Single Chip micro Mote
SCμM-3C
User Guide

Lydia Lee, Fil Maksimovic, Alex Moreno, Kris Pister,
Titan Yuan, Brad Wheeler

Last Compiled October 1, 2019

Contents

1 Intro

The goal of the Single Chip micro Mote project is to develop a complete self-contained wireless sensor node on a single chip, including sensing, computation, communication, and power, with no external components. SCμM3C is an important step along that path, containing everything needed for sensing, computation, and communication. This includes a 32 bit Cortex M0 processor, 2.4 GHz radio transceiver, and both a temperature sensor and a lighthouse location sensor. With a printed solar cell and battery, SCμM3C would satisfy the project goal. SCμM3C is the product of graduate research over a period of roughly six years by a core group of four students and a postdoc, and an extended group that includes dozens of graduate and undergraduate students and visiting scholars. The academic contributions are featured in three doctoral dissertations [1, 2, 4] and one master's thesis [3], as well as an expanding number of research publications describing the crystal-free approach [**xxx**], circuit design [**xxx**], and applications [**xxx**].

This User Guide is intended as a practical explanation of how to use SCμM3C. Because the chip was designed by students learning new skills and pursuing a research agenda, it has some quirks not found in typical production silicon. There are often optional ways of doing things that were included out of concern that new circuits would not work precisely as designed, and many cases where legacy mechanisms were left in place because there simply was not enough time to fix them. There are also a large number of debugging pads and interfaces. And of course, not everything works.

Because this chip is a research project not a commercial product, the documentation is not as well organized or complete as it might be. Work is in progress to improve that situation. This document attempts to consolidate information useful for operating and debugging on SCM-3C.

For Your Safety: As discussed later in the optical bootloading section, be careful with the optical programmers! They use high intensity infrared non-visible light which can damage eyes and skin at very short distances or long exposure times.

For SCM's Safety: Take precautions against ESD by grounding yourself before handling SCM. If you are applying external connections be sure to adhere to the maximum voltage ratings which differ depending on the pin. Take precautions to avoid scratching the epoxy covering the chip as this can degrade optical programming if the photodiode is obscured.

2 Quick Start - Hello World

The TL;DR version of this guide. Each of the following steps are described in detail in this guide and the mentioned references. A high level summary of the steps to get started running code on SCM are:

- Download and install ARM Keil (see [3]).
- Download example software from GitHub (see Section 3.1).
- Download bootloader script from GitHub (see Section 3.1).
- Update bootloader script with COM port and binary file path.
- Obtain a SCM-3C test PCB.
- Obtain an optical programmer PCB and Teensy with correct firmware.
- Plug both boards in via USB and open a COM port to SCM.
- Align the programmer and execute the bootload script.
- SCM should print over UART and toggle its GPIOs.

3 Where To Go For Resources and Help

Documentation, code, and source files are located in several different places which are listed below.

3.1 GitHub

Software for version 3C is hosted on GitHub or here: `https://github.com/PisterLab/scum-test-code`

3.2 Box

Bootloader scripts, Teensy bootloader firmware, scripts for utilizing scan chains, and PCB documentation can be found on Box in the uRobots/SCM_3C folder. Documentation can also be found in this folder. A detailed description of the Cortex-M0 and its associated digital blocks can be found in [3]. The hardware implementation of the radio and optical receivers is detailed in [4]. The RF local oscillator, transmitter, and divider are discussed in [2]. Clocks, POR, and reset logic are discussed in [1]. A useful book describing various aspects of the ARM Cortex-M0 can be found here [5]. Permission to access this folder must be given by someone within the group.

3.3 BWRC Repo

The source Verilog and digital synthesis flow used to create the SCM ASIC are located in this repo which is private due to IP restrictions. This repo, the analog portion of the design, and the final GDS reside on BWRC servers.

3.4 OpenWSN Repo

The OpenWSN repo which supports SCM can be found here: `https://github.com/openwsn-berkeley/openwsn-fw`

3.5 JIRA

JIRA was used for bug tracking during tapeouts and contains historical details of issues and the status of their resolution: `https://openwsn.atlassian.net/projects/SCUM/issues`

3.6 EECS Repo

Some software still resides in the EECS Repo which was where software was stored at the beginning of the project: `repo.eecs.berkeley.edu/git/projects/pistergroup/singlechip-digital.git`

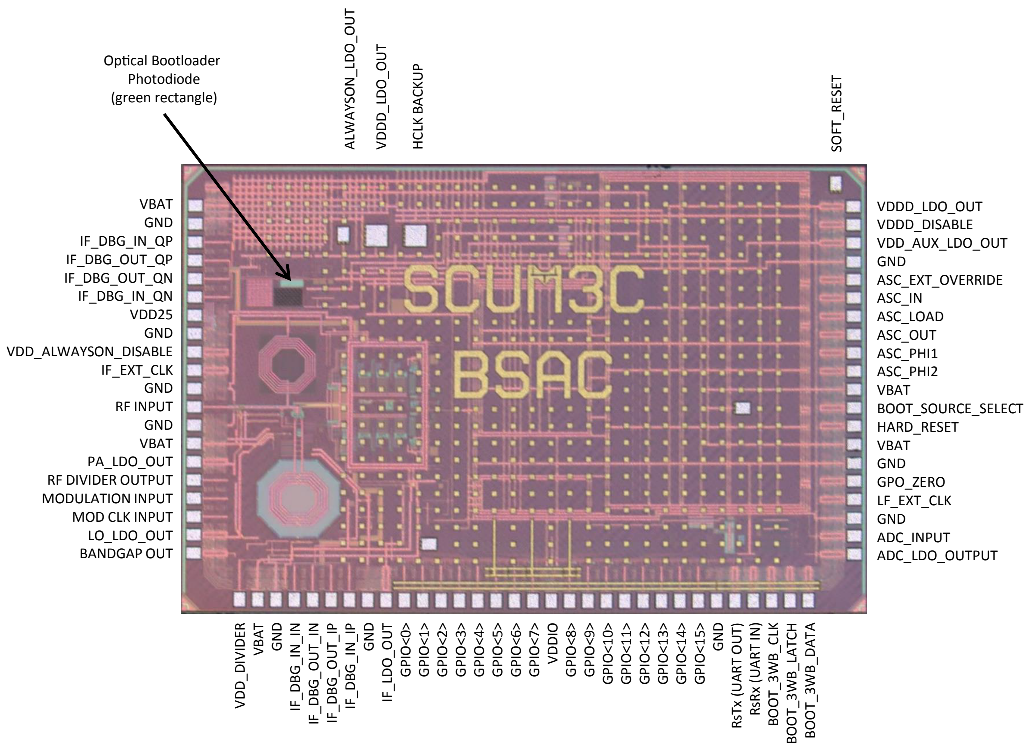


Figure 1: Die photo of SCM-3C with pad labels. Dimensions are 3mm × 2mm × 0.3mm.

4 Chip Overview

SCµM-3C is a 3x2x0.3 mm^3 silicon chip made in the TSMC 65nm LP RF MS CMOS process (Figure 1). It contains approximately 8 million transistors. SCµM-3C has 64 kB of program SRAM memory (IMEM) and 64 kB of data SRAM memory (DMEM). There are also 16 kB of program ROM which contain a bootloader. The process does not offer any writeable non-volatile storage, so every time that power is removed from the chip, both the program and data are lost.

Loading program memory can be done either optically or via a wired connection, as described in section 7. Once program memory is loaded, the chip needs to be configured for the desired functionality. SCµM-3C has a tremendous amount of configurability, most of which is not of interest to most users. For example, there are eight voltage regulators on the chip, each

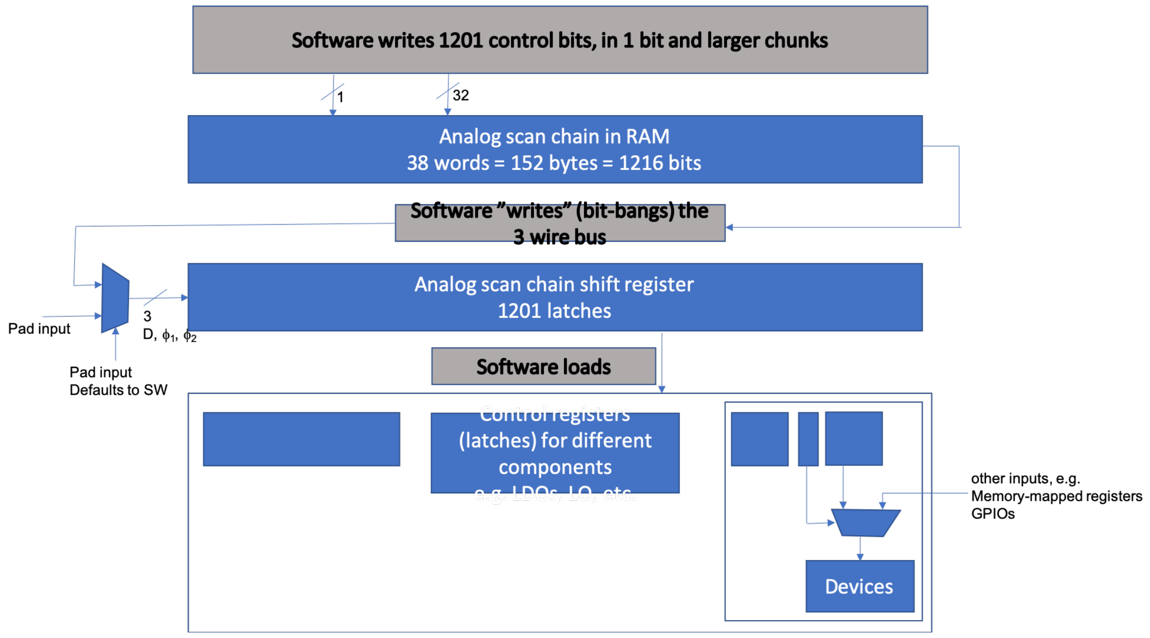

Software writes 1201 control bits, in 1 bit and larger chunks
1
32
Analog scan chain in RAM
38 words = 152 bytes = 1216 bits
Software "writes" (bit-bangs) the 3 wire bus
Analog scan chain shift register
1201 latches
Pad input
3
D, ϕ1, ϕ2
Pad input
Defaults to SW
Software loads
Control registers (latches) for different components e.g. LDOs, LO, etc.
other inputs, e.g.
Memory-mapped registers
GPIOs
Devices

Figure 2: The Analog Scan Chain.

with a tunable output voltage. Except in rare circumstances, most of these should simply be configured to the recommended setting. Some small subset of the configurable parameters is of high importance to users, such as the radio oscillator frequency or the processor clock speed.

Configuration bits control, for example, enabling and voltage settings on LDOs, LC oscillator frequency, which oscillators are connected to which clock lines, and how the chip's pads are connected to internal inputs and outputs. Configuration bits are controlled with a combination of the Analog Scan Chain (ASC) and Memory Mapped Registers (MMR).

4.1 Analog Scan Chain

The ASC is a shift register with 1201 bits that contains almost all of the configuration information for the chip. Changing configuration with the ASC requires that the entire desired sequence of bits be shifted into position and then loaded in parallel into control registers for the various operational units on the chip (Figure 2. This process can be accomplished either from off-chip through six pads on the East side of the chip, or via software on the

Figure 3: Pad Descriptions

microprocessor. By default, the processor is in control of the ASC.

At boot, the ASC registers default to all 0. Since some of the functional units that they control need to have non-zero values during boot, this can lead to some unusual coding of control words, but most of this is hidden from the user by a collection of functions designed to simplify interface to the ASC. For example, the VDDD, AUX_DIGITAL, and VDD_ALWAYS_ON voltage regulators invert the top two bits of their control word so that they are biased relatively high at boot. The ASC interface functions XXX() take this into account.

Programming the ASC requires several milliseconds, as the entire 1201 bit sequence must be shifted into the scan chain, and each bit requires several clock cycles to shift. For many operations, this is painfully slow, so a faster control mechanism is provided via memory mapped registers. Often the control authority selection between ASC and MMR is yet another ASC bit, as shown in Figure 2.

4.2 Memory Mapped Registers

4.3 Pads

As shown in Figure 1 there are pads on three sides of the chip, known for convenience as West, South, and East. Most of the East and West pads are for debugging purposes. One notable exception is the antenna pad on the West side, labeled RF INPUT. For size-constrained applications, this can be wire bonded to any floating pad on the chip, such as ASC_PHI1.

Supply pads with the same name are wired together on chip, and no issues have been found which would indicate that more than one bond wire is needed for VBAT (5 pads) or GND (9 pads). There are two separate pad voltage domains. VDDIO is used as the pad ESD and IO voltage domain for all of the GPIO, UART, and BOOT pads, all of which form a continuous line on the South side of the chip. All other pads use VBAT for ESD, and use a mix of VDDD and VBAT for the IO voltage. The specifics for each pad are in xxx-document.

The minimum number of pads that must be connected to boot the mote is two: VBAT and GND. For roughly half of the chips, the optical boot

Pad Name	Voltage Domain	Protection	comment
GND			
VBAT			
VDDIO			
RF INPUT			
GPIO	VDDIO	VDDIO	
UART	VDDIO	VDDIO	
3WB	VDDIO	VDDIO	
ADC_INPUT			
GPO0			
HARD_RESET			
BOOT_SOURCE_SELECT			
Less likely to be used by applications			
VDD25			
IF_*			
*_LDO_OUT			
RF_DIVIDER_OUT			
MOD CLK INPUT			
BANDGAP OUT			
VDD_DIVIDER			
LF_EXT_CLK			
ASC_*			
VDDD_*			

Table 1: Common pads and their IO voltage and protection voltage domains.

Domain	leakage	powers
VDDD	80	Cortex, RAM
AlwaysOn	45	Optical receiver
LO	13	
Sensor ADC	12	
Aux Digital	8	
IF		
PA		TX power amplifier

Table 2: Voltage domains

sequence described below will work with just these two pins connected. For the other half, VDDD_LDO_OUT must be briefly raised to VBAT in order for the chip to boot. Exact reason and timing are under investigation.

For wired bootloading, the minimum number of pads is six: VBAT, VDDIO, GND, and the three wire bus: BOOT_3WB_CLK, BOOT_3WB_DATA, and BOOT_3WB_LATCH. VDDIO may not be needed - this is still under investigation.

To make a minimal BLE or 802.15.4 transmitter sending data from the on-board temperature or lighthouse sensors, the antenna pad must also be connected. With a 3 mm wirebond across the chip, the antenna loss is approximately -25 dBi [**xxx**].

4.4 Voltage Domains

There are eight different voltage domains and five different bandgap references feeding them.

5 PCB Overview

The first implementation of the 3C software development board is shown in Figure 4. The SCM chips are packaged into QFN-100 packages from QuikPak to make them replaceable, reduce the board cost (relaxed trace/space), and to speed up board assembly. The PCBs use ENIG surface treatment so it is possible to directly wirebond to them although the bonds and angles can get rather extreme due to the large QFN landing. The boards also include a SMA connector to ease RF testing, all GPIOs broken out on 0.1” headers, and

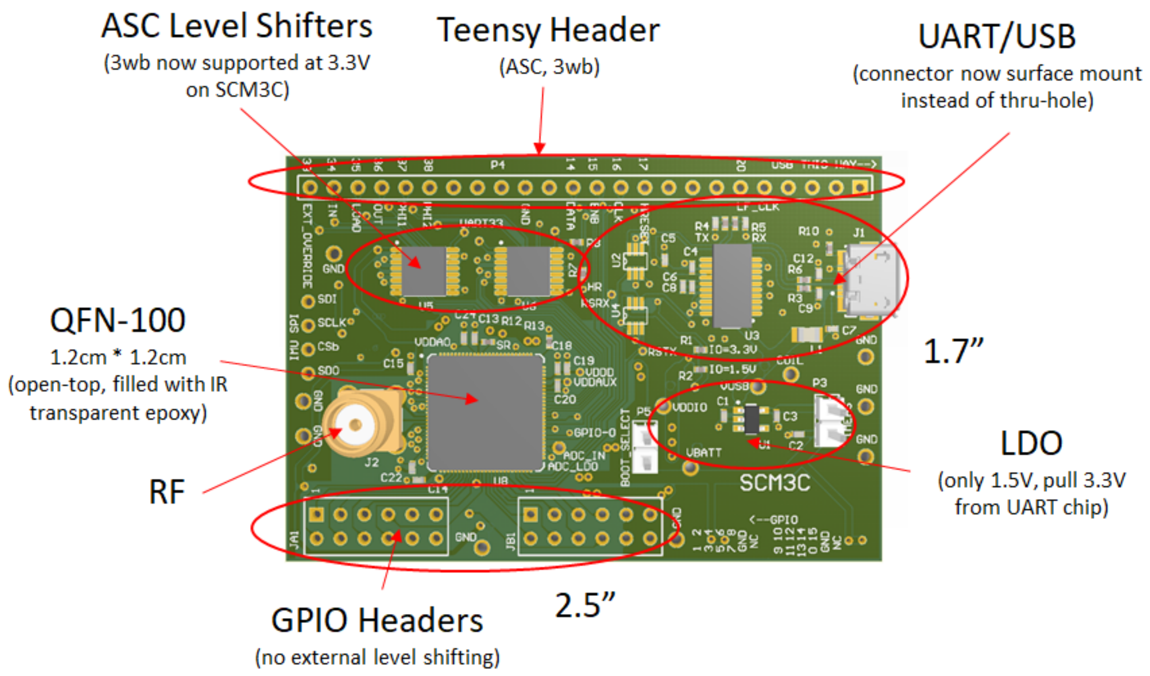

ASC Level Shifters
(3wb now supported at 3.3V on SCM3C)
Teensy Header
(ASC, 3wb)
UART/USB
(connector now surface mount instead of thru-hole)
QFN-100
1.2cm * 1.2cm
(open-top, filled with IR transparent epoxy)
1.7”
LDO
(only 1.5V, pull 3.3V from UART chip)
RF
GPIO Headers
(no external level shifting)
2.5”
SCM3C

Figure 4: SCM-3C Software Development PCB

UART/USB conversion. The boards are powered via the 5V USB connection and a 1.5V LDO is included to generate VBAT for SCM (with an inline jumper to make it easier to measure current). The FTDI UART/USB chip generates a 3.3V supply internally which can be used for VDDIO. A zero-ohm jumper on the board determines whether VDDIO on SCM is connected to the 3.3V or 1.5V supply domain. The Teensy header and ASC level shifters are not used since the analog scan chain can be controlled from the Cortex on chip and optical bootload is easier than wired programming. A future board revision could be made smaller by removing the Teensy header and level shifters as well as moving the UART/USB bridge off board - perhaps instead to one of FTDIs cable-based solutions. The back side of the PCB also has footprints included for an Invensense IMU (should support either the MPU-9250 or newer ICM-20948), a ADT7302 SPI temperature sensor, and a LTC4123 wireless battery charger. There is also a footprint for attaching a coin cell battery holder.

6 Electrical Specification

7 Bootloading

There are two bootload modes: optical and wired. The mote defaults to optical bootload mode which requires no external connections. The "bootload_source_select" pin is used to switch the mote to wired mode. This pin is internally pulled to ground selecting optical mode, and should be driven high (either to VDDD or VBAT) to select wired boot mode. See Appendix B in [4] for hardware details on the on-chip optical receiver. See [3] for details on the 3-wire bus mode.

7.1 Optical Bootloading

WARNING

The optical programmer uses a high intensity OSRAM SFH4555 non-visible infrared LED which outputs enough power to cause eye and skin damage at short range or prolonged exposure. As a general safety rule, always treat the programmer as if it were active, even when you know it is not. A visible red warning LED is included on the transmitter PCB to indicate when power is applied to the LED driver. It is recommended to turn off power to the transmitter Teensy when not in use.

For the approximately 1s duration of programming when the LED is active, the minimum safe distance is about 3 cm. A standoff is recommended to physically prevent getting closer than this to the LED. A section of a straw serves the purpose well and also provides a guide for aiming the LED at the chip.

Both optical and wired programming use a Teensy 3.6 for the programmer. The same Arduino firmware is used for both modes and is available on Box. When flashing the Teensy with code it is best to only have one Teensy plugged in at a time to ensure you are programming the correct device. Both MATLAB and Python are supported for transferring a compiled binary from Keil (the .bin file) to the Teensy which then programs the mote. The programming scripts are available on Box and should be updated with the appropriate COM port and path to the binary file. Flags set which bootload mode is used. The script also optionally allows for adding CRC checking to ensure the integrity of the software payload after it is transferred over optical into SRAM. The Keil project for SCM must include support for CRC checking (see software in git repo) as the length and CRC are written to memory locations that are predefined in the software. The bootloader script also has flags for choosing to fill unused program memory with zeros or random data. This can be useful for checking for errors during programmed or checking that SRAM is fully functional. The choice of skipping hard reset during optical program is also set via a script flag.

The optical transmitter is best used at distances < 5cm with care taken to align the IR LED to the mote due to its ± 5 degree half angle. The diode/straw of the optical programmer should be roughly aimed at the photodiode on the chip which is annotated as the small black rectangle in Figure 1. Details on the optical programmer PCB can be found in Section 23 and the PCB source files are available on Box. The presence of epoxy covering the chip wirebonds appears to slightly reduce the distance that the chip can be reliably programmed. Care should be taken to avoid damaging the epoxy above the chip or allowing the photodiode to be obstructed in any way. If difficulties are encountered when programming, try adjusting the position or vertical height of the programmer. A "helping hands" solder holder is very convenient for positioning the programmer.

The programmer will send a preamble to allow the on-chip receiver to settle, then send a start symbol, followed by dummy data to wait for hard reset to execute, and then the binary program data encoded with 4B/5B. The on-chip optical receiver will wait to recognize the start symbol and will issue a hard reset to get the chip to re-execute its ROM. The receiver will then decode the data, convert it to the same format as 3-wire bus, and load the program data into SRAM. Consult [4] for further details on the on-chip optical receiver.

The programmer provides the option of whether or not to issue a hard reset to the chip prior to bootloading. For including a hard reset, the start symbol is the sequence [169 176 167 50]. For skipping the hard reset the start symbol is [184 84 89 40]. If the reset is skipped then no dummy data is sent.

7.2 3-Wire Bus Bootloading

The wired mode requires three wires (data, clk, latch) and is henceforth referred to as 3wb for 3-wire bus. If the mote has already been programmed once then access to Hard Reset is also needed to re-execute the boot ROM. Alternatively, power cycling will also re-execute the ROM. To switch to 3wb mode the BOOTLOAD_SOURCE_SELECT pin must be pulled high to VBAT. A jumper position is provided on the PCB for this purpose. Note that pulling this pin high also disables the optical receiver analog frontend.

The same Teensy 3.6 microcontroller and firmware is used for 3wb bootloading. The clock, data, latch, and hard reset pins should be connected to their respective pins on SCM. Level-shifters are not required for clock, data, and latch on 3C whereas they were on earlier versions. The hard reset pin on SCM however will not tolerate 3.3V so either a level shifter is required or the Teensy should be configured as an open drain output for that pin (since the chip has an internal pullup). A compiled binary is loaded onto the mote using the same MATLAB or Python scripts which interface to the Teensy.

7.3 Mote Startup

When the mote first boots up it executes the following steps from ROM:

- Set BOOT_MODE to 3wb
- Assert GPIO-0 high
- Wait until 64 kB have been written to IMEM via 3wb interface
- Set BOOT_MODE to none (regular execution)
- Switch instruction execution from ROM to RAM
- Issue a soft reset

After soft reset has finished the mote should be running the software that was bootloaded into RAM. A useful troubleshooting step for debugging startup is to see if GPIO-0 is high. If it is then the mote is likely waiting to be bootloaded. If not, then something went wrong with the startup process. Other useful debug steps are to check that HCLK is coming out GPIO¡12¿ and that both hard (GPIO¡13¿) and soft (GPIO¡14¿) resets are high. It is recommended to consult [3] for further details.

7.4 Optical Boot Troubleshoot

More often than not when optical programming is being problematic the issue is alignment or distance related. Moving the programmer slightly and trying a few times is generally enough to resolve the issue. If problems persist then make sure there is nothing obstructing the photo diode. Take care to not make contact with the epoxy covering the chip as it can become marred and lead to issues with light reaching the photo diode.

A more detailed debugging step involves looking at the signals coming out of the optical receiver through the GPIO bank. When the mote first boots up it defaults to GPIO bank 0 which outputs OPTICAL_CLK_RAW and OPTICAL_DATA_RAW. Observing these two signals at the same time will shed some light on whether the alignment is correct. These signals should be time shifted versions of one another and the width of the narrow pulses should be shorter than the time shift between the two traces. Another debugging step is to check whether Hard Reset is being asserted during the programming process. If Hard Reset is occurring then at least some of the bits are being properly received but perhaps not the entire 64 kB is error free. If there is no Hard Reset then either something else is wrong or the alignment is so poor that even the first 32 bits are not being properly received.

It is possible to adjust the pulse width parameters that the Teensy uses when flashing the programming LED. Figure 5 shows the parameters that are used to set the on and off times for one and zero data bits. These values are passed to the Teensy in the bootloader script using the command 'configopt' followed by four integer values. These values are used as a for loop index that ultimately determines how wide each pulse is. The values {80,80,3,80} were determined experimentally to work relatively well but are unlikely to be optimum. The third value is likely to be the most beneficial to tweak as it determines the width of the zero bit pulse, which is the most critical for achieving error free operation.

8 Misc Optical Bootload Related Items

8.1 Optical Data Transfer

The optical receiver also provides a mechanism for wirelessly sending data to the mote after it has been programmed. Bits received over optical are clocked into a 32-bit shift register and an interrupt called "optical_irq_in"

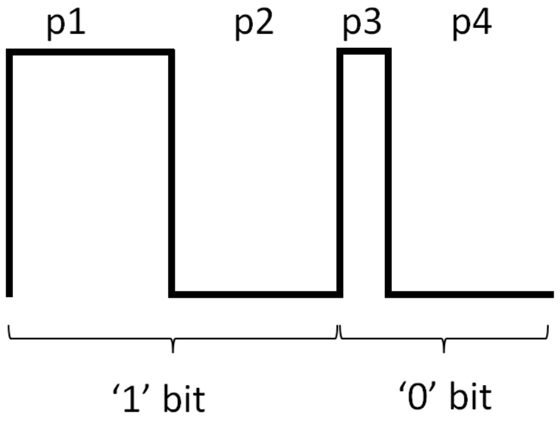


Figure 5: Variables describing pulse width properties of the Teensy optical transmitter.

is asserted every 32 new bits. A SFD (start frame delimiter) of [221 176 231 47] is also available to help synchronize the mote to the data sequence to be transferred. When the 32-bit register matches this value, a separate interrupt called "optical_sfd_interrupt" is asserted. By first waiting for the SFD interrupt and then reading the new data every time the "optical_irq_in" IRQ executes, one can send commands or transfer arbitrary amounts of data to the mote and have it stored in DMEM.

```
analog_rdata[335:304] = Optical 32-bit Register
```

8.2 Optical Timing Transfer

Since the mote needs programmed every time it loses power, bootloading provides an opportunity to also provide initial timing calibration. That can be accomplished by sending the optical SFD sequence at a fixed rate after bootload has completed. This will cause the optical SFD interrupt to go off at a known rate which can be used to calibrate the clocks on chip. The timing interval of the interrupt is based on the crystal-based Teensy clock and is thus much more accurate than SCM.

8.3 Lighthouse Localization Receiver

The optical receiver on SCM can also be used at limited range to receive localization pulses from a HTC Lighthouse. A demo of this capability is included in the Git repo.

8.4 Future Feature Idea: Hardware IDs & Calibration Data

While SCM has no hardware support for permanently setting identifying addresses on a per-node basis, this functionality could be added to the bootloader script. In the same way that the CRC is calculated and then inserted at a known memory location in the payload, an unique identifying address could be inserted during the programming process for individual nodes. The same technique could be used for inserting calibration tables into memory if desired.

8.5 Future Feature Idea: RF Bootloader

While not officially implemented or even attempted, it may be possible to implement an RF reprogramming capability by executing instructions out of data memory. The FPGA version of SCM will execute from DMEM but with strange behavior. The memory instantiations are not identical between ASIC and FPGA versions so it is possible that the ASIC version behaves differently when executing from DMEM (Modelsim indicates this is the case). If the ASIC version is capable of executing from DMEM, then it may be possible to copy new binary data into DMEM from whatever source desired, ie over the air via the radio, and then transfer execution to those new instructions. Note that using DMEM is required since IMEM becomes read-only after the initial bootload process. Also note that the mote would first need programmed with software to support this functionality so its utility is limited (ie this won't let you RF bootload from cold startup).

9 GPIO

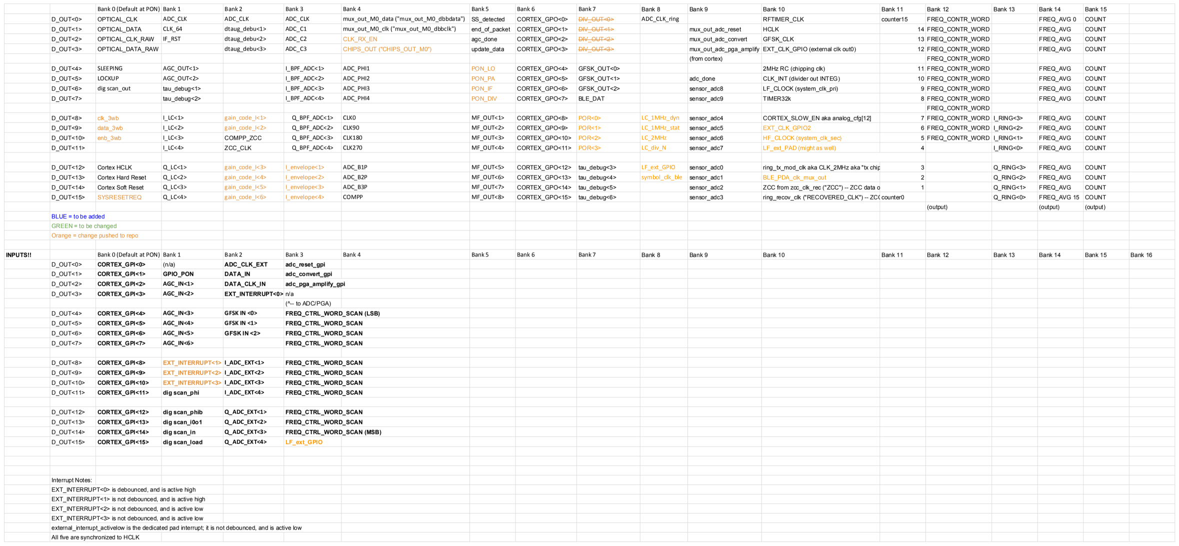
```
% GPIO Direction Control
% 1 = output, 0 = input
% out_mask<0:15> = gpio_direction;
% in_mask<0:15>  = 1 - gpio_direction;

% On-chip mapping is:
% out_en<0:15> = ASC<1131>,ASC<1133>,ASC<1135>,ASC<1137>,
%     ASC<1140>,ASC<1142>,ASC<1144>,ASC<1146>,
%     ASC<1115>,ASC<1117>,ASC<1119>,ASC<1121>,
%     ASC<1124>,ASC<1126>,ASC<1128>,ASC<1130>

% in_en<0:15> = ASC<1132>,ASC<1134>,ASC<1136>,ASC<1138>,
%     ASC<1139>,ASC<1141>,ASC<1143>,ASC<1145>,
%     ASC<1116>,ASC<1118>,ASC<1120>,ASC<1122>,
%     ASC<1123>,ASC<1125>,ASC<1127>,ASC<1129>

ASC(1131:2:1137) = 1 - out_mask(1:4);   //Outputs are active low
ASC(1140:2:1146) = 1 - out_mask(5:8);
ASC(1115:2:1121) = 1 - out_mask(9:12);
ASC(1124:2:1130) = 1 - out_mask(13:16);

ASC(1132:2:1138) = in_mask(1:4);        //Inputs are active high
ASC(1139:2:1145) = in_mask(5:8);
ASC(1116:2:1122) = in_mask(9:12);
ASC(1123:2:1129) = in_mask(13:16);
```



	Bank 0 (Default at PON)	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7	Bank 8	Bank 9	Bank 10	Bank 11	Bank 12	Bank 13	Bank 14	Bank 15
D_OUT<0>	OPTICAL_CLK	ADC_CLK	ADC_CLK	ADC_CLK	mux_out_M0_data ("mux_out_M0_dbbdata")	SS_detected	CORTEX_GPO<0>	~~DIV_OUT<0>~~	ADC_CLK_ring		RFTIMER_CLK	counter15	FREQ_CONTR_WORD		FREQ_AVG 0	COUNT
D_OUT<1>	OPTICAL_DATA	CLK_64	dtaug_debu<1>	ADC_C1	mux_out_M0_clk ("mux_out_M0_dbbclk")	end_of_packet	CORTEX_GPO<1>	~~DIV_OUT<1>~~		mux_out_adc_reset	HCLK	14	FREQ_CONTR_WORD		FREQ_AVG	COUNT
D_OUT<2>	OPTICAL_CLK_RAW	IF_RST	dtaug_debu<2>	ADC_C2	CLK_RX_EN	agc_done	CORTEX_GPO<2>	~~DIV_OUT<2>~~		mux_out_adc_convert	GFSK_CLK	13	FREQ_CONTR_WORD		FREQ_AVG	COUNT
D_OUT<3>	OPTICAL_DATA_RAW		dtaug_debu<3>	ADC_C3	CHIPS_OUT ("CHIPS_OUT_M0")	update_data	CORTEX_GPO<3>	~~DIV_OUT<3>~~		mux_out_adc_pga_amplify	EXT_CLK_GPIO (external clk out0)	12	FREQ_CONTR_WORD		FREQ_AVG	COUNT
										(from cortex)			FREQ_CONTR_WORD			
D_OUT<4>	SLEEPING	AGC_OUT<1>		I_BPF_ADC<1>	ADC_PHI1	PON_LO	CORTEX_GPO<4>	GFSK_OUT<0>			2MHz RC (chipping clk)	11	FREQ_CONTR_WORD		FREQ_AVG	COUNT
D_OUT<5>	LOCKUP	AGC_OUT<2>		I_BPF_ADC<2>	ADC_PHI2	PON_PA	CORTEX_GPO<5>	GFSK_OUT<1>		adc_done	CLK_INT (divider out INTEG)	10	FREQ_CONTR_WORD		FREQ_AVG	COUNT
D_OUT<6>	dig scan_out	tau_debug<1>		I_BPF_ADC<3>	ADC_PHI3	PON_IF	CORTEX_GPO<6>	GFSK_OUT<2>		sensor_adc8	LF_CLOCK (system_clk_pri)	9	FREQ_CONTR_WORD		FREQ_AVG	COUNT
D_OUT<7>		tau_debug<2>		I_BPF_ADC<4>	ADC_PHI4	PON_DIV	CORTEX_GPO<7>	BLE_DAT		sensor_adc9	TIMER32k	8	FREQ_CONTR_WORD		FREQ_AVG	COUNT
													FREQ_CONTR_WORD			
D_OUT<8>	clk_3wb	I_LC<1>	gain_code_I<1>	Q_BPF_ADC<1>	CLK0	MF_OUT<1>	CORTEX_GPO<8>	POR<0>	LC_1MHz_dyn	sensor_adc4	CORTEX_SLOW_EN aka analog_cfg[12]	7	FREQ_CONTR_WORD	I_RING<3>	FREQ_AVG	COUNT
D_OUT<9>	data_3wb	I_LC<2>	gain_code_I<2>	Q_BPF_ADC<2>	CLK90	MF_OUT<2>	CORTEX_GPO<9>	POR<1>	LC_1MHz_stat	sensor_adc5	EXT_CLK_GPIO2	6	FREQ_CONTR_WORD	I_RING<2>	FREQ_AVG	COUNT
D_OUT<10>	enb_3wb	I_LC<3>	COMPP_ZCC	Q_BPF_ADC<3>	CLK180	MF_OUT<3>	CORTEX_GPO<10>	POR<2>	LC_2MHz	sensor_adc6	HF_CLOCK (system_clk_sec)	5	FREQ_CONTR_WORD	I_RING<1>	FREQ_AVG	COUNT
D_OUT<11>		I_LC<4>	ZCC_CLK	Q_BPF_ADC<4>	CLK270	MF_OUT<4>	CORTEX_GPO<11>	POR<3>	LC_div_N	sensor_adc7	LF_ext_PAD (might as well)	4		I_RING<0>	FREQ_AVG	COUNT
D_OUT<12>	Cortex HCLK	Q_LC<1>	gain_code_I<3>	I_envelope<1>	ADC_B1P	MF_OUT<5>	CORTEX_GPO<12>	tau_debug<3>	LF_ext_GPIO	sensor_adc0	ring_tx_mod_clk aka CLK_2MHz aka "tx chip	3		Q_RING<3>	FREQ_AVG	COUNT
D_OUT<13>	Cortex Hard Reset	Q_LC<2>	gain_code_I<4>	I_envelope<2>	ADC_B2P	MF_OUT<6>	CORTEX_GPO<13>	tau_debug<4>	symbol_clk_ble	sensor_adc1	BLE_PDA_clk_mux_out	2		Q_RING<2>	FREQ_AVG	COUNT
D_OUT<14>	Cortex Soft Reset	Q_LC<3>	gain_code_I<5>	I_envelope<3>	ADC_B3P	MF_OUT<7>	CORTEX_GPO<14>	tau_debug<5>		sensor_adc2	ZCC from zcc_clk_rec ("ZCC") -- ZCC data o	1		Q_RING<1>	FREQ_AVG	COUNT
D_OUT<15>	SYSRESETREQ	Q_LC<4>	gain_code_I<6>	I_envelope<4>	COMPP	MF_OUT<8>	CORTEX_GPO<15>	tau_debug<6>		sensor_adc3	ring_recov_clk ("RECOVERED_CLK") -- ZC(	counter0		Q_RING<0>	FREQ_AVG 15	COUNT
													(output)		(output)	(output)

BLUE = to be added

GREEN = to be changed

Orange = change pushed to repo

INPUTS!!		Bank 0 (Default at PON)	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7	Bank 8	Bank 9	Bank 10	Bank 11	Bank 12	Bank 13	Bank 14	Bank 15	Bank 16
	D_OUT<0>	**CORTEX_GPI<0>**	(n/a)	**ADC_CLK_EXT**	**adc_reset_gpi**													
	D_OUT<1>	**CORTEX_GPI<1>**	**GPIO_PON**	**DATA_IN**	**adc_convert_gpi**													
	D_OUT<2>	**CORTEX_GPI<2>**	**AGC_IN<1>**	**DATA_CLK_IN**	**adc_pga_amplify_gpi**													
	D_OUT<3>	**CORTEX_GPI<3>**	**AGC_IN<2>**	**EXT_INTERRUPT<0>**	n/a													
					(^-- to ADC/PGA)													
	D_OUT<4>	**CORTEX_GPI<4>**	**AGC_IN<3>**	**GFSK IN <0>**	**FREQ_CTRL_WORD_SCAN (LSB)**													
	D_OUT<5>	**CORTEX_GPI<5>**	**AGC_IN<4>**	**GFSK IN <1>**	**FREQ_CTRL_WORD_SCAN**													
	D_OUT<6>	**CORTEX_GPI<6>**	**AGC_IN<5>**	**GFSK IN <2>**	**FREQ_CTRL_WORD_SCAN**													
	D_OUT<7>	**CORTEX_GPI<7>**	**AGC_IN<6>**		**FREQ_CTRL_WORD_SCAN**													
	D_OUT<8>	**CORTEX_GPI<8>**	**EXT_INTERRUPT<1>**	**I_ADC_EXT<1>**	**FREQ_CTRL_WORD_SCAN**													
	D_OUT<9>	**CORTEX_GPI<9>**	**EXT_INTERRUPT<2>**	**I_ADC_EXT<2>**	**FREQ_CTRL_WORD_SCAN**													
	D_OUT<10>	**CORTEX_GPI<10>**	**EXT_INTERRUPT<3>**	**I_ADC_EXT<3>**	**FREQ_CTRL_WORD_SCAN**													
	D_OUT<11>	**CORTEX_GPI<11>**	**dig scan_phi**	**I_ADC_EXT<4>**	**FREQ_CTRL_WORD_SCAN**													
	D_OUT<12>	**CORTEX_GPI<12>**	**dig scan_phib**	**Q_ADC_EXT<1>**	**FREQ_CTRL_WORD_SCAN**													
	D_OUT<13>	**CORTEX_GPI<13>**	**dig scan_i0o1**	**Q_ADC_EXT<2>**	**FREQ_CTRL_WORD_SCAN**													
	D_OUT<14>	**CORTEX_GPI<14>**	**dig scan_in**	**Q_ADC_EXT<3>**	**FREQ_CTRL_WORD_SCAN (MSB)**													
	D_OUT<15>	**CORTEX_GPI<15>**	**dig scan_load**	**Q_ADC_EXT<4>**	**LF_ext_GPIO**													

Interrupt Notes:

EXT_INTERRUPT<0> is debounced, and is active high

EXT_INTERRUPT<1> is not debounced, and is active high

EXT_INTERRUPT<2> is not debounced, and is active low

EXT_INTERRUPT<3> is not debounced, and is active low

external_interrupt_activelow is the dedicated pad interrupt; it is not debounced, and is active low

All five are synchronized to HCLK

Figure 6: GPIO Bank

10 Interrupts

There are four external interrupts available in the GPIO bank with various trigger polarities and debouncing. All four are synchronized to HCLK.

```
EXT_INTERRUPT<0> is debounced, and is active high
EXT_INTERRUPT<1> is not debounced, and is active high
EXT_INTERRUPT<2> is not debounced, and is active low
EXT_INTERRUPT<3> is not debounced, and is active low
```

The interrupt mask is set using the register 0xE000E100 in the startup file (cm0dsasm.s). The order of the interrupt mask is as follows, ordered from LSB to MSB. For example, to activate only the UART and RFTIMER ISRs the mask would be 0x0081.

```
UART
interrupt_gpio3_activehigh_debounced
optical_irq_in
ADC
0
0
RF_FSM
RFTIMER
interrupt_rawchips_startval
interrupt_rawchips_32
0
optical_sfd_interrupt
interrupt_gpio8_activehigh
interrupt_gpio9_activelow
interrupt_gpio10_activelow
0
```

These are useful macros for dynamically controlling interrupts. See ARM documentation for further details.

```
// Interrupt set enable reg
#define ISER *(unsigned int*)(0xE000E100)
// Interrupt clear enable reg
#define ICER *(unsigned int*)(0xE000E180)
```

```
// Interrupt clear pending reg
#define ICPR *(unsigned int*)(0xE000E280)
// Interrupt set pending reg
#define ISPR *(unsigned int*)(0xE000E200)
```

11 UART

SCM has no JTAG capability so UART is the primary debugging method. All printf output is printed out over UART. Assuming a 5 MHz HCLK frequency the default baud rate is 19200. If HCLK is some other frequency then the baud rate should be adjusted accordingly since the divide ratio is fixed (e.x. if you doubled HCLK to 10 MHz you should set baud rate to 38400). Other settings that should be used: 8 data bits, 1 stop bit, and no parity. Every character input to SCM via the UART will cause an interrupt to go off. Each character needs to be read as SCM doesn't have any UART framing capability. See the demo software for how to send commands to the chip. A three character format is adopted where three ASCII characters are sent to the chip followed by a newline (e.x. sta\n). A bank of IF statements checks to see if the input command sequence matches any of the pre-programmed ones and then executes the appropriate section of code.

A link to to a useful terminal program for Windows is given below. This program is useful for passively monitoring debug output from the chip as well as sending commands over UART. It is also possible to both send and receive UART data from Python or MATLAB. This can be a very useful way to automate data collection and testing.

`https://sites.google.com/site/terminalbpp/`

12 SPI

13 Clock Configuration

14 Frequency Counters

15 Timers

See Section 3.35 in [3] for a detailed description of the RFTIMER module. Note the design assumption that HCLK and RFTIMER are phase aligned. This means that both clocks should be derived from the same oscillator and that HCLK must be an integer multiple of RFTIMER. A moderately complex example of using the RFTIMER module can be found in the 802.15.4 TSCH demo in the Git repo. Unfortunately only 8-bit frequency dividers are available for generating the RFTIMER clock (The JIRA feature SCUM-124 for changing to a 16-bit divider was not able to be added to SCM-3C in time for tape-out).

16 LO, PA, and Divider Hardware Details

16.1 Analog Scan Chain

In this section I will attempt to describe the functionality of the chain bits. The scan chain is nominally controlled by functions from the Cortex M0, so I will not go into excessive detail regarding the exact bits in the array that need to be changed to perform certain functions. Rather, I will describe them in chunks.

- `fine_code` 5 bit control of the LC tank's fine tuning DAC f0, ... ,f4, fd
- `mid_code` 5 bit control of the LC tank's mid tuning DAC m0, ... m4, md
- `coarse_code` 5 bit control of the LC tank's coarse tuning DAC c0, ... c4, cd
- `lo_tune_select` set to 0 for LC control from the Cortex `analog_cfg`, 1 for scan
- `polyphase_enable` set to 0 to disable the polyphase filter. 1 to enable.
- `lo_current_tune` 8 bit control of LC tank's current between 180 μA and 800μA LSB to MSB.
- `test_bg` 7 bit control of the test band gap (connected to pad) from 0.75 V to 0.85 V if MSB = 0, and from 1.05 V to 1.12 V if MSB = 1 (panic).
- `pa_ldo_rdac` see `test_bg`, controls power amplifier supply.
- `lo_ldo_rdac` see `test_bg`, controls LO supply.
- `div_ldo_rdac` see `test_bg`, controls divider supply.
- `mod_logic` controls the source of modulation for the 802.15.4 capacitor. The MSB determines whether the modulation is inverted or not (1 for inversion). The next bit selects cortex or pad modulation (0 for cortex, 1 for pad). The next two bits can be used to test the modulation, setting the control bit to VDD or to ground. Note: for 802.15.4 modulation,

the bits should be set to 1000. For BLE modulation from pad using the 802.15.4 capacitor, it should be set to 0111. A schematic of this control is shown in Fig. 7.

- `mod_15_4_tune` tunes the frequency spacing of the 802.15.4 modulation capacitor. 4 bits, but the LSB is a dummy (not binary weighted).
- `sel_1mhz_2mhz` 0 uses the x2 XOR multiplier. 1 is pass throgh.
- `pre_2_backup_en` enables the static flip flop div-by-2 prescaler.
- `pre_5_backup_en` enables the static flip flop div-by-5 prescaler. This is recommended for standard operation.
- `pre_dyn` is a 3-bit, one-hot selection of three different injection-lock TSPC-esque pre-scalers. It needs to be inverted (so all 1s will disable all three dynamic pre-scalers). The second of the three is the strongest, and will result in a div-by-2 for almost all LO and divider settings. The first can consistently give a div-by-5 for most settings. The third one is the weakest, and it is possible to divide by up to 7 using it.
- `div_64mhz_enable` enables a 64 MHz output frequency divider to clock the baseband stages and receiver ADC.
- `div_20mhz_enable` enables a 20 MHz output frequency divider to clock the BLE GFSK module. This is also the clock output that is connected to the LC counter.
- `div_static_code` sets the static divider ratio. This can also be controlled from the Cortex's analog_config, and generally should be. More on that later.
- `div_static_reset_b` active low reset of the static divider.
- `dyn_div_N` there is another, theoretically lower power, divider on the chip as well. It has never been tested.
- `div_tune_select` set to 0 to have the divider controlled from the cortex. Set to 1 for the divider to be controlled from scan chain.
- `BLE_module_settings` will be obfuscated in future versions of the chip.

../figures/appendix/mod_logic_small.pdf

Figure 7: 802.15.4 Modulation Logic Schematic

16.2 Cortex Code

I will now describe some of the very low-level functions for directly controlling parts of the transmitter from the cortex. Start with the relevant config and rdata registers:

```
#define ACFG_DIV__ADDR   *(unsigned int*)(APB_ANALOG_CFG_BASE + 0x00140000)
#define ACFG_DIV__ADDR_2 *(unsigned int*)(APB_ANALOG_CFG_BASE + 0x00180000)
#define ACFG_LO__ADDR    *(unsigned int*)(APB_ANALOG_CFG_BASE + 0x001C0000)
#define ACFG_LO__ADDR_2 *(unsigned int*)(APB_ANALOG_CFG_BASE + 0x00200000)

#define ASYNC_FIFO__ADDR *(unsigned int*)(APB_ANALOG_CFG_BASE + 0x00680000)
```

`ACFG_DIV__ADDR` contains two control bits to set the RF divider divide ratio.
`ACFG_DIV__ADDR_2` has the remaining control bits, as well as enable and reset signals for the divider. Quick note here: this divider struggles at low supply voltages, and will not work for odd divide ratios if the input frequency is high (around 1.2 GHz). The code to control these divider registers is called `digProgram(div_ratio, reset, enable)`. Reset is active low. A diagram of these two registers is shown in Fig. 8. The pre-scaler must be enabled for this divider to have an output (see scan chain for details).

../figures/appendix/div_reg_small.pdf

Figure 8: Divider registers, bit by bit

`ACFG_LO__ADDR_2` has the fine frequency control LSB and fine frequency control dummy bit. `ACFG_LO__ADDR` has the remaining control bits. The function `LC_FREQCHANGE(coarse, mid, fine)` controls these overlapping capacitor DACs. This function obfuscates the dummy bits. The function `LC_monotonic(LC_code, mid_divs, coarse_divs)` implements the function in Chapter 6. If you are tuning the LC oscillator, this you will probably use `LC_monotonic`. In any case, the diagram of the two LC control registers is shown in Fig. 9. The bits labeled with a "d" are the dummy bits, and will change the frequency by approximately five LSBs of the given DAC.

`../figures/appendix/lo_reg_small.pdf`

Figure 9: Oscillator frequency tune registers, bit by bit

There are a number of functions that control the scan chain, but they are relatively straightforward, especially considering the exposition given in the previous section. There are also a functions written to generate BLE advertising packets (`gen_ble_packet`) and to use the on-chip asynchronous FIFO to transmit BLE packets (`transmit_ble_packet`). At the moment, they are designed for transmitting 128 bit packets but they can easily be repurposed to transmit larger payloads. The transmit function in particular can be repurposed with completely arbitrary data for potential experimentation with 802.15.4 chipping sequences.

16.3 Common Configurations

This section describes the necessary scan/Cortex procedure to use the radio in various modes.

16.3.1 Receive Mode (RF only)

1. Enable the LC tank LDO and current source. This can be done either through GPIO (for fast start) or with the scan chain. Enabling the LC tank can be configured to start automatically via the radio state machine on SCM v3b.

2. Set the LC tank current to an appropriate level (most people have used a current level of 127 - this appears to offer a good tradeoff between

PA efficiency, LO current, and phase noise).

3. Enable the polyphase filter.
4. Program the LC tank so that it is 2.5 MHz ABOVE the receive channel frequency.
5. Enable the IF chain and digital baseband. This is documented elsewhere.

16.3.2 Transmit Mode - 802.15.4

1. Enable the LC tank LDO and current source.
2. Enable the PA LDO. Again, on SCM v3 this can be done with either GPIO (fast) or scan chain.
3. Ensure that the polyphase filter is disabled.
4. Tune the LC tank frequency to 500 kHz ABOVE the desired channel frequency.
5. If data is transmitted from the on-chip state machine, set the four `mod_logic` bits to 1000. If data is transmitted from a pad, set `mod_logic` to 1000. The bit-bang direct modulation is inverted from the bits coming from the state machine.
6. Ensure that the transmit clock (source can be chosen) is within 40 ppm of 2 MHz.
7. Use instructions in [**Sahar˙thesis**] to load and transmit a packet.

16.3.3 Transmit Mode - BLE

1. Enable the LC tank LDO and current source.
2. Enable the PA LDO.
3. Enable the divider LDO. The divider does not need to run, BUT the data buffers run off of the divider supply.
4. Ensure that the polyphase filter is disabled.

5. Tune the LC tank frequency to approximately 250 kHz BELOW the desired channel frequency.

6. Disable the 802.15.4 DAC by setting `mod_logic` to 0010 or 0001.

7. If data is transmitted from a pad, set the `mod_logic` bits to 0010. If data is transmitted from the Cortex (requires some trickery here: the Cortex clock needs to be an exact integer multiple of 1 MHz, and bits go straight from memory mapped IO to the LC tank. This also requires assembly code. Fortunately Keil allows in-line assembly).

8. Bypass both the FIFO and the GFSK module.

9. Run the BLE transmitting assembly code.

It is also possible to use the 802.15.4 capacitor DAC to transmit BLE. However, this is only possible from off-chip. The only DAC that can be controlled directly from the Cortex is the set of 500 kHz BLE capacitors. On SCM v3b, this procedure will be significantly different for two reasons. First, there is a 2048-bit FIFO so that the Cortex does not need to run at an exact multiple of 1 MHz. Second, there is a multiplexer to select whether the data goes to the 802.15.4 capacitors or the BLE capacitors.

17 RF Receiver Analog Scan Chain

The portion of the analog scan chain (ASC) that connects to the receiver analog circuits (some of the ASC also connects to digital baseband) are largely unchanged between all hardware generations. The exception being the clock generation which has an addition bit called 'high_range' which extends the upper frequency range of the oscillator. All of these settings are already initialized to appropriate values in software but this section is included to document the control settings.

17.1 Mixer Bias

Mixer DC bias is set by driving a programmable current source through a variably sized diode connected load. The LO is then AC coupled to the mixer gates. There are four total mixer switches, a differential set each for I and Q labeled as Ip, In, Qp, and Qn.

The diode connected load is labeled as ndac and consists of four equally sized devices connected in parallel. The four bit scan code is thermometer and each bit turns on an individual diode connected nmos. Setting all bits to '1' generates the lowest bias voltage.

```
ASC<294:297> = mix_bias_In_ndac<4:1>
ASC<299:302> = mix_bias_Ip_ndac<4:1>
ASC<303:306> = mix_bias_Qn_ndac<4:1>
ASC<308:311> = mix_bias_Qp_ndac<4:1>
```

The current sources are labeled as pdac and consist of 16 equally sized current sources in parallel. The four bit pdac code is binary weighted and a '0' activates the current source. ie all zeros is the most current and thus the highest bias voltage.

```
ASC<312:315> = mix_bias_In_pdac<4:1> (<4(MSB):1>)
ASC<316:319> = mix_bias_Ip_pdac<4:1>
ASC<320:323> = mix_bias_Qn_pdac<4:1>
ASC<324:327> = mix_bias_Qp_pdac<4:1>
```

The I/Q mixers can be independently disabled by these two bits. When disabled the mixer DNW goes high impedance in an attempt to improve PA efficiency so the mixers should be disabled when transmitting. '0' is enabled and ’1’ is disabled. Note that 3C added a memory map control option for enabling and disabling the mixers which is discussed in Section 18.3.

```
ASC<298> = mix_off_i
ASC<307> = mix_off_q
```

17.2 Transimpedance Amplifiers

Each transimpedance amplifier has enable, bandwidth, and gain control bits. Each TIA has three separate enable bits to allow for flexibility in shutdown modes. The amplifiers are inverter-based and have separate enable bits for the nmos and pmos labeled as ’enn’ and ’enp’ respectively. There is another control bit which activates a pull-down switch to ground the TIA inputs. This pulls the bottom plate of the AC coupling caps to ground and makes the mixer essentially operate in voltage mode driving a capacitive load. Bandwidth control is binary weighted with all ’1’s setting the bandwidth to its minimum value.

```
ASC<328:331> = tia_cap_on<4:1> (<4(MSB):1>)
ASC<332> = tia_pulldown (1=pull TIA input to ground)
ASC<333> = tia_enn (1=enabled)
ASC<334> = tia_enp (0=enabled)

ASC<462> = tia_enp (0=enabled)
ASC<463> = tia_enn (1=enabled)
ASC<464> = tia_pulldown (1=pull TIA input to ground)
ASC<465:468> = tia_cap_on<1:4> (<1:4(MSB)>)
```

17.3 Gain Control

```
Determines whether gain is controlled by scan chain or AGC:
’1’ = AGC control, ’0’ = ASC
ASC<271> = agc_gain_mode (Q chan)
ASC<491> = agc_gain_mode (I chan)

Gain control code, d63 is max gain:
As value is decreased from d63, first the TIA gain is decreased,
followed by stage 1 gm, then stage 2 gm.
```

```
ASC<272:277> = code_scan<5:0> Q chan (<5(MSB):0>)
ASC<485:490> = code_scan<0:5> I chan (<0:5(MSB)>)

Gm control for filter stage 3 (ADC driver); thermometer coded.
All '1's = max gm.
ASC<278:290> = stg3_gm_tune<1:13>
ASC<472:484> = stg3_gm_tune<13:1>
```

17.4 Filters

Each filter stage has control over the amplifier enable, the clock enable, and tuning of the capacitor ratio that sets the filter's pole location. The transconductance of each stage is controlled as described in the gain control section.

```
Q Chan amplifier enables:
ASC<347> = stg3_amp_en
ASC<351> = stg2_amp_en
ASC<355> = stg1_amp_en

Q Chan individual clock enables:
ASC<369> = stg3_clk_en
ASC<376> = stg2_clk_en
ASC<383> = stg1_clk_en

Q Chan stage 3 cap ratio:
ASC<370:372> = stg3_C2<3:1>
ASC<373:375> = stg3_C1<3:1>

Q Chan stage 2 cap ratio:
ASC<377:379> = stg2_C2<3:1>
ASC<380:382> = stg2_C1<3:1>

Q Chan stage 1 cap ratio:
ASC<384:386> = stg1_C2<3:1>
ASC<387:389> = stg1_C1<3:1>

I Chan amplifier enables:
```

```
ASC<441> = stg1_amp_en
ASC<445> = stg2_amp_en
ASC<449> = stg3_amp_en

I Chan individual clock enables:
ASC<400> = stg3_clk_en
ASC<407> = stg2_clk_en
ASC<414> = stg1_clk_en

I Chan stage 3 cap ratio:
ASC<401:403> = stg3_C2<3:1>
ASC<404:406> = stg3_C1<3:1>

I Chan stage 2 cap ratio:
ASC<408:410> = stg2_C2<3:1>
ASC<411:413> = stg2_C1<3:1>

I Chan stage 1 cap ratio:
ASC<415:417> = stg1_C2<3:1>
ASC<418:420> = stg1_C1<3:1>
```

17.5 ADC

```
Q Chan comparator offset trim:
Binary weighted, increase from 0 to add cap to either side of comparator.
ASC<335:339> = pctrcl<4:0> (<4(MSB):0>)
ASC<340:344> = nctrcl<4:0>

Activate 1-bit mode for ZCC on Q channel:
ASC<345> = mode_1bit

Enable Q channel comparator:
ASC<346> = adc_comp_en

Enable Q channel ADC FSM:
ASC<367> = adc_fsm_en
```

```
Q channel common mode reference generation:
Separate amp/clock enables;
The select signal controls a capacitive voltage divider.
ASC<359> = vcm_amp_en
ASC<360:361> = vcm_vdiv_sel<0:1>
ASC<362> = vcm_clk_en

Q channel reference voltage generation:
Separate amp/clock enables;
The select signal controls a capacitive voltage divider.
ASC<363> = vref_clk_en
ASC<364:365> = vref_vdiv_sel<1:0>
ASC<366> = vref_amp_en



I channel common mode reference generation:
Separate amp/clock enables;
The select signal controls a capacitive voltage divider.
ASC<390> = vcm_amp_en
ASC<391:392> = vcm_vdiv_sel<0:1>
ASC<393> = vcm_clk_en

I channel reference voltage generation:
Separate amp/clock enables;
The select signal controls a capacitive voltage divider.
ASC<394> = vref_clk_en
ASC<395:396> = vref_vdiv_sel<1:0>
ASC<397> = vref_amp_en

Enable I channel ADC FSM:
ASC<398> = adc_fsm_en

Enable I channel comparator:
ASC<450> = adc_comp_en

Activate 1-bit mode for ZCC on I channel:
```

```
ASC<451> = mode_1bit

I Chan comparator offset trim:
Binary weighted, increase from 0 to add cap to either side of comparator.
ASC<452:456> = nctrl<0:4> (<0:4(MSB)>)
ASC<457:461> = pctrl<0:4>
```

17.6 Clock Generation

```
Enables for filter and ADC clock generation; '1' = on
ASC<422> = adc_phi_en
ASC<423> = filt_phi_en

Mux select for IF clock source:
00=gnd, 01=internal RC, 10=divided LC, 11=external pad
ASC<424:425> = clk_select<1:0>

IF RC clock enable; '1' = on
ASC<426> = RC_clk_en

Coarse and fine frequency tune, binary weighted
ASC<427:431> = RC_coarse<4:0> (<4(MSB):0>)
ASC<433:437> = RC_fine<4:0>   (<4(MSB):0>)

Switch between high and low speed ranges for IF RC:
'1' = high range
ASC<726> = RC_high_speed_mode
```

17.7 LDO and Reset

```
Resistor for setting bandgap reference voltage
ASC<492:498> = if_ldo_rdac<0:6> (<0:6(MSB)>)

'1' = disable ability for POR to reset IF blocks
ASC<499> = por_disable
```

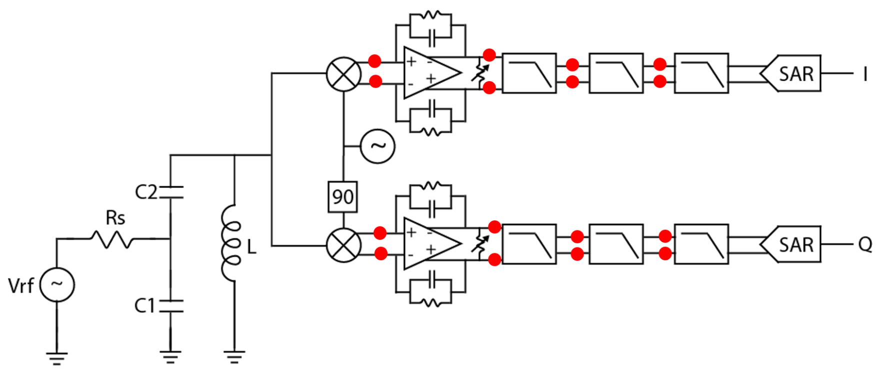


Figure 10: IF Debugging Path

```
'0' = force reset from scan chain
ASC<500> = scan_reset
```

17.8 Debug Path

IF debugging access is available at the nodes marked with dots in Figure 10. Differential IF signals are connected to source followers to pseudo-differentially route signals out to an off-chip instrumentation amplifier for viewing. The four possible observation locations all share a pair of chip pads and thus should only be turned on one at a time. Input stimulus can also be injected at these same locations from off-chip. A differential pair of input pads is driven from an off-chip single ended to differential converter. The I and Q paths have independent debug paths and thus there are 8 total pads used for debug observation (2 inputs and 2 outputs for I, 2 inputs and 2 outputs for Q).

The debug path is controlled by an enable bit to activate the source follower bias, a bit to enable an analog pass gate connecting the source follower output to the pads, and a bit which enables another analog pass gate connecting the input pads to a particular node. Thus each of the 8 total debug access points are controlled by 3 bits each for a total of 24 scan bits.

```
Q channel mixer output node:
ASC<291> = dbg_bias_en_Q
ASC<292> = dbg_out_en_Q
ASC<293> = dbg_input_en_Q
```

```
Q channel input to stage 3 filter:
ASC<348> = dbg_out_on_stg3
ASC<349> = dbg_input_on_stg3
ASC<350> = dbg_bias_en_stg3

Q channel input to stage 2 filter:
ASC<352> = dbg_out_on_stg2
ASC<353> = dbg_input_on_stg2
ASC<354> = dbg_bias_en_stg2

Q channel input to stage 1 filter:
ASC<356> = dbg_out_on_stg1
ASC<357> = dbg_input_on_stg1
ASC<358> = dbg_bias_en_stg1

I channel input to stage 1 filter:
ASC<438> = dbg_bias_en_stg1
ASC<439> = dbg_input_on_stg1
ASC<440> = dbg_out_on_stg1

I channel input to stage 2 filter:
ASC<442> = dbg_bias_en_stg2
ASC<443> = dbg_input_on_stg2
ASC<444> = dbg_out_on_stg2

I channel input to stage 3 filter:
ASC<469> = dbg_input_on_I
ASC<470> = dbg_out_on_I
ASC<471> = dbg_bias_en_I

I channel input to stage 3 filter:
ASC<446> = dbg_bias_en_stg3
ASC<447> = dbg_input_on_stg3
ASC<448> = dbg_out_on_stg3
```

There are also digital output signals which can be enabled for the ADCs and clock generation. The ADC debug outputs allow observation of internal ADC FSM signals. The clock debug signals enable the output of the multiphase clock signals used to run the filters and ADC. All signals are output through GPIO.

```
ASC<368> = Enable I channel ADC debug outputs
ASC<399> = Enable Q channel ADC debug outputs

ASC<421> = Enable debug output of ADC clock phases
ASC<432> = Enable debug output of filter clock phases
```

18 Power On Control

Several LDO control methods are implemented to enable flexibility in quickly turning the transceiver on and off for tightly duty cycled operation. There are four separate LDOs in the transceiver: the LO, PA, divider, and IF. Each can be independently controlled through any combination of the means outlined below. The possible ways to turn on an LDO are: scan chain, GP input, hardware FSM control, and through memory mapped registers from the Cortex-M0. Note that there is a fairly significant difference in the implementation of this power control logic between SCM3 and SCM3B/C. The memory mapped control option is only available in 3C.

At the time of this writing, the most practical control method appears to be using the memory mapped control option. While this does require the Cortex-M0 to be involved, it only needs to respond to an interrupt long enough to toggle one memory mapped register and then exit. This is primarily due to the turn-on time requirements of the LO. At the time when the radio FSM was designed, no consideration had been given to how long it would take for LO start-up to occur, so there is no handling of that in the FSM. Thus the FSM based control signals only assert at the actual packet event times, not offset by a start-up period.

The four ASC control signals for each LDO:

```
master_ldo_en = '0' will force the LDO off, regardless of other settings
scan_pon = '1' turns on the LDO via scan chain
gpio_pon_en = '1' allows the LDO to be turned on via GPIO_PON input
fsm_pon_en = '1' allows the LDO to be turned on via PON_XX signal
```

Verilog implementation of the power on logic:

```
assign x1 = gpio_pon && gpio_pon_en_xx;
assign x2 = fsm_pon && fsm_pon_en_xx;
assign x3 = x1 || x2 || scan_pon_xx;
assign LDO_enable_xx = x3 && master_ldo_en_xx;
```

Power-on signals generated by radio FSM:

```
assign FSM_TX_PON = (tx_state == TX_FIFO_DRAIN) || (tx_state == TX_DONE);
assign FSM_RX_PON = (rx_state != RX_SLEEP) & (rx_state != RX_DONE) & ...
    (rx_state != RX_CRC_CHECK) & ...
    (rx_state != RX_DMA_WAIT2);
```

Individual LDO 'PON_XX' enable signals from digital. These can be generated either directly from the radio FSM, or from memory mapped registers. The mux control signals are also set via memory mapped registers.

```
assign PON_LO = pon_cfg[0] ? (FSM_TX_PON || FSM_RX_PON) : pon_cfg[3];
assign PON_IF = pon_cfg[1] ? FSM_RX_PON : pon_cfg[4];
assign PON_PA = pon_cfg[2] ? FSM_TX_PON : pon_cfg[5];
assign PON_DIV = pon_cfg[6];
```

Memory mapped control signals for LDOs. The mux selects determine whether the hardware FSMs or a memory mapped register get connected to the LDO 'PON_XX' signals that go to analog. For the mux selects, '1' = FSM control, '0' = memory mapped control. The memory mapped address for this register is ANALOG_CFG_REG__10.

```
analog_cfg[160] = pon_cfg[0] = LO LDO mux select
analog_cfg[161] = pon_cfg[1] = IF LDO mux select
analog_cfg[162] = pon_cfg[2] = PA LDO mux select
analog_cfg[163] = pon_cfg[3] = LO LDO memory-mapped enable
analog_cfg[164] = pon_cfg[4] = IF LDO memory-mapped enable
analog_cfg[165] = pon_cfg[5] = PA LDO memory-mapped enable
analog_cfg[166] = pon_cfg[6] = Divider LDO memory-mapped enable
analog_cfg[167] = Aux LDO memory-mapped enable
```

The analog scan chain bits used to configure the power control block:

```
ASC<501> = scan_pon_if
ASC<502> = scan_pon_lo
ASC<503> = scan_pon_pa
```

```
ASC<504> = gpio_pon_en_if
ASC<505> = fsm_pon_en_if
ASC<506> = gpio_pon_en_lo
ASC<507> = fsm_pon_en_lo
ASC<508> = gpio_pon_en_pa
ASC<509> = fsm_pon_en_pa
ASC<510> = master_ldo_en_if
ASC<511> = master_ldo_en_lo
ASC<512> = master_ldo_en_pa
ASC<513> = scan_pon_div
ASC<514> = gpio_pon_en_div
ASC<515> = fsm_pon_en_div
ASC<516> = master_ldo_en_div
```

18.1 Aux Digital LDO

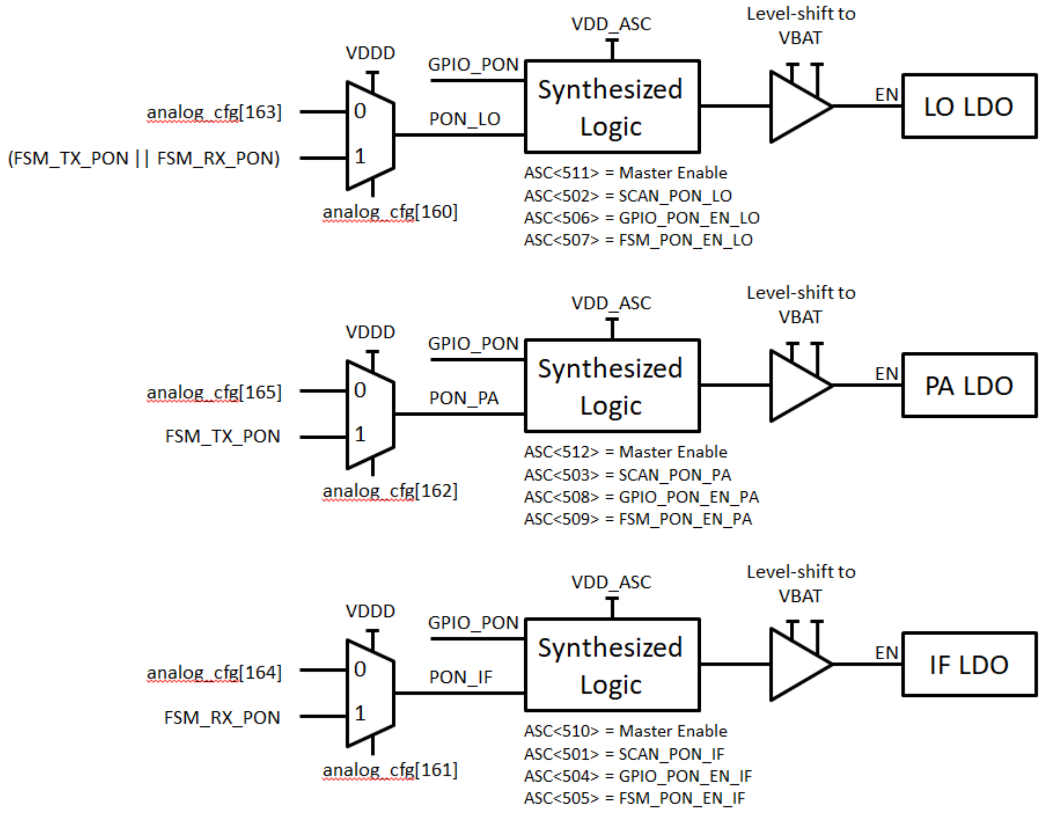
The aux digital LDO can also be turned in multiple ways. The LDO enable is multiplexed between analog scan chain and a memory mapped register as shown below. Note that per JIRA-101 an inversion was added here so that the aux digital LDO defaults to enabled at cold start. Thus the enable for the AUX LDO is active low.

```
assign aux_ldo_enable_needs_levelshifted = ...
    ~(ASC_aux_ldo_enable_mux_select? ...
    analog_cfg[167] : ASC_aux_ldo_enable);

The mux select is ASC<914>,
0 = controlled by ASC<916>,
1 = controlled by analog_cfg<167>
ASC<914> = Mux select
ASC<916> = Aux digital LDO enable
```

18.2 Power Sequencing

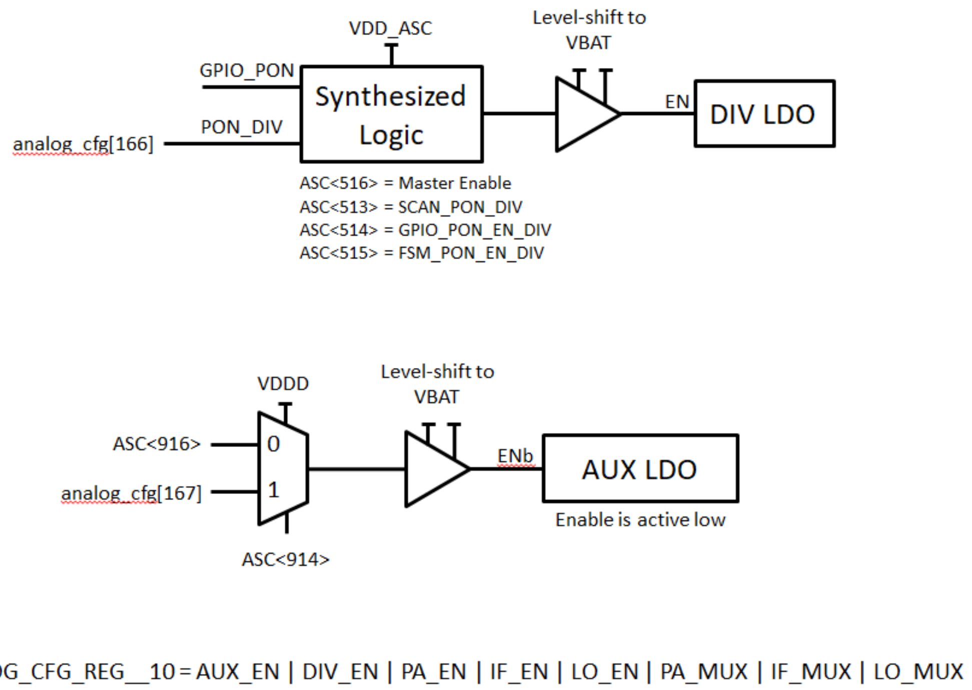
The LO takes $> 50\mu s$ for its frequency to settle. Turning on the PA also causes a frequency transient, but of much smaller magnitude. Ideally in transmit the LO should be turned on at least 50 μs before the PA is activated



802.15.4 Radio FSM

```
FSM_TX_PON = (tx_state == TX_FIFO_DRAIN) || (tx_state == TX_DONE);
FSM_RX_PON = (rx_state != RX_SLEEP) & (rx_state != RX_DONE) & (rx_state != RX_CRC_CHECK) & (rx_state != RX_DMA_WAIT2);
```

Figure 11: Diagram of power control options for radio LDOs.



ANALOG_CFG_REG__10 = AUX_EN | DIV_EN | PA_EN | IF_EN | LO_EN | PA_MUX | IF_MUX | LO_MUX

Figure 12: Diagram of power control options for divider and VDD_AUX LDOs.

to avoid broadcasting the LO settling characteristics. The PA also settles within about 50 μs. In receive, the LO should be given enough time to settle before the start of the guard time. The IF/RX LDO can be enabled at the same time as the LO. The AUX LDO must also be enabled during both transmit and receive mode.

18.3 TX/RX Switching

In addition to needing to turn the various LDOs on and off to switch between transmit and receive, the polyphase filter and mixer also need to be disabled/enabled. These can be controlled either via analog scan chain or memory mapped registers. It is recommended to use the memory mapped option to avoid the slow program time of the scan chain for fast TX/RX switching. Note that the memory mapped option is a new addition for SCM-3C. There are three bits which control whether the polyphase filter is activated and whether the mixers are switched to high impedance mode to improve the TX efficiency (there is one bit each for I and Q mixer). The memory mapped address for the analog_cfg bits is 0x52400000 (which should have the macro name ANALOG_CFG_REG__16). To use the memory mapped interface, first set all mux select bits (ASC$< 744 : 746 >$) to '1'.

```
Enable polyphase (In RX set to '1', in TX set to '0'):
Mux select: ASC<746>
S0: ASC<971>
S1: analog_cfg<256>

Enable I mixer (In RX set to '0', in TX set to '1'):
Mux select: ASC<744>
S0: ASC<298>
S1: analog_cfg<258>

Enable Q mixer (In RX set to '0', in TX set to '1'):
Mux select: ASC<745>
S0: ASC<307>
S1: analog_cfg<257>

To activate RX mode:
ANALOG_CFG_REG__16 = 0x1;
```

```
To activate TX mode:
ANALOG_CFG_REG__16 = 0x6;
```

19 Digital Baseband

Various debugging signals are available for output via the GPIO bank. Reset can come from either analog scan chain or a memory mapped register and is active low.

```
ASC<240> = mux select for reset source; 0 = ASC[241], 1 = analog_cfg[75]
ASC<241> = active low reset from ASC

Choose whether I/Q input signals are from on-chip or off-chip.
'0' = On-chip
'1' = Inject from GPIO bank
ASC<96> = IQ_select

Choose where IF amplifier gain control is connected
00=AGC FSM, 10 or 01=analog_cfg, 11=GPIN
ASC<102:101> = vga_gain_select

Gain control settings from memory mapped register (analog_cfg)
I channel gain = analog_cfg_agc[229:224]
Q channel gain = analog_cfg_agc[235:230]

Choice of detector for AGC
'0' = Use envelope detector, '1' = Use original overflow detector
ASC<100> = sel_envelope_0

For adding a gain offset to correct for I/Q amplitude mismatch
gain_offset = analog_cfg_agc[238:236]

Choose which channel to add gain offset to
'0' = subtract 'gain_offset' from Q channel
'1' = subtract 'gain_offset' from I channel
```

```
gain_imbalance_select = analog_cfg_agc[239]

Memory mapped register that controls how long AGC FSM waits for settling
AGC_wait_time = analog_cfg_agc[251:240]

Signal envelope level that triggers a gain reduction
Envelope_threshold = analog_cfg_agc[255:252]

Activate AGC mode where only the TIA is under automatic control
ASC<97> = agc_TIA_mode

Register to read the min/max values of I and Q
IQ_minmax_rdata = analog_rdata[239:224]

CDR: Sampling point for clock recovery; use 3
sample_point = analog_cfg[78:76]

CDR: Feedback parameters for clock recovery
e_k_shift = analog_cfg[58:55]  Use x
tau_shift = analog_cfg[63:59]  Use x

Sign control bit for matched filter
'0' = , '1' =
ASC<103> = mf_data_sign

Memory mapped register for reading average frequency value
freq_result = analog_rdata[265:256]
freq_valid  = analog_rdata[266]

// Check valid flag
if(ANALOG_CFG_REG__16 & 0x400)
return ANALOG_CFG_REG__16 & 0x3FF;
else
return 0;
```

19.1 RSSI

An estimate of received signal quality can be obtained by reading the Link Quality Indicator (LQI) and RSSI registers. The RSSI value corresponds to the gain setting after Automatic Gain Control has settled and has a maximum value of 63, which roughly corresponds to an input power of $\leq$ -85 dBm. For every unit value below 63, the received signal amplitude has increased by approximately 1 dB.

```
RSSI = analog_rdata[255:240] = (ANALOG_CFG_REG__15 & 0x0F)
```

19.2 Link Quality Indicator

The LQI register contains the total number of chip errors found within the first eight payload symbols. Dividing the LQI register value by 256 (8*32) gives the corresponding chip error rate. Note that if they are less than eight payload symbols, then the error rate should be calculated by dividing by the appropriate number of total chips.

```
LQI = analog_rdata[343:336] = (ANALOG_CFG_REG__21 & 0xFF)
```

19.3 Chip Rate Error Estimate

During zero-drift operation the clock and data recovery module shifts in eight new samples and outputs a strobe to the demodulator to make a decision. To correct for chip rate error it periodically clocks in either more or less than eight samples before a decision gets made. By keeping track of how many extra samples are added or dropped over the course of the packet, an estimate of the difference between the TX and RX chip rates can be obtained.

A tau value of zero indicates there is no rate mismatch between the TX and RX chip clocks. The cdr_tau_value corresponds to the number of samples that were added or dropped by the CDR (each sample point is 1/16MHz = 62.5 ns). For a fixed chip rate error, this value will vary depending on the length of the packet so need to convert this to a ppm error for making decisions. The RX chip clock has adjustment steps of about 2000 ppm so corrections should be made when this error exceeds about 1000 ppm. Note that the CDR tau value is a signed number. The ppm error can be calculated and then further simplified as shown below.

$$\text{error\_ppm} = \frac{10^6 \times (\text{cdr\_tau\_value}) \times 62.5\text{ns}}{(\text{packet\_length\_bytes}) \times 64\frac{\text{chips}}{\text{byte}} \times 500\frac{\text{ns}}{\text{chip}}} \tag{1}$$

```
cdr_tau_value = analog_rdata[415:400] = ANALOG_CFG_REG__25
chip_rate_error_ppm = (cdr_tau_value * 15625) / (packet_len * 8);
```

20 Sensor ADC

20.1 Hardware

This subsystem has several mutually exclusive purposes which can be chosen by modifying analog scan chain settings:

- Interface to an external sensor whose output is an analog voltage
- Battery voltage monitor
- Temperature sensor

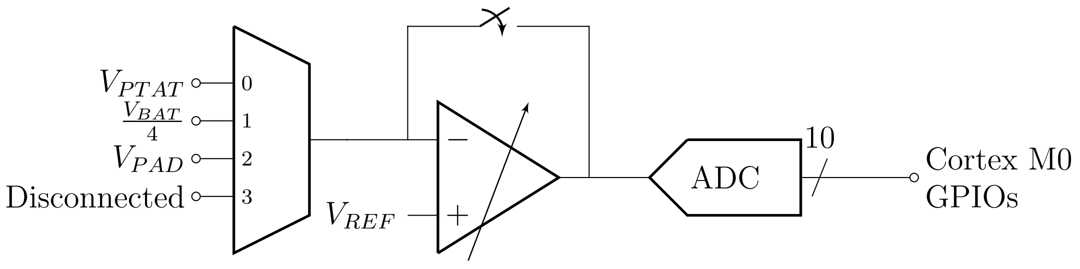


Figure 13: A pared-down block diagram of the subsystem. The key components are (1) the mux which passes the appropriate input to the sensor system (2) the programmable gain amplifier (PGA)—bypassable—which can be used to amplify the analog signal by a user-defined amount (3) the analog-to-digital converter (ADC) which digitizes the analog input and can pass that information to the on-chip microprocessor or user-facing GPIOs.

Relevant code has been provided at `https://github.com/PisterLab/scum-test-code/tree/master/scm_v3c/sensor_adc`. Information on use can be found in subsequent subsections.

Relevant Subsystem Scan Configuration Bits		
Name	# Bits	Description
sel_reset	1	Chooses the source for the ADC reset signal. 0 sources from the taped-out FSM in digital, and 1 sources from the GPI for 'adc_reset_gpi' (Figure 6).
sel_convert	1	Chooses the source for the the-ADC-needs-to-convert signal. 0 sources from the taped-out FSM in digital, and 1 sources from the GPI for 'adc_convert_gpi'.
sel_pga_amplify	1	Chooses the source for the signal which tells the PGA to amplify. 0 sources from the taped-out FSM in digiatl, and 1 sources from the GPI for 'adc_pga_amplify_gpi'.
pga_gain	8	Controls the gain of the programmable gain amplifier. Going from MSB $\rightarrow$ LSB, the gain of the PGA is (the value of the binary code)+1, e.g. 0000_0000 equates to a gain of 1 for the PGA, 0000_0001 a gain of 2, etc.
adc_settle	8	Controls the settling time allowed the ADC for each conversion. Going from MSB $\rightarrow$ LSB, the settling time decreases from as the code increases. 0000_0000 will give the ADC roughly 1.72µs to retrieve all 10 bits, and 1111_1111 will give the ADC roughly 0.35µs to retrieve all 10 bits.
bgr_tune	7	Controls the reference voltage to the LDO. The LSB is the "panic bit" which sets the reference voltage to the highest possible value ($\approx$ 1.2V). Otherwise, increasing values (MSB $\rightarrow$ LSB) leads to a decreasing output voltage with the lowest possible output value $\approx$ 0.8V
constgm_tune	8	Controls the reference current used in the ADC's comparator and the PGA's amplifier. MSB $\rightarrow$ LSB, increasing values leads to a decreasing reference current.
vbatDiv4_en	1	1 enables the V_{BAT} divide-by-4 input to the multiplexer, otherwise it's disabled. Note that this *does not choose* the divide-by-4 as the output of the multiplexer.
ldo_en	1	1 enables the on-chip LDO (Section 20.1.1), otherwise it's disabled.
input_mux_sel	2	The selection bits for the multiplexer shown in Figure 13.
pga_byp	1	Controls both the PGA bypass and the PGA enable; 1 means the PGA is disconnected from the signal path, disabled, and bypassed. 0 means the PGA is enabled and included in the signal path.
$V_{DD,\text{always on}}$	7	Controls the voltage reference for the LDO providing $V_{DD,\text{always on}}$. This particular LDO controls the enable signal for the subsystem's regulator, so having its value set too low could lead to issues with disabling and enabling this subsystem's LDO. Going from 0 to 127, increasing the tuning value leads to a decreasing output voltage with a min of $\approx$ 0.8V and a maximum of $\approx$ 1.2V. This is a known issue when the output of this LDO is set to its minimum value. This has been verified functional with the subsystem when set to the maximum voltage.
V_{DDD}	7	Controls the voltage reference for the LDO for a large portion of digital, including the Cortex M0. If the LDO voltage is set to too low or high a value, timing violations may occur within digital, leading to problems with anything which requires the Cortex M0. Going from 0 to 127, increasing the tuning value leads to a decreasing output voltage with a min of $\approx$ 0.8V and a maximum of $\approx$ 1.2V. In testing so far, leaving this value at the default midpoint setting produces no noticeable issues. Increased voltages can cause baud rate mismatch when communicating via UART.

Table 3: Subsystem scan bits which can affect functionality and performance. The bottom portion includes auxiliary scan settings which aren't directly applicable but are still potentially important (read: they can cause problems if set to undesirable values).

20.1.1 LDO

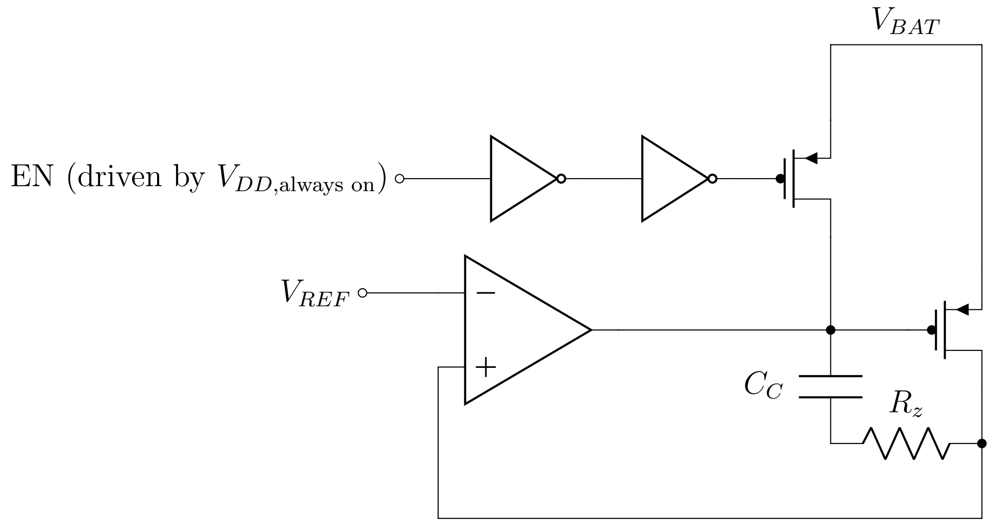


Figure 14: Schematic of the low drop-out regulator. All active blocks are driven by V_{BAT} unless otherwise specified. V_{REF} is provided by an on-chip band gap reference circuit (not shown).

20.1.2 PTAT

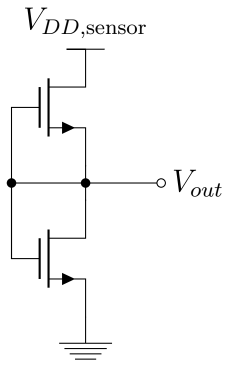


Figure 15: Schematic (without sizing) of a PTAT cell. Body connections are tied to ground.

20.1.3 PGA

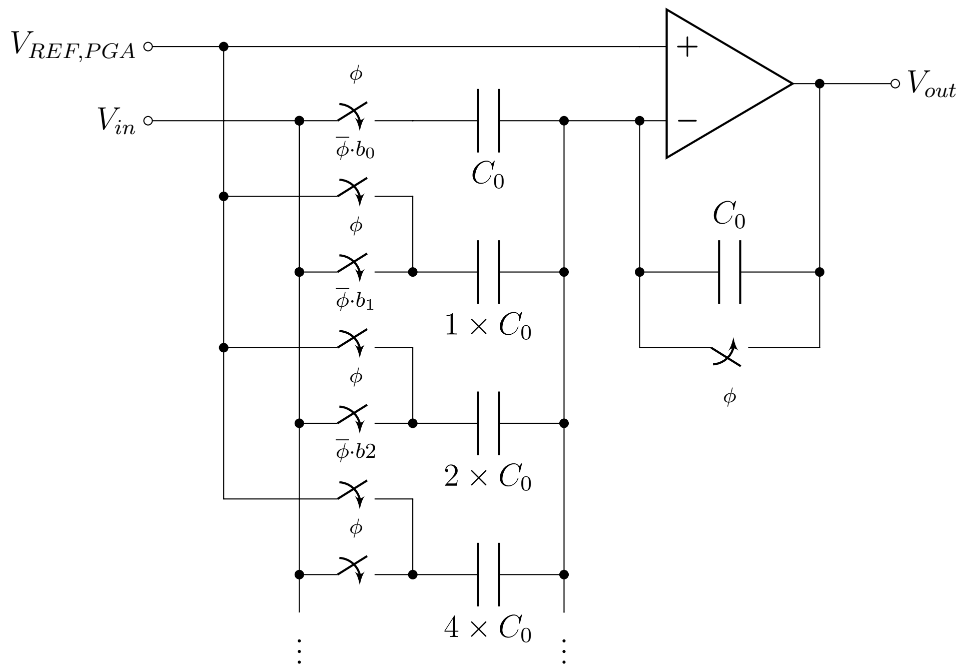


Figure 16: Schematic of the programmable gain amplifier. All active blocks are driven by $V_{DD,\text{sensor}}$ (Section 20.1.1). The reference voltage is provided by a secondary PTAT (not shown, similar to Section 20.1.2).

20.1.4 ADC

For the love of all things good, why did Cadence think it would take every block and name them the same thing?!? I know this thing was based off of Mike Scott's paper from 2003, but IEEE is down and I can't pull up the citation for it. INL, DNL, ENOB vs. frequency

20.2 Initializing Analog Scan Chain

This is done whenever you boot SCM and/or program the analog scan chain on SCM. C code has been provided in `sensor_adc/adc_config.c/scan_config_adc()` which allows you to set the values defined in the upper half of 3. Call this in `scm3C_hardware_interface.c/initialize_mote()`. An example of its use is provided below.

```
// SENSOR ADC INITIALIZATION
if (1) {
  unsigned int sel_reset      = 0;
  unsigned int sel_convert    = 0;
  unsigned int sel_pga_amplify  = 0;
  unsigned int pga_gain[8]    = {0,0,0,0, 0,0,0,0};
  unsigned int adc_settle[8]    = {1,1,1,1, 1,1,1,1};
  unsigned int bgr_tune[7]    = {0,0,0, 0,0,0,1};
  unsigned int constgm_tune[8]  = {1,1,1,1, 1,1,1,1};
  unsigned int vbatDiv4_en    = 1;
  unsigned int ldo_en       = 0;
  unsigned int input_mux_sel[2]   = {0,1};
  unsigned int pga_byp      = 0;

  // Set all GPIOs as outputs
  GPI_enables(0x0000);
  GPO_enables(0xFFFF);

  // Select banks for GPI/O
  GPI_control(0,0,0,0);
  GPO_control(9,9,9,9);

  scan_config_adc(sel_reset, sel_convert, sel_pga_amplify,
          pga_gain, adc_settle,
          bgr_tune, constgm_tune,
          vbatDiv4_en, ldo_en,
          input_mux_sel, pga_byp);
}
```

20.3 Triggering A Measurement

All of the methods described below rely on a command sent over UART, but the low-level way of triggering a reset relies on the memory-mapped register ADC_REG__START. Setting this to 0x1 primes the ADC for reading. If the on-chip FSM is used to control the ADC, one only needs to wait until the ADC-related interrupt is triggered. Otherwise, some software to control the ADC (like that in adc_test.c/loopback_control_adc_shot()) will be necessary.

Python functions have been provided in `sensor_adc/adc.py` which will be referred to throughout this subsection. Note that this is simply a user guide and assumes you aren't writing large amounts of additional software for SCM. There are many permutations of the methods described below which are possible.

20.3.1 On-Chip FSM

Before getting started, make sure

- Scan settings `sel_[reset/convert/pga_amplify]` will all need to be set to 0 so the ADC will receive its control signals from the on-chip FSM.

This uses a memory-mapped register to start the taped-out finite state machine. The simplest way of starting a conversion this is by feeding SCM the appropriate command over UART. The function `sensor_adc/adc.py/test_adc_spot()` can do this by setting `control_mode` to 'uart'.

20.3.2 GPIO Loopback

Before getting started, make sure

- Scan settings `sel_[reset/convert/pga_amplify]` should all be set to 1 so the ADC will receive its control signals via GPI.

- The function `sensor_adc/adc_config.c/loopback_control_config_adc()` should be called when initializing the scan settings for the mote. The recommend location for calling this is in function `scm3C_hardware_interface.c/initialize_mote()`. This enables the appropriate buffers on SCM to perform GPIO loopback for the control signals for the subsystem, and it sets the banks appropriately. At the moment, this will overwrite any previously-defined bank settings.

- Within `Int_Handlers.h`, there is a section which defines several variables you will need to set
  - `cycles_reset`: The number of cycles to pull the reset signal low
  - `cycles_to_start`: The number of cycles after resetting until continuing with the rest of the FSM
  - `cycles_pga`: The number of cycles given allowed for the PGA to settle

This uses software (i.e. the Cortex M0) to step the ADC through its various states rather than relying on the on-chip FSM to step the ADC. Function `sensor_adc/adc.py/test_adc_spot()` can do this by setting `control_mode` to ‘loopback’.

20.3.3 External GPI

The code for this is currently under construction. For now, please use other means of triggering a subsystem reading.

20.4 Reading the Output

Python functions have been provided in `sensor_adc/adc.py` which will be referred to throughout this subsection. Note that this is simply a user guide and describes only the baseline methods of reading the subsystem output. There are many possible permutations of what’s described below.

20.4.1 Memory-Mapped Register

Whenever the on-chip interrupt indicating that the subsystem has completed its measurement is triggered, the result of that measurement is contained in memory-mapped register `ADC_REG__DATA` (you can find this in `Memory_Map.h`). The simplest way of retrieving this value upon the interrupt’s trigger is by having SCM print the register’s value to UART. This is what the current ISR does, and function `sensor_adc/adc.py/test_adc_spot()` can read that by setting `read_mode` to ‘uart’.

20.4.2 GPO Readout

The code for this is currently under construction. For now, please use other means of reading the subsystem output.

20.5 Debugging/Known Issues

20.5.1 Supply Bounce

When using the on-chip subsystem LDO with $V_{DD,\text{always on}}$ set to a value $\geq$ 0.5-0.6V lower than V_{BAT}, the enable signal for the subsystem LDO may not be clearly defined, resulting in supply "bouncing" due to the enable vacillating between on and off.

Solution Set the voltage for $V_{DD,\text{always on}}$ within 0.3V of V_{BAT}. Lower voltages have not been verified.

20.5.2 First-Reading 511

When triggering an ADC conversion via on-chip FSM, the first reading after a power cycle is always all ones save for the MSB (see Section 20.5.3), giving a reading of 511.

No Solution Unfortunately, this seems to be due to an incorrect startup at the initial boot of the chip. More specifically, the on-chip FSM doesn't reset properly before taking the first reading.

Work In Progress It may be possible to trigger a reset before the first reading by pulling on the chip soft reset.

20.5.3 MSB "Sticking"

For voltages $\geq \frac{V_{DD,\text{sensor}}}{2}$, the MSB of the ADC output is stuck to 0.

Partial Solution When using GPIO loopback control of the ADC, triggering a soft reset after each reading (yikes) avoids this issue. This solution does not work when using the on-chip FSM to control the ADC

Full Solution When using GPIO loopback control of the ADC, increasing the ADC settling time seems to have resolved the issue with the MSB. It's confirmed that it doesn't work with the on-chip FSM because adc_done never goes low due to an architectural error where the ADC reset which is needed between each reading was tied directly to the Cortex M0's soft reset (which restarts all software).

21 Raw Bit Receive Mode

Note that this block appears to be broken on SCM3C. The shift register itself appears to be working fine (and thus you can still throw interrupts based on incoming 32-bit sequences) but the data read by the Cortex from the memory mapped register is garbage. See slides on Box for more details.

In addition to being connected to the radio FSM, recovered clock and data are connected to a 32-bit shift register with a correlator and interrupt capability. The 32-bit correlation target is set via a memory mapped register along with a programmable threshold. When the Hamming distance between the current shift register contents and the target is less than the this threshold the "interrupt_rawchips_startval" interrupt is asserted. There is also a separate interrupt called "interrupt_rawchips_32" that is asserted every time 32 new bits are clocked into the shift register so that the value can be read by software. By using the start value interrupt to search for the beginning of a packet and the 32-bit interrupt to successively receive the payload, and arbitrarily formatted data can be received.

Note that on SCM3C the direction of shifting into this register was reversed. Now new bits are shifted in at the LSB position instead of previously where they were shifted from the MSB direction. This new ordering matches the ordering from the transmitter.

```
analog_cfg[147:144] = Correlation threshold
analog_cfg[54:16] = Correlation target value

analog_rdata<287:272> = raw_chips<15:0>     offset=0x440000 (LSBs)
analog_rdata<303:288> = raw_chips<31:16>    offset=0x480000 (MSBs)
```

22 802.15.4 Radio Demo Software

22.1 Overview

The goal of the radio demo software was to validate the board support package (BSP) software development for eventual integration with OpenWSN. The software executes functionality similar to that found in a TSCH network.

22.2 CRC Check

To identify errors during optical programming, the bootlaoder Teensy calculates a CRC value on the code and inserts it into the binary at a specific memory location. The first thing the mote does after being programmed is to check the integrity of the program data by calculating its own CRC value based on instruction memory contents and compares to the value computed by the bootloader. If they match then execution continues, otherwise the program halts.

22.3 Initialize Analog Scan Chain

After confirming the integrity of its IMEM contents the mote will program the ASC. This step initializes all required settings to utilize the radio and other functionality. At this point the system clock source (HCLK) is also switched over to system_clk_sec.

22.4 Optical Calibration

The next step is to perform an initial frequency calibration using the Teensy as a reference. Since the mote needs bootloaded every time it starts up (no NVRAM in this generation), this provides an essentially free opportunity to transfer timing information to the mote. The bootloader will finish optical programming and then begin sending optical SFD sequences at a fixed 100ms rate. The timing in the teensy code was adjusted such that the SFD interrupt rate on the mote should be very close to 100ms. The mote will use these interrupts happening at a known rate to tune all its on-chip oscillators as close to their desired values as possible.

22.5 Building a Channel Table

LO tuning remains the biggest challenge for using SCM. Once communication has been established on all channels then traffic can be used to estimate frequency errors and apply corrections. First though the appropriate LO settings for all channels need to be found. There are many ways one could go about this and there are many underlying details related to monotonic tuning of the LO. What follows is one idea for extrapolating known information about one channel to all other 802.15.4 channels. These functions exist in the code base but should not be considered working. Many complications with LO tuning prevented this strategy from being fully tested. Those tuning issues need to first be resolved before attempting to build a channel table. This strategy is also complicated by the high jitter of on-chip clocks that are available for counting against and thus this strategy may never work well. This strategy may not be the best way to go about building a channel table (or it may not work at all) but is included here for reference.

After completing an initial calibration, the mote has a reasonable idea of what the correct LO code is for RX on channel 11 (at least at this current temperature). The mote however needs at least an initial guess at what LO codes it should use for TX and RX on every channel. The LO code is different for RX and TX on a given channel for a couple reasons: 1) The RX operates at an intermediate frequency of 2.5 MHz so in RX the LO needs to be set 2.5MHz below the actual 802.15.4 channel. In TX, due to the way the modulation works the LO needs to be set 500 kHz above the 802.15.4 channel (modulated bits will cause it to step 1 MHz down in frequency, thus toggling between +/- 500 kHz around the center value for the channel). 2) Switching between TX and RX requires turning off the polyphase filter (I/Q generation in RX) and turning on the power amplifier (PA) in TX. Both of these actions cause a frequency shift in the LO which is channel dependent.

Starting from the assumption that the mote knows the appropriate LO code for RX on channel 11 (however that happened, whether it was via the optical programmer calibration or by blindly searching for beacons) the mote proceeds to estimate LO codes for other channels by doing the following:

- Activate RX mode and set the LO to the known code for ch11 RX
- Turn on the LO divider and count how many ticks occur in some arbitrary time period
- Make a guess at what LO code is appropriate for the next RX channel
- Turn on the LO divider and count how many ticks occur in the same arbitrary time period
- Tune the LO code based on the ratio of the number of ticks in the new channel vs ch11 by exploiting their known frequency relationship (ie, ch11 RX = 2402.5 MHz and ch12 RX = 2407.5 MHz. Their ratio is 24075/24025 = 963/961.)

- Repeat this process for all RX channels
- Activate TX mode and set the LO to the best guess at ch11 TX
- Repeat the above process for all TX channels using ratios and comparing the TX counts to the count of ch11 RX

The mote won't have an accurate idea of what the time period that it is counting over actually is, but since it's a ratio the time period will cancel out as long as it is much greater than the variation caused by the RC clock. Note that turning the LO divider on and off also causes a frequency shift.

22.6 Acquiring Packet Rate

The mote begins listening on channel 11 and assumes that an OpenMote is sending 20B packets at a rate of 8 Hz. After SCM hears an appropriate packet on the channel it is listening to, then it will begin using its timers to attempt to turn its radio on and off in sequence with the OpenMote. If the mote hears a packet during its listening period, then it will transmit an 'ack'. The timing of this packet exchange is implemented in a manner similar to OpenWSN. The schedule is built around an expected packet arrival rate which is expressed in terms of the number of counts of the 500 kHz RFTimer (this value doesn't necessarily have to be 500 kHz but that's what it is in this demo code). The RX begins listening for the packet a guard time in advance of the expected arrival time. The RX hardware is turned on some amount of time before this to allow it to settle. A watchdog timer is used to detect if a packet never arrives within +/- a guard time of the expected arrival time. If the packet is received, then the mote prepares an 'ack' (just a packet filled with nonsense). The actual transmission occurs a turnaround time after the RX was completed. Again the TX hardware is turned on in advance of the actual communication time to allow it to settle

22.7 Frequency Management

At the end of an ack transmission (which indicates that the RX was successful otherwise we wouldnt have ack'd) several adjustments are made to correct for crystal-free issues. This demo assumes that the receiving node should always update its timing by trusting the transmitting node. These steps don't necessarily need to be done at the end of the TX ack but should be completed sometime before the next listening period.

The digital baseband provides an estimate for each packet of the error in the intermediate frequency, which should be corrected by adjusting the LO code. There will be variation in this measurement based on noise and other impairments so the mote needs to keep track of this information on a per-channel basis. It can then filter the historical information and make corrections to each channel code. It is assumed here that when a correction is applied for a RX channel that the same correction should be applied to that channel's TX code. It is not yet known if this is the best thing to do or if there is a better solution.

The mote needs to make sure its RX ADC clock is the correct frequency since the chip clock is derived from this clock source. The mote uses the information available from its CDR to calculate a ppm error relative to the transmit chip clock and will apply a correction if the error is $> 1000ppm$.

The expected packet arrival time should be updated to correct for the drift between timers on motes. This is accomplished by monitoring the error between expected and actual packet arrival time and using that information to adjust how many ticks the mote thinks is one packet interval. The mote doesn't need to update the actual frequency of its RF Timer (which will have pretty coarse adjustment steps) but if the source it is deriving HCLK from drifts too much then UART will start encountering errors.

The last clock that the mote should monitor is the TX chip clock. This can be compared against the RF timer using counters. Since the number of RF timer ticks that corresponds to the known packet rate is being continuously updated, this should give a fairly accurate source to compare the TX chip clock against.

22.8 BSP-like Radio Control

An attempt was made to implement the radio control in a manner which facilitates BSP development for OpenWSN. The relevant functions are listed below. Since the LO channel tuning is dependent on whether the mote is in TX or RX mode, the setFrequency() function must be different depending on whether TX or RX is happening first. The other functions behave as their names imply and are documented in the code.

```
void setFrequencyRX(unsigned int channel)
void setFrequencyTX(unsigned int channel)
void radio_loadPacket(unsigned int len)
void radio_txEnable()
void radio_txNow()
void radio_rxEnable()
void radio_rxNow()
void radio_rfOff()
void radio_enable_interrupts()
void disable_radio_interrupts()
void rftimer_enable_interrupts()
void rftimer_disable_interrupts()
```

22.9 Hard-Wired Radio Connection

In order to directly connect the TX/RX using wires instead of the radio for testing, a few configuration changes are needed. The GPIO input mux on the FPGA needs to be set to bank 2 to select the clock and data inputs. The multiplexers for choosing the source of clk/data going to the baseband need to be set to external GPI. The TX_CLK needs to be routed out through bank 10 of GPO4.

23 Optical Programmer Details

The 3.3V / GND pins on the transmitter board should connect to their corresponding pins on Teensy 3.6 and the DATA connection should connect to pin 24. The Teensy code should be downloaded from the repo.

References

[1] David Burnett. "Crystal-free wireless communication with relaxation oscillators and its applications". PhD thesis. PhD thesis, EECS Department, University of California, Berkeley, 2019.

[2] Filip Maksimovic. "Monolithic Wireless Transceiver Integration". PhD thesis. PhD thesis, EECS Department, University of California, Berkeley, 2018.

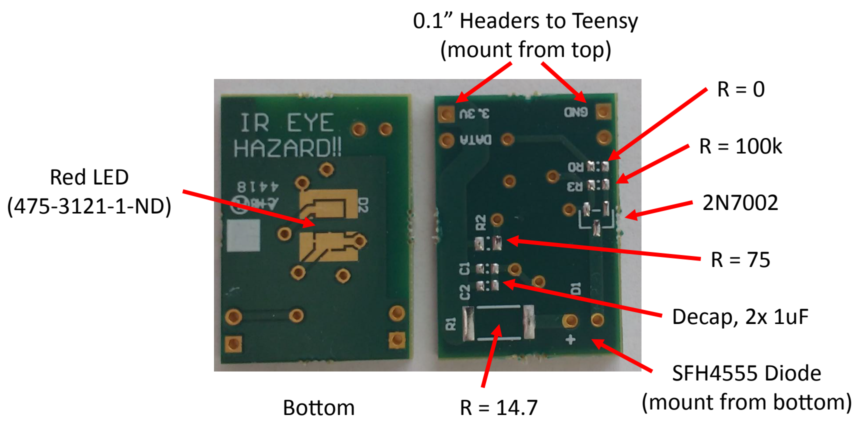


Figure 17: Programmer PCB

[3] Sahar Mesri. "Design and user guide for the single chip mote digital system". PhD thesis. Master's thesis, EECS Department, University of California, Berkeley, 2016.

[4] Bradley Wheeler. "Low Power, Crystal-Free Design for Monolithic Receivers". PhD thesis. PhD thesis, EECS Department, University of California, Berkeley, 2019.

[5] Joseph Yiu. *The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ Processors*. Academic Press, 2015.

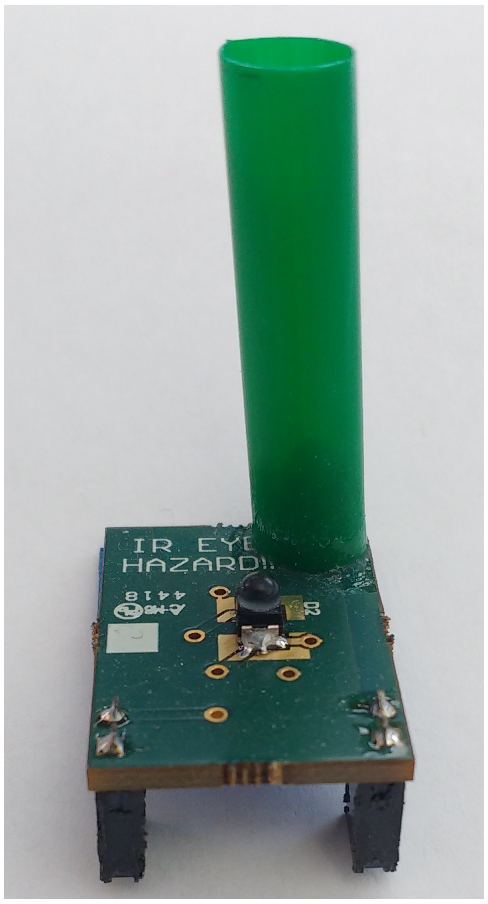


Figure 18: Assembled Programmer. A 3cm section of a plastic straw is attached to serve as a eye safety standoff and to help aim the programmer.

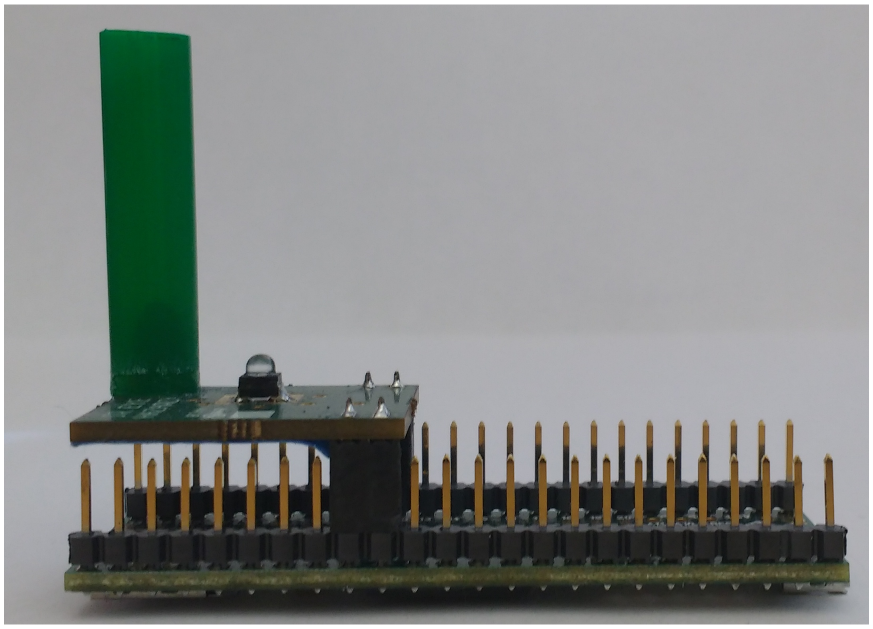


Figure 19: Programmer after attaching to Teensy 3.6 uController. It is recommended to place some tape on the programmer board between it and the uController pins to avoid incidental shorting as the headers tend to have enough play to allow contact.