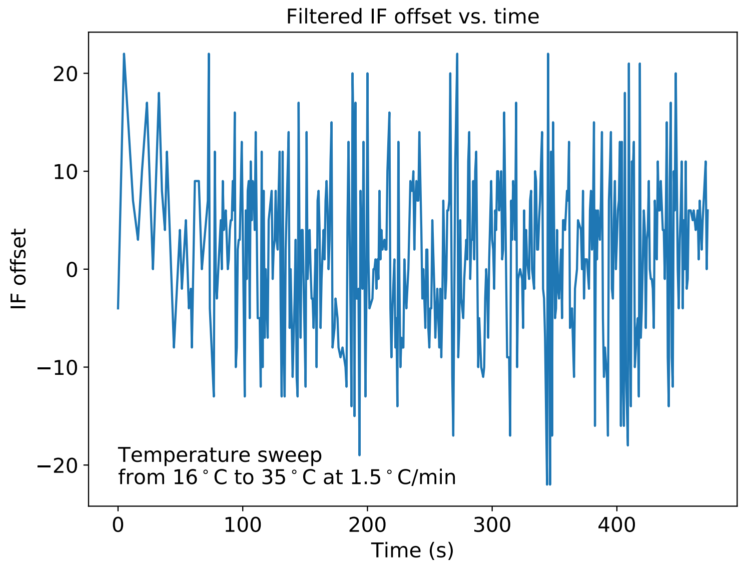
In Figure 4.4, we plot the RX and TX frequency tuning codes over time as the temperature increases from 16 °C to 35 °C with the corresponding filtered IF offset. The filtered IF offset is always within ±20, indicating that the RX frequency is within ±40 ppm of 802.15.4 channel 11. During this entire experiment, we confirmed with a phone that we were receiving BLE packets from SCµM in the temperature chamber.

We also characterize the effect of the Gaussian FIR filter with which we convolve the raw IF offsets before adjusting the 802.15.4 RX and BLE TX fine codes. In Figure 4.5, we plot the raw IF offset of all 802.15.4 packets received during this experiment. Note that compared to Figure 4.4b, we observe that the unfiltered IF offset of all received packets is within ±40, around twice the range of the filtered IF offsets. Therefore, the Gaussian FIR filter is necessary to remove some of the noise present in the digital baseband's IF estimates.

While frequency compensation using the IF offset allows for near real-time tuning code updates, we noticed that this method took some time to find the correct tuning codes if the initial LO frequency was not near the correct value. The cause of this is that we only use the IF offsets from packets with a low LQI error rate and we need to receive at least as many error-free packets as taps in the FIR Gaussian filter before updating the tuning codes. If the RX LO frequency is tuned far from the desired frequency of 2.4025 GHz, most of the received packets have a high LQI error rate, so it takes long to receive enough correct 802.15.4 packets for IF compensation.



(a) The 802.15.4 RX and BLE TX fine codes over time after IF compensation. The TX fine code was adjusted by the same amount as the RX fine code to keep the filtered IF estimate at its nominal value of 500, or the filtered IF offset at its nominal value of 0.



(b) The corresponding filtered IF offset over time from the nominal value of 500. An IF offset of 20 corresponds to around 40 ppm of deviation from the channel frequency.

Figure 4.4: SCμM's RX and TX fine codes and the corresponding filtered IF offset during a temperature sweep from 16 °C to 35 °C at a ramp rate of 1.5 °C/min. Both the fine codes and the filtered IF offset were recorded every 800 ms.

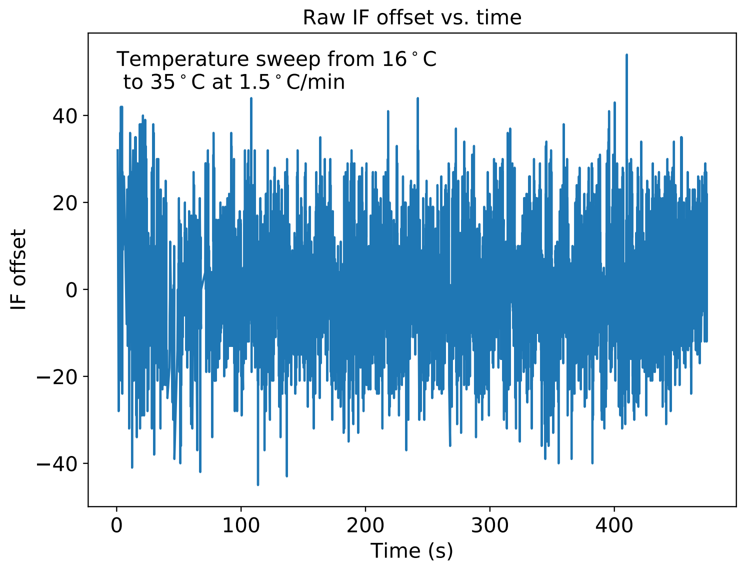


Figure 4.5: The raw IF offset of all 802.15.4 packets with a sufficiently low LQI error rate that were received from the OpenMote. The raw IF offsets were recorded during a temperature sweep from 16 °C to 35 °C at a ramp rate of 1.5 °C/min with IF compensation based on the filtered IF offset. The Gaussian FIR filter has 10 taps, so convolving every 10 of the raw IF offsets gives the filtered IF offset shown in Figure 4.4b.

Averaging Over RX Fine Codes

An alternative to using the IF offset is to simply find the fine code on which SCμM receives the most 802.15.4 packets. In order to realize this, every few seconds, we would have SCμM stop its 802.15.4-to-BLE translation task, which involves listening for packets and intermittently broadcasting BLE packets, and just listen for 802.15.4 packets.

When SCμM listens for 802.15.4 packets, we sweep its RX frequency tuning code within a range of ±2 fine codes of the current RX tuning code without rolling over. This corresponds to listening for 802.15.4 packets on five different frequency settings in a range of approximately 500 kHz. At each of the five fine codes, we listen for incoming 802.15.4 packets for 800 ms, recording how many packets are received at each frequency setting. Therefore, we spend a total of 4 s listening for 802.15.4 packets.

Afterwards, we find the weighted average of the fine codes of all received packets within the last 4 s, which indicates the best fine code to listen on for incoming 802.15.4 packets. We then adjust the RX tuning code accordingly and change the TX tuning code by the same amount.

In a temperature sweep from 16 °C to 35 °C, we verified that this method also allows us to compensate for any frequency drift in the LO and that we are continuously receiving BLE packets during the entirety of the sweep. Although listening for incoming 802.15.4 packets pauses 802.15.4-to-BLE translation for a total of around 4 s, this method does work better if the initial LO frequency is further off from the correct setting.

One issue that I ran into while running this experiment is that I would randomly not receive packets even when I expected to receive packets, skewing the weighted average and causing the LO frequency to be tuned incorrectly. For example, I would receive some 802.15.4 packets on fine code x, but after incrementing the RX frequency tuning code to $x+1$, I would not receive any packets at all. However, if I increment the RX frequency tuning code again to $x+2$, I would receive some 802.15.4 packets again. The next time SCμM sweeps the RX tuning code again to listen for packets, I would receive 802.15.4 packets on all three frequency settings. On other occasions, I would not receive any 802.15.4 packets for the entirety of the 4 s even though I would receive some packets during the next round of calibration.

I did not manage to debug the root cause of this radio issue. Instead, a quick fix I implemented is to threshold the number of packets that need to be received before the RX fine code is set equal to the weighted average of the received fine codes. I set an arbitrary threshold of 10, so that I would need to receive at least ten 802.15.4 over the 4 s of listening before adjusting the RX and TX frequency tuning codes.

4.6 Conclusion

We showed how we could create a Bluetooth Low Energy (BLE) beacon and a 802.15.4-to-BLE translator using a crystal-free mote with a standards-compatible radio. Without an external frequency reference, the LO is subject to a temperature coefficient of around $-40\,\text{ppm}/^\circ\text{C}$ [9], exceeding the frequency specification in the BLE standards. After tuning the LC frequency via 100 ms SFD interrupts to find the frequency settings closest to the desired target frequency (2.401 75 GHz for BLE channel 37), the naive solution is to sweep the fine code of the frequency tuning code while transmitting BLE-compliant packets to compensate for temperature changes.

To be more efficient, we notice that the phone has some frequency tolerance, so we can use a temperature sweep to find a linear model that determines the fine code to use at each temperature. Otherwise, we can use network compensation by either measuring the IF offset of incoming 802.15.4 packets to tune both the 802.15.4 RX and BLE TX tuning codes or by averaging over the fine codes of incoming 802.15.4 packets to correct any LO frequency drift. While measuring the IF offset allows for real-time frequency compensation, it performs worse with a large frequency drift. Future work could involve combining these two network compensation methods to leverage both of their benefits and create a tiny, temperature-independent 802.15.4 and BLE-enabled IoT mote without any external frequency reference.

Details on the software for BLE TX as well as on performing the fine code calibration and using 802.15.4 RX to compensate the BLE TX frequency can be found in Appendix B.

Chapter 5

BLE RX

In Chapter 4, we discussed how to transmit BLE packets from SCµM over temperature variations without an external frequency reference. However, to be a fully fledged BLE-compatible mote, SCµM would need to receive BLE packets from off-the-shelf devices as well.

In particular, two-way communication with SCµM over BLE enables many interesting applications because almost all smartphones support BLE but not 802.15.4. For example, for microrobotic applications, if SCµM is placed on a robot, we could control it using our smartphones by sending control signals to it over BLE.

However, since SCµM's radio was originally designed to support 802.15.4 packets, we cannot use the RFcontroller's RX finite state machine to receive BLE packets. Furthermore, BLE technically requires Gaussian frequency-shift keying (GFSK) as the channel frequencies are more closely spaced together. However, as shown in Figure 5.1, the packet error rate is considerably higher for packets using GFSK modulation at the same TX power.

In this chapter, we describe the progress made in receiving BLE packets from other SCµMs and from off-the-shelf devices. We explain how we can detect a BLE advertising packet by comparing the 32-bit access address to a 32-bit shift register containing the recovered data. Afterwards, we show that transmitting FSK BLE packets from SCµM to SCµM works, but SCµM is unable to receive GFSK BLE packets from off-the-shelf BLE devices.

5.1 BLE RX on SCµM

To configure SCµM to receive BLE packets, we use the zero crossing counter, which can be initialized by calling `radio_init_rx_ZCC_BLE`[1]. This function initializes the ZCC module to recover the clock and data, which will then be used to check whether we have received a valid BLE advertising packet. For BLE, we want the recovered clock frequency to be 1 MHz. If the recovered clock frequency seems to be off by more than 1% of 1 MHz, we can

[1]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/scm_ble_functions.c#L382`

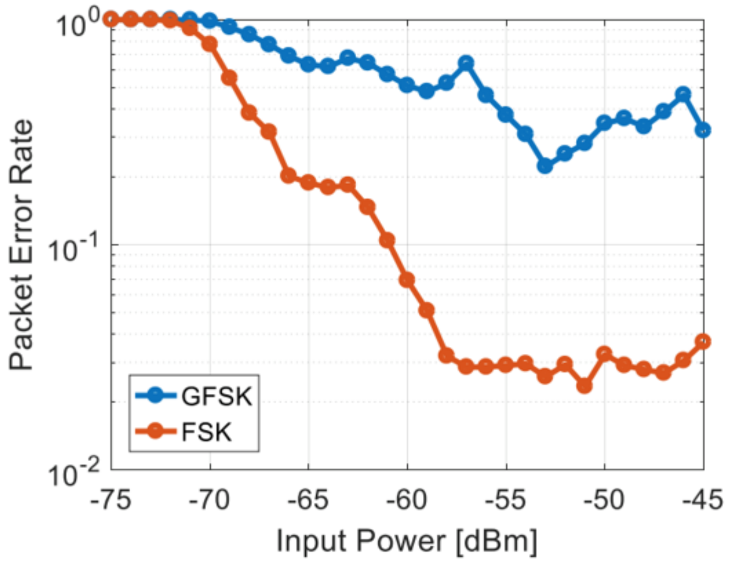


Figure 5.1: Packet error rate vs. input power between FSK and GFSK modulation. This plot was generated by Brad Wheeler.

adjust the IF clock by modifying `IF_clk_target`[2] before optical frequency calibration. To receive BLE packets, we then first turn on the corresponding LDOs by calling the function `radio_rxEnable` and `radio_rxNow` to reset the digital baseband[3].

The recovered data and clock from the ZCC module are available as GPIO outputs[4] in bank 4 as `mux_out_M0_data` and `mux_out_M0_clk`. Probing the recovered clock is the easiest way to verify that the recovered baseband clock frequency for BLE packets is 1 MHz. The recovered data is then fed through a 32-bit shift register clocked by the recovered baseband clock.

There is another memory-mapped 32-bit register that contains the 32-bit target value we are searching for within the 32-bit shift register. The 32-bit target value is set through `ANALOG_CFG_REG__1` for the 16 LSBs and `ANALOG_CFG_REG__2` for the 16 MSBs[5]. As described in Section 4.1, the PDU of the BLE packet is whitened prior to transmission. Therefore, we choose the access address of a BLE advertising packet, `0x6B7D9171` in little-endian

[2]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/main.c#L78`

[3]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/main.c#L366`

[4]`https://docs.google.com/spreadsheets/d/1aphqlyBsOSbV8ofCgYJlZNo2-GQ3CV6BmYxwak9xiTg/edit?usp=sharing`

[5]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/main.c#L267`

format, as the 32-bit target value since this 32-bit sequence will be present in all BLE advertising packets.

Thus, the 32-bit shift register is continuously updated with the next demodulated bit once every 1 µs for BLE. When the Hamming distance between the 32-bit shift register and the 32-bit target value is below some threshold, a `RAWCHIPS_STARTVAL` interrupt is triggered. The Hamming distance can be set using the last five bits of `ANALOG_CFG_REG__3`[6]. In general, `ANALOG_CFG_REG__3[6:0] = {clear_interrupt_startval, clear_interrupt32, threshold[4:0]}`.

Once a `RAWCHIPS_STARTVAL` interrupt[7] has been triggered, for every successive 32 bits that are shifted through the 32-bit shift register, a `RAWCHIPS_32` interrupt[8] is triggered. This allows us to theoretically piece together the raw BLE packet bits, where the 32-bit access address is found in the first `RAWCHIPS_STARTVAL` interrupt and each successive 32-bit blocks can be read during the `RAWCHIPS_32` interrupts.

To read the data in the shift register, there is a buffer between the 32-bit shift register and the Cortex M0 microprocessor, such that we can access the buffer using the memory-mapped registers `ANALOG_CFG_REG__17`, containing the 16 LSBs, and `ANALOG_CFG_REG__18`, containing the 16 LSBs of the buffer. However, there is some error in the hardware when copying the bits from the 32-bit shift register to the buffer, possibly a hold time violation, that causes the buffer to contain erroneous bits. In fact, only around the first 20 to 25 bits of each BLE packet can be recovered perfectly. Therefore, with the SCµM3C boards, we can unfortunately only verify whether we have received a certain 32-bit sequence instead of recovering the entire BLE packet.

If we want to test the Hamming distance and the interrupts without the recovered data and clock signals from the ZCC module, we can set the ASC bits 269 and 270 to be high[9], so that the recovered data and clock signals are taken from the GPIOs. The GPIO inputs are in bank 2 as `DATA_IN` and `DATA_CLK_IN`.

5.2 Results

For the BLE RX experiments, we used SCµM development board QX7 since the GPIO pins on board Q4 are non-functional. Unfortunately, since QX7 does not have an FTDI RS232/UART chip, we rely on raising GPIO pins as flags for having received a BLE packet.

Before attempting to receive BLE packets using the ZCC module, we first used an Analog Discovery 2 to generate a 1 MHz baseband clock with the dummy bits of a BLE packet as the data signal. In `RAWCHIPS_STARTVAL_ISR`, we would then set a GPIO pin high as a flag that

[6]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/main.c#L274`

[7]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/Int_Handlers.h#L714`

[8]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/Int_Handlers.h#L645`

[9]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/scm_ble_functions.c#L441`

32-bit target value	Hexadecimal	Binary
Expected	`0x6B7D9171`	$\texttt{0b0110101101111101100100010\underline{1}110001}$
Actual	`0xD6FB2263`	`0b11010110111110110010001001100011`

Table 5.1: Recovered data vs. the 32-bit target value with a Hamming distance of 0. The `RAWCHIPS_STARTVAL` is still being triggered correctly, but there is a bit error when reading the recovered data from `ANALOG_CFG_REG__17` and `ANALOG_CFG_REG__18`.

the 32-bit access address was found within the generated data signal. We then tested the Hamming distance by flipping bits within the access address and verifying that the GPIO pin would no longer be set high if the number of bit flips exceeded the Hamming distance threshold.

In `RAWCHIPS_STARTVAL_ISR`, when we printed the contents of `ANALOG_CFG_REG__17` and `ANALOG_CFG_REG__18`, we noticed that spurious bit errors would occur after around the 20th bit, as shown in Table 5.1. This means that we would be able to verify whether the 32-bit target value was found in the recovered data, but we would not be able to recover the entire packet contents in software.

SCµM to SCµM

Afterwards, we first attempted to receive SCµM's BLE packets on another SCµM board. We used SCµM development board QX7 to transmit BLE advertising packets to SCµM development board Q4. We set the PA current to be at its maximum value on QX7, and we attached a WiFi antenna to each of the two SCµM development boards. As expected, the distance at which we could receive the BLE advertising packets varies with the Hamming distance threshold.

1. Hamming distance = 1: Q4 could pick up BLE packets from within a few millimeters.
2. Hamming distance = 2: Q4 could pick up BLE packets from within a few centimeters.
3. Hamming distance ≥ 3: Q4 starts to pick up erroneous packets due to noise.

Thus, setting the Hamming distance to 2 allows us to receive BLE packets over the longest distance without picking up too many erroneous packets. To characterize the frequency tolerance of SCµM, we set the Hamming distance to 2 and placed the antennas of Q4 and QX7 approximately 2 cm apart. We then swept through all mid and fine codes on Q4 while keeping the coarse code at 23 and recorded all pairs of mid and fine codes on which we received a BLE packet from QX7. The results are shown in Figure 5.2.

Finally, to verify that we were indeed receiving the entire BLE packet, we probed the recovered baseband clock and data signals with a Saleae digital logic analyzer. Using the baseband clock, we sampled the data signal and recovered the entire BLE packet. In the

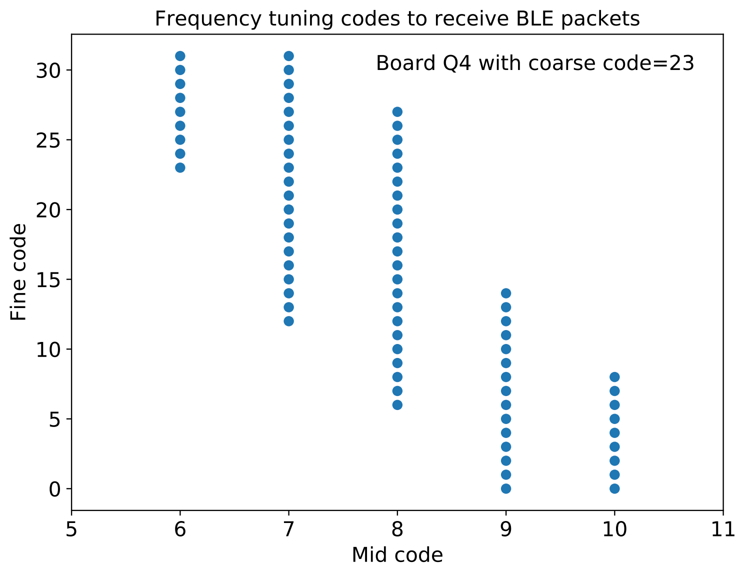


Figure 5.2: The LC frequency codes on SCμM development board Q4 on which we could receive BLE packets from board QX7.

entire 120-bit test BLE packet, the recovered BLE packet exhibited four bit errors, one of which was in the 32-bit access address. Thus, SCμM-to-SCμM BLE transmission works with some transmission errors.

Phone to SCμM

However, phone-to-SCμM BLE transmission is much more useful than SCμM-to-SCμM as 802.15.4 communication works between two SCμM boards. We used the nRF Connect for Mobile app[10] on a Google Pixel 2 XL to transmit BLE advertising packets. Using a mixer, we verified that the 32-bit target value consisting of the BLE advertising address was indeed present in the packet. We set the LO input of the mixer to a 2.402 GHz tone, attached an antenna to the RX input held next to the phone, and probed the IF output with an oscilloscope. Indeed, the 32-bit access address, `0x6B7D9171` in little-endian format, is present in the phone's BLE advertising packets.

Afterwards, we held the transmitting phone next to the antenna of SCμM, but we did not receive any BLE packets. We probed the recovered data and clock signals, and when correlating the sampled data with the 32-bit target value, the minimum number of bit errors within the 32-bit chunk was 10. When nothing was transmitting next to SCμM, the minimum

[10]`https://web.archive.org/web/20200523001918/https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Connect-for-mobile`. Archived from `https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Connect-for-mobile` on May 22, 2020.

number of bit errors within the 32-bit chunk was 8. This implies that the GFSK BLE packets from the phone look like noise to SCµM's ZCC module.

One possible explanation for SCµM's inability to pick up GFSK BLE packets is shown in Figure 5.1, where the packet error rate for GFSK is much higher than for FSK at similar input power. Future work will involve using a waveform generator to generate a GFSK-modulated signal of `0xAAAAAAAA` with a bit rate of 1 Mbit/s and probe what SCµM's recovered data and clock signals look like.

5.3 Conclusion

In this chapter, we described the current progress on receiving BLE packets on SCµM. We showed how we can demodulate incoming BLE signals by using the ZCC module. The recovered data is then fed through a 32-bit shift register, where we check the Hamming distance between a certain 32-bit target value and each 32-bit sequence in the recovered data bitstream. Since the access address, `0x6B7D9171` in little-endian format, is not whitened in BLE packets, we set it as the 32-bit target value. SCµM-to-SCµM BLE transmission works, although at a small range, but phone-to-SCµM transmission does not work. The most likely root cause is that SCµM's BLE packets are modulated using FSK while the phone's BLE packets are modulated using GFSK.

Details on the software for BLE RX can be found in Appendix C.

Chapter 6

Conclusion

In this thesis, we presented some work on the Single Chip Micro Mote (SCµM). We described how it is programmed using an infrared diode and how the on-chip oscillator frequencies are calibrated using 100 ms optical start frame delimiter (SFD) interrupts. We then discussed the oscillators on SCµM and characterized their frequency stability. In particular, we showed that the frequency of the 2.4 GHz LC oscillator can be tuned using a 15-bit LC frequency tuning code, but the LC frequency is not monotonic with respect to this tuning code. Thus, we presented two approaches at creating an LC monotonic function in order to facilitate future LC frequency compensation.

Since the LC oscillator has a temperature coefficient of around −40 ppm/°C [9], we described a method to estimate the ambient temperature by finding a linear relationship between the temperature and the ratio of the 2 MHz chipping clock for the chip's transmitter and the 32 kHz oscillator. Using a two-point calibration, the coefficients of the linear model can be easily determined, and after averaging over 5 temperature samples, we showed that the error of the mote's measured temperature is less than 2 °C between 0 °C and 100 °C and less than 1 °C between 5 °C and 85 °C. This allows SCµM to generate fairly accurate temperature estimates and perform temperature compensation using these estimates, which eliminates the need for network-based compensation.

Afterwards, we showed how we could create a Bluetooth Low Energy (BLE) beacon and a 802.15.4-to-BLE translator using SCµM that operates across temperature. After tuning the LC frequency via a 15-bit frequency tuning code using 100 ms SFD interrupts, the naive solution is to sweep the fine code of the LC frequency tuning code while transmitting BLE-compliant packets to compensate for temperature changes. To be efficient, though, we discussed two possible approaches. We can use a temperature sweep to find a linear model that determines the fine code to use at each temperature. Otherwise, we can use network compensation by either measuring the intermediate frequency (IF) offset of incoming 802.15.4 packets from an external OpenMote to tune both the 802.15.4 RX and BLE TX tuning codes or by averaging over the fine codes of incoming 802.15.4 packets to correct any LO frequency drift. While measuring the IF offset allows for real-time frequency compensation, it performs worse with a large LO frequency drift. Either of these methods allows for BLE transmission within a temperature range of around 20 °C without rolling over from the fine code to a mid code, which causes monotonicity issues.

6.1 Future Work

IF compensation is more robust than using the temperature estimate when adjusting the LC frequency tuning code. The next step is to have SCμM transmit BLE packets over a temperature range from 0 °C to 100 °C using IF compensation. Since 32 fine codes allow SCμM to operate as an 802.15.4-to-BLE translator over a range of 20 °C, one could repeat LC frequency calibration at intervals of 20 °C between 0 °C and 100 °C. In other words, we would find the coarse and mid codes for BLE TX and 802.15.4 RX at, for example, 10 °C, 30 °C, 50 °C, 70 °C, and 90 °C. During normal operation as a 802.15.4-to-BLE translator, SCμM would then use IF compensation to adjust the BLE TX and 802.15.4 RX frequency tuning codes within the 32 fine codes. When the fine code rolls over, we would change the coarse and mid codes to the ones corresponding to the next 20 °C interval. In general, though, an LC monotonic function would eliminate the need for calibration at multiple temperatures.

It is also possible to combine both the IF offset and RX fine code averaging approaches to perform network compensation for BLE TX. Since RX fine code averaging performs better with a large LO frequency drift, SCμM could halt 802.15.4-to-BLE translation and sweep the RX fine code for incoming 802.15.4 packets when it receives fewer 802.15.4 packets than expected. Otherwise, it will continue using the IF offset approach as this is more suited for real-time frequency compensation.

Finally, all of the experiments described in this work have the LC divider turned on as the LC divider is needed to measure and calibrate the LC frequency. However, the LC divider consumes a third of SCμM's power, as shown in Figure 5.21 in [9]. Therefore, in order to reduce power consumption, the next step would be to find a way to calibrate SCμM to operate over a wide temperature range without the need for the LC divider.

Bibliography

[1] D. C. Burnett et al. "CMOS oscillators to satisfy 802.15.4 and Bluetooth LE PHY specifications without a crystal reference". In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. Jan. 2019, pp. 0218–0223. DOI: 10.1109/CCWC.2019.8666473.

[2] David Burnett. "Crystal-free wireless communication with relaxation oscillators and its applications". PhD thesis. EECS Department, University of California, Berkeley, Apr. 2019. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-5.html.

[3] T. Chang et al. "QuickCal: Assisted Calibration for Crystal-Free Micro-Motes". Submitted to *IEEE Internet of Things Journal*. 2020.

[4] Tengfei Chang et al. "6TiSCH on SCμM: Running a Synchronized Protocol Stack without Crystals". In: *Sensors* 20.7 (Mar. 2020), p. 1912. ISSN: 1424-8220. DOI: 10.3390/s20071912.

[5] D.S. Contreras and Kristofer Pister. "A Six-Legged MEMS Silicon Robot Using Multichip Assembly". In: May 2018, pp. 54–58. DOI: 10.31438/trf.hh2018.15.

[6] D. S. Drew et al. "Toward Controlled Flight of the Ionocraft: A Flying Microrobot Using Electrohydrodynamic Thrust With Onboard Sensing and No Moving Parts". In: *IEEE Robotics and Automation Letters* 3.4 (2018), pp. 2807–2813.

[7] D. Griffith et al. "A 190nW 33kHz RC oscillator with ±0.21% temperature stability and 4ppm long-term stability". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. Feb. 2014, pp. 300–301. DOI: 10.1109/ISSCC.2014.6757443.

[8] F. Maksimovic et al. "A Crystal-Free Single-Chip Micro Mote with Integrated 802.15.4 Compatible Transceiver, sub-mW BLE Compatible Beacon Transmitter, and Cortex M0". In: *2019 Symposium on VLSI Circuits*. June 2019, pp. C88–C89. DOI: 10.23919/VLSIC.2019.8777971.

[9] Filip Maksimovic. "Monolithic Wireless Transceiver Design". PhD thesis. EECS Department, University of California, Berkeley, May 2020. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-33.html.

[10] Sahar Mesri. “Design and User Guide for the Single Chip Mote Digital System”. MA thesis. EECS Department, University of California, Berkeley, May 2016. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-71.html`.

[11] S. Park, C. Min, and S. Cho. “A 95nW ring oscillator-based temperature sensor for RFID tags in 0.13μm CMOS”. In: *2009 IEEE International Symposium on Circuits and Systems*. May 2009, pp. 1153–1156. DOI: `10.1109/ISCAS.2009.5117965`.

[12] I. Suciu et al. “Dynamic Channel Calibration on a Crystal-Free Mote-on-a-Chip”. In: *IEEE Access* 7 (2019), pp. 120884–120900. ISSN: 2169-3536. DOI: `10.1109/ACCESS.2019.2937689`.

[13] I. Suciu et al. “Experimental Clock Calibration on a Crystal-Free Mote-on-a-Chip”. In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Apr. 2019, pp. 608–613. DOI: `10.1109/INFCOMW.2019.8845103`.

[14] B. Wheeler et al. “A Low-Power Optical Receiver for Contact-free Programming and 3D Localization of Autonomous Microsystems”. In: *2019 IEEE 10th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*. 2019, pp. 0371–0376.

[15] B. Wheeler et al. “Crystal-free Narrow-band Radios for Low-cost IoT”. In: *2017 IEEE Radio Frequency Integrated Circuits Symposium (RFIC)*. 2017, pp. 228–231.

[16] Bradley Wheeler. “Low Power, Crystal-Free Design for Monolithic Receivers”. PhD thesis. EECS Department, University of California, Berkeley, May 2019. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-36.html`.

Appendix A

Temperature Estimation

All of the code can be found on the `titan/ble_temp_sweep` branch of the repository found here: `https://github.com/tryuan99/scum-test-code`.

For the temperature estimation, I used a TestEquity Model 107 temperature chamber. Although the chamber has a temperature controller by itself to enable temperature ramps, it is not very accurate because the actual temperature in the chamber lags behind the displayed temperature. Therefore, I used a SparkFun TMP102 digital temperature sensor connected to a Teensy 3.6 microcontroller as the reference temperature sensor. The `.ino` file for the Teensy 3.6 microcontroller with the pin connections for the TMP102 breakout board can be found in `scum/temp_sense/temp_sense_v2/temp_sense_v2.ino`. To read the measured temperature on a computer, run the Python script found in `scum/counter_temp/read_temp.py` with `python read_temp.py` to measure the temperature once every second.

All of the code for SCµM can be found in `scm_v3c_BLE`. To enable temperature calibration, set the flag `sweeping_2M_32k_counters = true` in `main.c`. If this flag is true, SCµM will continuously measure the 2 MHz and the 32 kHz oscillator frequency counts over an interval of 50,000 cycles of RFTimer, which corresponds to approximately 100 ms, and print the frequency counts over UART.

After programming SCµM, the UART output from SCµM should look like the following:

```
count_32k: 3213, count_2M: 200014
count_32k: 3212, count_2M: 200003
count_32k: 3212, count_2M: 200031
count_32k: 3213, count_2M: 200022
count_32k: 3211, count_2M: 199953
count_32k: 3214, count_2M: 200065
count_32k: 3212, count_2M: 200010
```

Place SCµM into the temperature chamber with the TMP102 digital temperature sensor. Run the Python script found in `scum/counter_temp/2M_32k_counter_temp.py` with `python 2M_32k_counter_temp.py` to read the 2 MHz and 32 kHz frequency counts and the reference temperature together. The rate at which these three values are read is limited by the period between the frequency count measurements on SCµM, nominally around 100 ms.

We can now perform a temperature sweep by adjusting the chamber temperature, during which the frequency counts and the chamber temperature will be recorded. Since the development PCB for SCµM has a large thermal mass, I limited the temperature ramp rate to 1.5 °C/min or less to minimize hysteresis. If a two-point calibration is preferred, wait for a few minutes until the measured reference temperature has stabilized. When quitting the Python script, all of the data will be written to a NumPy `.npz` file.

To find the linear model, I used a linear regression. In Python, I opened the `.npz` file, loaded the 2 MHz and 32 kHz frequency counts, and calculated their ratios. I then used SciPy to find the linear regression that relates the reference temperature to the frequency ratio as shown in `scum/counter_temp/2M_32k_counter_temp.ipynb`.

The coefficients for the linear model can now be set in SCµM's firmware[1] to estimate the temperature. The firmware already implements averaging over five ratio measurements before updating the temperature to increase the precision of the temperature estimate. Finally, the temperature estimate can be used to tune the fine code of the local oscillator and is also included in the BLE packet.

[1]`https://github.com/tryuan99/scum-test-code/blob/cf82f44c9b990296e95a0af586c22b51c2e766da/scm_v3c_BLE/Int_Handlers.h#L663`

Appendix B

BLE TX

All of the code can be found on the `titan/SW-23` branch in the `scm_v3c` directory of the repository found here: `https://github.com/tryuan99/scum-test-code`. An older version of the code before refactoring can be found on the `titan/ble_temp_sweep` branch of the same repository.

A sample application to transmit BLE packets is the `applications/ble_tx` project. All BLE-related functions can be found in `ble.c` and the corresponding header file `ble.h`. In particular, `ble_init_tx()` should be called after calling `initialize_mote()` in the application file. `ble_init_tx` sets the corresponding analog scan chain bits to configure the radio to enable BLE modulation and initialize the FIFO for BLE transmission.

To generate a BLE packet, we can use `ble_gen_test_packet` to generate a dummy BLE test packet that simply has `SCUM3` as the short name. Otherwise, to customize the contents of SCμM's BLE packet, call `ble_set_[data]([value])` to change the data values and `ble_set_[data]_tx_en(true)` to include this data in the BLE packet, where `[data]` represents one of the following BLE packet content types:

1. `name` for the short name
2. `lc_freq_codes` for the LC frequency tuning codes
3. `counters` for the 2 MHz and 32 kHz frequency counts
4. `temp` for the temperature
5. `data` for four bytes of custom data

Afterwards, call `ble_gen_packet()` to generate the BLE packet.

For example, to generate a BLE packet with the short name `HELLO` and the temperature 20 °C, run the following:

```c
char name[5] = {'H', 'E', 'L', 'L', 'O'};
ble_set_name(name);
```

```
ble_set_name_tx_en(true);
ble_set_temp(20.00);
ble_set_temp_tx_en(true);
ble_gen_packet();
```

Before generating the BLE packet, we can also modify the channel on which to transmit the BLE packet using the function `ble_set_channel`, and we can modify the advertiser address with the function `ble_set_AdvA`.

By default, the short name `SCUM3` is always included in the BLE packet, and the packet is transmitted on channel 37 (2.402 GHz) with the advertiser address `0x0002723280C6`.

To transmit a BLE packet, first find the appropriate LC frequency tuning codes for the SCµM board through LC frequency calibration, as described below, or with a spectrum analyzer. Replace the corresponding LC frequency codes in `ble_tx.c`[1]. Transmitting the BLE packet involves just calling `ble_transmit_packet()` after the BLE packet has been generated.

If you would like to sweep the fine code and transmit one BLE packet for each of the 32 fine codes, set `#define BLE_SWEEP_FINE true` at the top of `ble_tx.c`. This is helpful if SCµM's BLE packets are not being received and you suspect that its LC frequency codes might not be tuned correctly.

B.1 LC Frequency Calibration

To enable LC frequency calibration with the 100 ms optical start frame delimiter (SFD) interrupts immediately after programming, set `#define BLE_CALIBRATE_LC true` at the top of `ble_tx.c`. Before enabling optical calibration, call `optical_setLCTarget(250182)`[2] to set the target LC frequency count, which is divided by 960, within 100 ms to 250182, which corresponds to a frequency of 2.401 75 GHz. This value may need to be adjusted for different SCµM chips. For example, for SCµM development board Q4, I set the target LC frequency count to 250020 in order for LC frequency calibration to find the correct frequency codes for BLE transmission.

After programming, SCµM will then find the best coarse and mid LC frequency codes with a fine code of 15, such that the LC frequency count within 100 ms is closest to the target LC frequency count[3]. In order to retrieve the found coarse, mid, and fine codes, we

[1]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/applications/ble_tx/ble_tx.c#L116`

[2]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/applications/ble_tx/ble_tx.c#L96`

[3]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/optical.c#L199`

call `optical_getLCCoarse()`, `optical_getLCMid()`, and `optical_getLCFine()`[4].

The coarse, mid, and fine codes found with optical LC frequency calibration are printed to UART as well. I have found that LC frequency calibration usually produces the same result after each calibration run, so it is sufficient to simply run LC frequency calibration once and use the calibrated frequency codes for subsequent programmings. Replace the corresponding hard-coded LC frequency codes in `ble_tx.c`[5] and turn off LC calibration for future programmings.

B.2 Fine Code Calibration

Fine code calibration was only performed using the older version of the code, which can be found on the `titan/ble_temp_sweep` branch.

We first confirmed that we could receive BLE packets at room temperature on a smartphone and then re-programmed SCµM to sweep through all fine codes, transmitting one BLE packet for each code, i.e., by setting `sweep_fine_codes = true` in `main.c`. We also set the last byte of the advertiser address to be equal to the fine code. This way, when the smartphone receives multiple BLE packets from SCµM at different fine codes, we can easily determine on which fine codes we could receive a packet.

To find the linear model that dictates which fine code at each temperature, we placed SCµM into a temperature chamber. We slowly decreased the chamber temperature and verified that we were still receiving some BLE packets from SCµM. For example, on SCµM development board Q4, at 20 °C, we received packets with fine codes 12 through 20, but at 9 °C, we only received BLE packets with fine codes between 0 and 4. Similarly, at the other end of the temperature range, we received BLE packets with fine codes between 28 and 30 at 31 °C.

To find the linear model, we declined to use a linear regression on all of the data points since a linear regression will be biased toward temperatures at which we could receive BLE packets over a wide range of fine codes. Instead, we applied linear regression on the means of the fine codes at each temperature.

We then turned off fine code sweeping and re-programmed SCµM with the coefficients we found[6].

[4]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/applications/ble_tx/ble_tx.c#L112`

[5]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/applications/ble_tx/ble_tx.c#L116`

[6]`https://github.com/tryuan99/scum-test-code/blob/cf82f44c9b990296e95a0af586c22b51c2e766da/scm_v3c_BLE/main.c#L368`

B.3 RX Tracking

The `applications/ble_tx_154_rx` project lets SCµM operate as a 802.15.4-to-BLE translator. SCµM will keep listening for 802.15.4 packets from OpenMote and record the first four bytes of each received 802.15.4 packet. Every `TIMER_PERIOD` cycles of the RFTimer clock, SCµM will then transmit a BLE packet with the four bytes of the most recently received 802.15.4 packet as the custom data within the packet. Nominally, we set `TIMER_PERIOD` to 200000, so SCµM sends one BLE packet around every 400 ms.

Note that for this translator application, we keep track of two LC frequency settings, one for receiving the 802.15.4 packets, called `rx_coarse`, `rx_mid`, and `rx_fine`, and one for transmitting BLE packets, called `tx_coarse`, `tx_mid`, and `tx_fine`. Additionally, note that when we receive a 802.15.4 packet, we can read the IF estimate of the packet and the LQI error rate by calling `radio_getIFestimate()` and `radio_getLQIchipErrors()`[7].

In order to use the received 802.15.4 packets to adjust the LC frequency code of BLE TX, there are two additional projects.

The `applications/ble_tx_154_rx_track_if` project uses the IF estimates of all received 802.15.4 packets and stores them in the array `IF_estimate_history`[8] by calling the function `radio_update_IF_estimate(IF_estimate, LQI_chip_errors)`. It then convolves all of the IF estimates with the Gaussian FIR filter. In `ble_tx_154_rx_track_if.c`, every `CALIBRATE_PERIOD * TIMER_PERIOD` cycles of the RFTimer clock, we call the function `calibrate_fine_code()`. Nominally, we set `CALIBRATE_PERIOD` to 2. In this function, we check whether SCµM has received enough 802.15.4 packets with a sufficiently low LQI error rate by calling `radio_get_IF_estimate_ready()`. If this is the case, then `radio_get_IF_estimate()` returns the IF offset, with which we can adjust the 802.15.4 RX and the BLE TX frequency codes[9]. Note that we adjust the frequency codes by one fine code for every IF offset of 12[10]. An IF offset of 20 corresponds to a frequency offset of 100 kHz, approximately the frequency difference between two successive fine codes, so 12 is a little more than halfway to the next fine code.

The other project, `applications/ble_tx_154_rx_track_mean`, adjusts the 802.15.4 RX and BLE TX frequency codes by listening for 802.15.4 packets every `CALIBRATE_PERIOD * TIMER_PERIOD` cycles of the RFTimer clock on all fine codes within `CALIBRATE_RX_DIFF` of the current fine code. For example, nominally, `CALIBRATE_RX_DIFF` is set to 2, so we listen on five fine codes for 802.15.4 packets. On each fine code, we spend `2 * TIMER_PERIOD` cycles of RFTimer listening for packets, which corresponds to around 400 ms on each fine code nominally. We record the number of 802.15.4 packets we receive on every fine code, and

[7]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/applications/ble_tx_154_rx/ble_tx_154_rx.c#L151`

[8]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/radio.c#L24`

[9]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/applications/ble_tx_154_rx_track_if/ble_tx_154_rx_track_if.c#L286`

[10]`https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/applications/ble_tx_154_rx_track_if/ble_tx_154_rx_track_if.c#L293`

at the end of this procedure, if we have received at least `CALIBRATE_MIN_SUCCESS` packets, nominally set to 10, we find the weighted average of the fine codes of all received 802.15.4 packets. We set the new 802.15.4 RX fine code to be equal to this weighted average, and we adjust the BLE TX fine code by the same difference[11].

[11] `https://github.com/tryuan99/scum-test-code/blob/483cea44ce66808409eeed8b5d9f2ab0ec53ae10/scm_v3c/applications/ble_tx_154_rx_track_mean/ble_tx_154_rx_track_mean.c#L316`

Appendix C

BLE RX

All of the code can be found on the `titan/ble_rx` branch of the repository found here: `https://github.com/tryuan99/scum-test-code`.

The code for SCµM can be found in `scm_v3c_BLE`. BLE RX from commercial off-the-shelf BLE devices does not work on SCµM, but SCµM-to-SCµM BLE transmission works. To listen for the incoming BLE packets, we configure SCµM to sweep through some LC frequency codes to find a frequency setting on which it can receive BLE packets. The 32-bit target value is set to the access address of BLE advertising packets by setting `ANALOG_CFG_REG__1 = 0x9171; ANALOG_CFG_REG__2 = 0x6B7D;`[1], and the Hamming distance is nominally set to 2 with `uint8_t hamming_distance = 2U`.

The `RAWCHIPS_STARTVAL_ISR` interrupt routine[2] is then executed when the 32-bit shift register has a Hamming distance equal to or less than `hamming_distance` from the 32-bit target value. For every subsequent 32-bit value that is shifted into the register, the `RAWCHIPS_32_ISR` interrupt routine[3] is executed.

If no BLE packets are being received, check that the recovered digital baseband clock frequency is within 1% of 1 MHz. The recovered clock is available as the GPIO output `mux_out_M0_clk` on pin 1 of bank 4[4]. Adjust the `IF_clk_target`[5] until the recovered clock frequency is within 1% of 1 MHz.

If you would like to bypass the ZCC module and input the recovered data and clock signals via the GPIO pins, in the function `radio_init_rx_ZCC_BLE`, replace `clear_asc_bit(269);`

[1]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/main.c#L267`

[2]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/Int_Handlers.h#L714`

[3]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/Int_Handlers.h#L645`

[4]`https://docs.google.com/spreadsheets/d/1aphqlyBsOSbV8ofCgYJlZNo2-GQ3CV6BmYxwak9xiTg/edit?usp=sharing`

[5]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/main.c#L78`

`clear_asc_bit(270);` with `set_asc_bit(269); set_asc_bit(270);`[6]. Furthermore, in the `initialize_mote_ble` function, we need to configure the GPIO pins corresponding to `DATA_IN` and `DATA_CLK_IN`, pins 1 and 2 of bank 2, as inputs. Thus, replace `GPI_enables(0x0000); GPO_enables(0xFFFF);` with `GPI_enables(0x000E); GPO_enables(0xFFF1);`. After programming SCμM, if the 32-bit target value is in the inputted data signal, the `RAWCHIPS_STARTVAL_ISR` interrupt routine should be executed.

[6]`https://github.com/tryuan99/scum-test-code/blob/7c208655add9f637837ec6e4b91dd4309c911fe1/scm_v3c_BLE/scm_ble_functions.c#L441`