TinySDR: Low-Power SDR Platform for Over-the-Air Programmable IoT Testbeds

Mehrdad Hessar, Ali Najafi, Vikram Iyer, and Shyamnath Gollakota,
University of Washington

https://www.usenix.org/conference/nsdi20/presentation/hessar

This paper is included in the Proceedings of the
17th USENIX Symposium on Networked Systems Design
and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

TinySDR: Low-Power SDR Platform for Over-the-Air Programmable IoT Testbeds

Mehrdad Hessar[†], *Ali Najafi*[†], *Vikram Iyer and Shyamnath Gollakota*
University of Washington
[†]Co-primary Student Authors

Abstract

Wireless protocol design for IoT networks is an active area of research which has seen significant interest and developments in recent years. The research community is however handicapped by the lack of a flexible, easily deployable platform for prototyping *IoT endpoints* that would allow for ground up protocol development and investigation of how such protocols perform at scale. We introduce tinySDR, the first software-defined radio platform tailored to the needs of power-constrained IoT endpoints. TinySDR provides a standalone, fully programmable low power software-defined radio solution that can be duty cycled for battery operation like a real IoT endpoint, and more importantly, can be programmed over the air to allow for large scale deployment. We present extensive evaluation of our platform showing it consumes as little as 30 uW of power in sleep mode, which is 10,000x lower than existing SDR platforms. We present two case studies by implementing LoRa and BLE beacons on the platform and achieve sensitivities of -126 dBm and -94 dBm respectively while consuming 11% and 3% of the FPGA resources. Finally, using tinySDR, we explore the research question of whether an IoT device can demodulate concurrent LoRa transmissions in real-time, within its power and computing constraints.

1 Introduction

Recent years have seen development of numerous wireless protocols for Internet of Things (IoT) devices. In addition to longtime standards such as Bluetooth and Zigbee, a number of new protocols including LoRa, Sigfox, NB-IoT and LTE-M have been developed that achieve long ranges of more than a few kilometers. Due to the lack of a de-facto standard, this space remains an active area of research for both industry and academia. The rapid advances in this space however present practical challenges for researchers: each of these protocols requires a dedicated radio chipset to evaluate, and these proprietary solutions often leave little room for protocol modification. The academic community is therefore severely handicapped by the lack of a *flexible* platform, as even a complex multi-radio prototype cannot adapt to evaluate new protocols or even customize existing solutions. The current ecosystem therefore discourages researchers from investigating the important questions that arise when scaling up IoT networks, and more importantly taking a systematic approach to developing new protocols from the ground up.

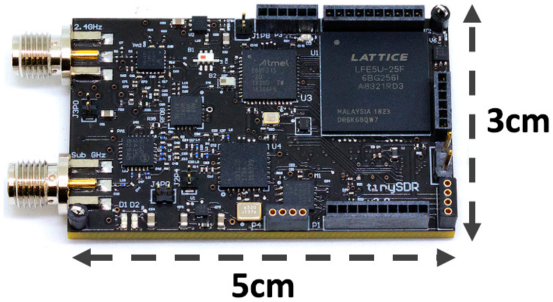


Figure 1: **TinySDR Hardware Platform.** It has two antenna ports for running IoT PHY and MAC protocols at 2.4 GHz and 900 MHz. This image is the actual size of the platform on printed paper.

Ideally, we would like a large scale IoT network testbed with the flexibility to run *any* IoT protocol at the PHY and MAC layers. Further, since many of these IoT testbeds can span hundreds of endpoints across a large campus or even a city, we need the ability to push changes to the PHY and MAC layers, using simple over-the-air software updates. This would allow for performance comparisons on a single testbed to investigate the trade-offs between existing standards as well as showcase the advantages of an entirely new custom protocol. Moreover, to make such a system representative of real-world deployments, individual network nodes should model the constraints of IoT endpoints. Specifically, these devices should have appropriate power controls and options to duty cycle transmissions, have an ultra-low power sleep mode and also have interfaces to connect sensors. Finally, the ability to run these endpoints on batteries would also allow for flexibility of deployment in spaces without dedicated power access, or even in mobile scenarios.

Realizing this vision however is challenging with existing software defined radio (SDR) platforms. Specifically, we require an SDR for the flexibility of implementing differ-

Platform	Sleep Power	Standalone	OTA	Cost	Max BW (MHz)	ADC (bits)	Frequency Spectrum (MHz)	Size (cm)
USRP E310 [7, 17]	2820 mW	✓	✗	$3000	30.72	12	70∼6000	6.8×13.3
USRP B200mini [6, 12]	N/A	✗	✗	$733	30.72	12	70∼6000	5×8.3
bladeRF 2.0 [1, 17]	717 mW	✓	✗	$720	30.72	12	47∼6000	6.3×12.7
LimeSDR Mini [2, 3, 25]	N/A	✗	✗	$159	30.72	12	10∼3500	3.1×6.9
Pluto SDR [18]	N/A	✗	✗	$149	20	12	325∼3800	7.9×11.7
μSDR [9, 10, 30]	320 mW	✓	✗	$150	40	8	2400∼2500	7×14.5
GalioT [5, 63]	350 mW	✓	✗	$60	14.4	8	0.5∼1766	2.5×7
TinySDR	**0.03 mW**	✓	✓	**$55**	**4**	**13**	**389.5∼510, 779∼1020, 2400∼2483**	**3×5**

Table 1: **Comparison Between Different SDR Platforms.** Costs are based on sale prices for commercial products without a public bill of materials (BOM) and published BOM prices for research prototypes. OTA refers to over-the-air programming capabilities.

ent PHY protocols; but there is currently no SDR platform that meets the requirements of IoT endpoints (see Table 1). Existing SDR systems consume large amounts of power for transmitting data, do not support ultra-low power sleep modes, require wired infrastructure and often a dedicated computer and furthermore, are expensive. More importantly, none of the existing SDR platforms support over-the-air programming to update PHY or MAC protocols. Finally, IoT devices prioritize power consumption and communication range and hence use limited radio bandwidth — LoRa, Sigfox, NB-IoT, LTE-M, Bluetooth and ZigBee use only 500 kHz, 200 Hz, 180 kHz, 1.4 MHz, 2 MHz and 2 MHz respectively. In contrast, existing SDR platforms focus on achieving high performance in terms of bandwidth because *they are tailored to the needs of gateway devices and not for IoT endpoint devices.*

Driven by a need for such a platform in our own research, we design tinySDR as shown in Fig. 1, the first SDR platform tailored to the needs of IoT endpoints. TinySDR provides an entirely standalone solution that incorporates a radio front-end, FPGA and microcontroller for custom processing, over-the-air FPGA and microcontroller programming capabilities, a micro SD card interface for storage, ultra-low power sleep modes and highly granular power management options to enable battery-powered operation. It is capable of transmitting and receiving in both the 900 MHz and 2.4 GHz ISM bands, supports 4 MHz of bandwidth which is sufficient for most IoT protocols including Bluetooth, Zigbee, LoRa, Sigfox, NB-IoT and LTE-M, and can achieve the high sensitivities of commercial solutions such as LoRa chips [24]. Additionally it includes multiple analog and digital I/O options for connecting sensors.

Designing such an SDR platform required addressing multiple systems, architecture, power and engineering challenges:

- **Low-power hardware architecture.** Achieving a small form-factor, low-power SDR requires a minimalist design approach that can satisfy the real-time needs of IoT protocols and ensure flexibility at the PHY and MAC layers. To do this, we exploit recent advances in small, low-power microcontrollers, FPGAs and flash memory to pick the right components for our platform (see §3.1). We use a low-power FPGA to run the PHY layer while the microcontroller runs the MAC protocols as well as handles the I/O operations between the FPGA, radio, memory and sensor interfaces (see §3.2).

- **Efficient power management.** Achieving highly granular power management needed for battery-powered operation and enabling ultra-low power sleep modes requires shutting down parts of SDR when not in use. This is important for IoT endpoints that perform duty-cycled operations and require an ultra-low power sleep mode to achieve a long battery life. This presents a design tradeoff between the complexity of toggling the power of each hardware component ON and OFF, and the cost of additional circuitry to do so. We address this challenge in §3.3 and achieve sleep power as low as 30 μW.

- **Over-the-air SDR programming.** Enabling a truly scalable system requires the ability to update the PHY and MAC layers on the platform, over-the-air, in a testbed deployment. This however also introduces the challenge of over-the-air FPGA and microcontroller programming as well as communicating these updates robustly to each device in the network while minimizing power consumption and network utilization. We use a dedicated wireless backbone subsystem complete with a MAC protocol and its own flash memory to program both the microcontroller and FPGA. Additionally we leverage compression and low-power decompression algorithms to minimize network downtime during the updates (see §3.4)

Fig. 2 shows the power consumption of the radio module in tinySDR compared to existing SDR platforms. We evaluate tinySDR's performance by presenting case studies of two common protocols: LoRa and BLE beacons, and also evaluate tinySDR in a campus-testbed of 20 devices.

- LoRa modulation and demodulation use 4% and 11% of the FPGA resources respectively and achieve a sensitivity of -126 dBm for 3.12 kbps, which is similar to an SX1276 [24] LoRa chip with the same configuration. Further, the FPGA supports real-time modulation and demodulation of all LoRa spreading factors from 6 to 12. A LoRa MAC implementation on our MCU is also compatible with the *The Things Network*.

- TinySDR supports 2.4 GHz BLE beacon transmissions. The full baseband packet generation on the FPGA uses 3% of its resources. The platform can perform frequency hopping with a delay of 220 us and achieves a sensitivity of -94 dBm which is comparable to the commercial BLE chipsets [21].

Finally, we present a case study of how the unique capabilities of tinySDR could be used to answer new research questions. Recent work has explored techniques to enable concurrent transmissions in LoRa networks [44, 47]; however these solutions were prototyped on USRPs and it is unclear if IoT endpoints can decode concurrent transmissions in real-time within their power and resource constraints. We implement a custom decoder on tinySDR to demonstrate for the first time that IoT endpoints *can* receive concurrent transmissions.

Contributions. To summarize, we design the first SDR platform tailored to the needs of IoT endpoint devices. By making careful design and architectural choices, our platform achieves low power, supports IoT protocols at both 900 MHz and 2.4 GHz and has computation resources to do on-board processing. We present a highly granular power management scheme that enables duty-cycled operation and 10,000x lower power sleep modes. We also develop the first over-the-air SDR programming capability to support PHY and MAC updates in a wireless testbed. We characterize and evaluate our platform with case studies of LoRa and BLE beacons. Finally, we present a research exploration of concurrently receiving multiple LoRa transmissions on our SDR platform.

Platform availability. TinySDR's hardware schematics and software are available at:

https://github.com/uw-x/tinysdr

2 SDR Requirements for IoT Nodes

To motivate the need for tinySDR and inform our design decisions, we begin by identifying the key requirements for an IoT endpoint. These include 1) operation in the 900 MHz and 2.4 GHz bands, 2) low power operation which requires the ability to transition to ultra-low power sleep mode, 3) standalone operation which requires an on-board control unit to duty cycle the radio, 4) over-the-air programming capabilities for large scale IoT testbeds, 5) low cost per node, and 6) at least 2 MHz bandwidth to support IoT protocols including LoRa, SIGFOX, LTE-M, NB-IoT, ZigBee and Bluetooth. While there are a number of commercially available SDRs such as the USRP, BladeRF, Pluto SDR, and LimeSDR [1, 3, 7, 31, 36] on the market and SDR research prototypes such as WARP, Argos, SORA, SODA, KUAR, Tick, μSDR, OpenMili, and GalioT [40, 41, 43, 46, 55–58, 60, 63, 64, 66–68, 70, 72, 73], all of them are designed as *gateway devices* and do not satisfy many of the above constraints. Here, we analyze the shortcomings of these platforms in the context of these requirements.

• **Low power operation and sleep mode.** Fig. 2 compares the power consumption of the radio module *alone* in existing SDR platforms, since each one has different peripherals. We find that most SDR platforms consume 200-300 mW in receive mode, but a lot more power when transmitting. While this may be acceptable for a gateway devices that are more often receiving, typical IoT endpoints do the opposite and are required to transmit data like sensor information. Moreover, real IoT nodes spend a very short time transmitting before transitioning to ultra-low power sleep modes. Although IoT radios often consume tens to hundreds of milliwatts of power, the key to achieving long battery lifetimes is exploiting their microwatt power sleep modes. Table 1 shows that none of the other platforms can benefit from duty cycling as they consume more power in sleep mode than tinySDR does when transmitting; tinySDR's microwatt power consumption in sleep mode enables dramatic power savings with duty cycling.

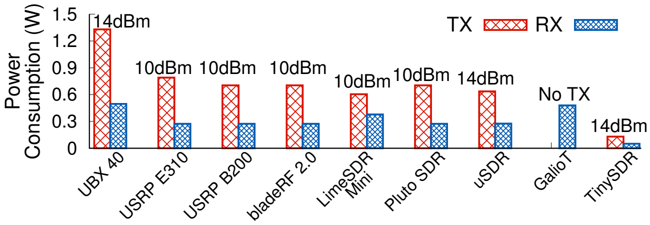


Figure 2: **Radio Module Power Consumption for Each Platform.** The TX output power of each radio module is shown on top of it.

• **Standalone operation and cost.** We observe that some of these platforms do not allow for standalone operation, i.e., they cannot be used in a testbed deployment without an external computer. Among the ones that do, the Embedded USRP and bladeRF cost $700 or more per unit making large scale deployments expensive. μSDR allows for standalone operation but only operates at 2.4 GHz and cannot support protocols like LoRa. GalioT [63] uses the low cost RTL2832U radio [5] connected to a Raspberry Pi computer which allows for standalone operation, however it does not support 2.4 GHz band. Moreover, this platform is *receiver only* and cannot be used to prototype a typical IoT node that transmits data.

• **Over-the-air (OTA) programming.** As shown in Table 1, all existing SDR platforms rely on wired interfaces for programming. This means that even if one of these systems were connected to a battery, running an experiment would require either tethering each one to a wired network or individually programming them. An OTA programming system is crucial to realizing the goal of a large scale wide area testbed as without it, researchers have to decide between limiting themselves to deployment scenarios with wired infrastructure that are not representative of real IoT use cases or traveling over *kilometer* distances to update individual nodes for *each* minor protocol modification, which would be unmanageable at scale.

3 TinySDR Platform

We first describe our design choices for the different components of our hardware shown in Fig. 3 and explain the interfaces between them. Next we present the power management module which enables our ultra-low-power sleep mode. Finally, we describe our over-the-air update protocol including decompression algorithms and over-the-air reprogramming.

3.1 Hardware Design

We seek to minimize power consumption and cost while offering the flexibility of an SDR to process raw samples.

3.1.1 Designing the Software Radio

The core block on our platform is the software-defined radio, a programmable PHY layer that processes and converts bits to radio signals and vice versa. We begin by explaining our choices for the primary components of an SDR which are a radio chip that provides an interface for sending and receiving raw samples of an RF signal as well as an FPGA that can process these signals in real time. We then discuss the supporting peripherals for these devices such as a power amplifier (PA) to boost the output of the radio chip and non-volatile memory for the FPGA to read and write data from.

Choosing a radio chip. We begin by choosing a radio chip as its specs define the requirements for the FPGA and other blocks. Our primary requirement is that the chip supports reading and writing raw complex I/Q samples of the RF signal. As shown in Table 2, current SDR systems use I/Q radio chips that are designed to cover a multi-GHz spectrum and have high ADC/DAC sampling rates to support large bandwidth. For example, the AD936x [17] series which is used in USRP and Pluto SDR can transmit up to 3.8 GHz and supports sampling rates as high as tens of MHz. Each of these specs such as wide bandwidth, low noise, and high sampling rate represent fundamental trade offs of power for performance, and therefore these chips consume watts of power. Moreover, some of these radio chips costs more than $100.

We instead take a different approach: identify the minimum required specs and find a radio that supports them. Specifically, a radio chip for an IoT platform must be able to operate in at least the 900 MHz and 2.4 GHz ISM bands, have 4 MHz of bandwidth, while otherwise minimizing power and ideally costing less than $10. We analyze all of the commercially available radio chips that provide baseband I/Q samples and list them in Table 2, where only the AT86RF215 supports all of our requirements. In addition to its lower cost and support for both frequency bands, it also consumes less power than the MAX2831 and the SX1257. Moreover, the AT86RF215 integrates all the necessary blocks including an LNA, programmable receive gain, automatic gain control (AGC) and low pass filter, ADC on the RX chain, as well as a DAC and programmable PA with a maximum power of 14 dBm on the TX side. In terms of noise, the RF front-end has a 3-5 dB noise figure which is even better than the noise figure of the front-end used in Semtech SX1276 LoRa chipset, suggesting it should be able to achieve long range performance. It consumes 5x less power than the radios used on other SDRs as shown in Fig. 2 and has built in support for common modulations such as MR-FSK, MR-OFDM, MR-O-QPSK and O-QPSK that can save FPGA resources or power by bypassing the FPGA entirely.

Picking an FPGA. Now that we have chosen a radio chip, the next step in our design process is to find an FPGA that can interface with it. Aside from minimizing power and cost, we would also like to maintain a small form factor and short wake-up time. Although flash-based FPGAs are capable of fast wake-ups, they are more expensive compared to SRAM-based FPGAs with the same number of logic elements. We use LFE5U-25F [33] FPGA from Lattice Semiconductor for baseband processing which is SRAM-based and has 24k logic units. This chip provides a greater number of look up tables (LUTs) than the FPGAs on the Pluto SDR and LimeSDR mini, and at a lower cost. Moreover, it is significantly cheaper than the flash-based FPGA used in uSDR [56].

Table 2: **Existing Off-the-Shelf I/Q Radio Modules.**

I/Q Radio	Frequency (MHz)	RX Power (mW)	Cost
AD9361 [17]	70~6000	262	$282
AD9363 [18]	325~3800	262	$123
AD9364 [12]	70~6000	262	$210
LMS7002M [25]	10~3500	378	$110
MAX2831 [10]	2400~2500	276	$9
SX1257 [35]	862~1020	54	$7.5
AT86RF215 [20]	**389.5~510** **779~1020** **2400~2483**	**50**	**$5.5**

Adding a power amplifier (PA). AT86RF215 only supports a maximum transmit power of 14 dBm which is traditionally used by IoT radios but is less than the 30 dBm maximum allowed by the FCC. To provide flexibility, we add optional PAs. Given the high cost and power requirements of wideband PAs that could operate at both 900 MHz and 2.4 GHz we instead select two different chips: the SE2435L [23] for 900 MHz and SKY66112 [28] for 2.4 GHz. Our 900 MHz PA supports up to 30 dBm output power, and the 2.4 GHz PA can output up to 27 dBm. Both chips also include an LNA for receive mode and a built in circuit to bypass either of these components for power savings. In receive mode, we can either pass the incoming signal through the LNA and then connect it to the radio or completely bypass the LNA and connect the signal directly. The maximum bypass current is 280 uA and the sleep current of both power amplifiers is only 1 uA. In transmit operation we can pass the signal through the PA and amplify the signal or turn off the PA and pass the signal directly to the antenna for transmit power $<$ 14 dBm.

Picking the microcontroller. We use a microcontroller to control all the individual chips and toggle all of these power saving options. In addition to having a low sleep current it must be able to support multiple control interfaces, have enough memory resources to support IoT MAC protocols and also be able to run a decompression algorithm for our OTA system. We select the MSP432P401R [27] a 32-Bit Cortex M4F MCU which meets all of our requirements with less than 1 uA sleep current, has 64 KB of onboard SRAM and 256 KB of onboard flash memory. In addition to controlling the I/Q and backbone radio parameters, and reprogramming of the FPGA, the MCU performs the important function of power management. It is responsible for toggling ON and OFF the power amplifiers, as well as performing power-gating by turning ON and OFF different voltage regulators in §3.3.

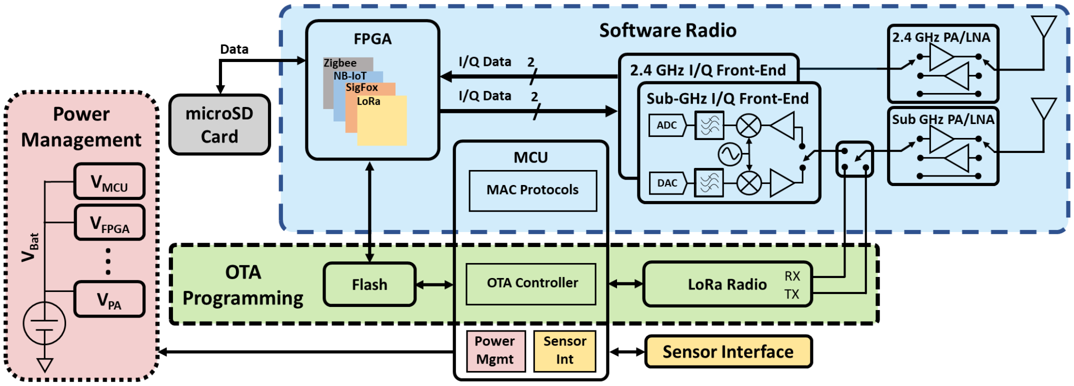


Figure 3: **TinySDR System Block Diagram.** A complete system diagram showing all of the components of tinySDR. This includes the software radio consisting of the radio, amplifiers, and FPGA, OTA programmer which uses a LoRa radio and flash memory to store programs, and a power managment system with the flexibility to turn off power consuming components. Each of these subsystems are controlled in software running on the MCU.

3.1.2 Designing OTA Update Hardware

While the above discussion enables a small, low power, low cost SDR for easy deployment, FPGAs and microcontrollers typically require a wired interface for reprogramming. There are two options for enabling wireless reprogramming: i) using the existing I/Q radio and FPGA and ii) using a dedicated wireless communication chipset. We chose the second option since it provides a fail-safe mode for updating the firmware. Moreover, a dedicated wireless communication chipset would consume less power compared to the first option.

OTA wireless chipset. A key question when designing an OTA update system is, what wireless protocol should be used? To support wide area networking, we focus on protocols designed for long range operation. We analyze all of the available long range protocols and select LoRa for our OTA system for a number of reasons. First, LoRa receivers have a high sensitivity which enables kilometer ranges. LoRa also support a wide range of data rates from 11 bps to 37 kbps which allows us to trade off rate for range depending on the deployment scenario. Moreover, LoRa is becoming more and more wide-spread in the US. We use the SX1276 Semtech chipset [24] which is available for $4.5, minimizing cost.

Flash Memory. Our FPGA is SRAM based and does not include on-chip non-volatile memory for storing programming data. We instead store the firmware bitstream on a separate flash memory chip. The FPGA programming bitstream is 579 KB and the MCU programs require a maximum of 256 KB. We chose the MX25R6435F flash chip with 8 MB memory. Although this is far more than the size required, it allows tinySDR to store multiple FPGA bitstreams and MCU programs to quickly switch between stored protocols without having to re-send the programming data over the air.

3.2 Interfacing Between Blocks

3.2.1 Reading and Writing I/Q Samples

The AT86RF215 radio chipset samples baseband signals at 4 MHz with a 13 bit resolution for both I and Q. Operating at the full rate therefore requires an interface which can support a throughput of over 100 Mbps without consuming a large amount of power to meet our design objectives. To do this we use low-voltage differential signaling (LVDS) [4] which is a high-speed digital interface that reduces power by using lower voltage signals but maintains good SNR by sending data over two differential lines to reduce common mode noise.

I_SYNC (0b10)	I_Data (13 bits)	Control (1 bit)	Q_SYNC (0b01)	Q_Data (13 bits)	Control (1 bit)

Figure 4: **I/Q Word Structure Used by I/Q Radio.**

Receiving serial I/Q data. Our system communicates over LVDS to the FPGA in serial mode to transfer I/Q data with a physical interface consisting of 4 I/O lines, pairs of which are used to send data and clock signals. The radio outputs 32-bit serial data words at 4 Mwords/s using the format in Fig. 4. Each data word starts with the *I_SYNC* pattern which indicates the start of the *I* sample which we use for synchronization. Next, it has 13 bits of *I_Data* followed by a control bit. The same format follows for *Q*, beginning with a synchronization pattern *Q_SYNC* and then 13 bits for *Q_Data* and the final control bit. The required 128 Mbps data rate is achieved using a 64 MHz clock provided by the radio operating at double data rate by sampling at both the rising and falling edges of the clock. We implement an I/Q deserializer on the FPGA to read the data which samples the input at both the rising and falling edges of the clock, uses the *I_SYNC* and *Q_SYNC* to detect the beginning of the data fields and loads the *I* and *Q* values into 13 bit registers for parallel processing.

Transmitting I/Q samples. In TX mode we need to do the opposite of the above sequence to convert from the parallel representation on the FPGA to a serialized LVDS stream. To do this, we use the FPGA's onboard PLL to generate the 64 MHz clock signal. Next to create our double data rate output signal that varies on both the positive and negative edges of this clock signal using a dual-edge D flip-flop design [48] resulting in the desired 128 Mbps data rate. We use this to generate the same I/Q word structure described above.

3.2.2 Memory Interfaces

After reading the raw data from the LVDS lines using the I/Q deserializer described above, we store the samples into a FIFO buffer implemented using the FPGA's embedded SRAM. We implement a simple memory controller to write data to the FIFO which generates the memory control signals and writes a full data word on each cycle. The embedded memory can run at rates significantly greater than 4 MHz meaning it is not a limiting factor for real-time processing. The SRAM can buffer up to 126 kB. The data stored in the FIFO can then be sent to signal processing blocks to implement filters, cryptographic functions, etc. or to non-volatile flash memory. For flash memory, we use a micro SD card which enables us to collect raw I/Q data and analyze the spectrum. The micro SD card supports two modes: native SD mode and standard SPI mode. In native SD mode, micro SD card's interface uses 4 parallel data lines to read/write data to/from the micro SD card. This mode supports a higher data rate compared to the SPI mode which only supports a 1-bit serial interface. However, we implement SPI mode since it supports the 104 Mbps data rate which we need to write data in real time. This allows us to re-use the same, simpler SPI block for multiple functions and save resources on the FPGA.

3.2.3 RF, Control and Sensor Interfaces

The AT86RF215 provides differential RF signals for both 900 MHz and 2.4 GHz and has an integrated TX/RX switch for both. At 2.4 GHz, the differential signal is transformed to a single-ended output using the 2450FB15A050E [8] balun and fed to the SKY66112 [28] front-end with the bypassable LNA and PA. Finally, after passing through a matching network, the 2.4 GHz signal is connected to an SMA output.

On the 900 MHz side, the differential output of the AT86RF215 is connected to 0896BM15E0025E [32] to convert it to a single-ended output. This must be shared between the backbone radio's two separate RF paths for transmit and receive and AT86RF215's 900 MHz single-ended signal. We choose between them using a ADG904 [19] SP4T RF switch. The single port side is connected to the SE2435L [23] 900 MHz front-end which is similar to the 2.4 GHz front-end. The MCU communicates with the I/Q radio, backbone radio, FPGA and Flash memory through SPI which it uses to send commands for changing the frequency, selecting the outputs, etc. It also has control signals for FPGA programming, 900 MHz and 2.4 GHz front-end modules, RF switch and voltage regulators for active power control. TinySDR supports common digital interfaces like SPI and I2C to communicate with digital sensors and two analog to digital converters (ADC) for interfacing with analog sensors. We leverage the internal flash memory of the MCU ($\approx$ 256 kB) and external flash memory ($\approx$ 8 MB) to store sensor data.

Table 3: **Power Domains in TinySDR.**

Component	Voltage [V]	Power Domain
MCU	1.8V	V1
FPGA	1.1, 1.8, 2.5, Vlvds	V2, V3, V4, V5
I/Q Radio	1.8< V5 <3.6	V5
Backbone Radio	1.8< V5 <3.6	V5
sub-GHz PA	3.5V	V6
2.4 GHz PA	1.8, 3.0	V3, V7
FLASH Memory	1.8	V3
Micro SD Memory	3.0	V7

3.3 Power Management Unit

Next, we present the design of our power management unit which seeks to maximize the system lifetime when running off of a 3.7 V Lithium battery. To enable long battery lifetimes we need to be able to duty-cycle our system and allow the MCU to toggle each of the above blocks ON and OFF when they are not in use. Further, different components have different supply voltage requirements and we wish to provide each one with the lowest voltage possible to minimize power usage.

Ideally we would want separate controllable voltage regulators for each component in the system. However, having many different regulators with individual controls significantly increases the complexity, number of components, and price. Moreover, it complicates the PCB design by requiring many control signals and a multitude of power planes. Therefore, there exists a trade-off between the granularity of power control and the price/complexity of a design. We outline the supply voltages needed for each component and the power domain supporting it in Table 3. Below, we show how we group components to balance power and complexity.

- **Power domain V1 (MCU).** Since the MCU is the central controller that implements power management, it needs to be powered at all times and therefore has its own power domain. To minimize its sleep current we need to use a voltage regulator with a low quiescent current. Although switching voltage regulators have higher conversion efficiency when active, they also have high quiescent currents so we instead select the TPS78218 linear regulator.
- **Power domains V2, V3, V4, V6 and V7.** These power domains provide power to blocks such as the FPGA, memory blocks, and PAs. Since these components can all be turned off when not operating, the voltage regulators for these domains should have low shut-down current during sleep and high efficiency when active. We therefore choose the TPS62240 which has a shutdown current of only 0.1 uA. It is highly efficient and is rated to support the current draw required by all components except the 900 MHz PA. To support this PA at its maximum output power we use the TPS62080 switching regulator which supports the required current.
- **Power domain V5.** V5 is a shared power domain for I/Q radio, backbone LoRa radio and FPGA I/O bank. This power domain is initially set to 1.8V to minimize power consumption, however components such as the radio chips can require higher voltage to achieve maximum output power. Therefore,

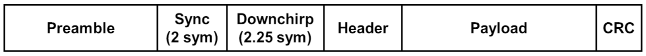


Figure 5: **LoRa Packet Structure.**

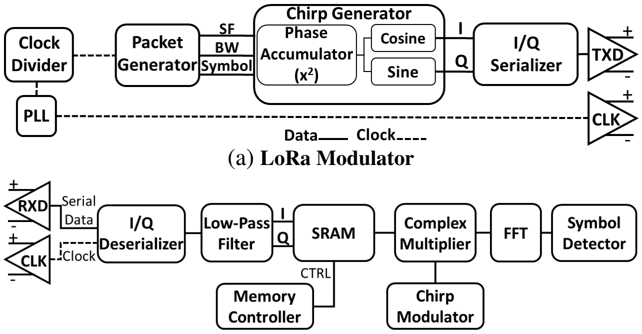
in addition to high efficiency and low shut-down current like the others, this domain should be programmable. To do this, we use Semtech SC195ULTRT [15] which provides an adjustable output that can be set from 1.8 V to 3.6 V.

3.4 Over-the-Air Programming protocol

OTA AP and MAC protocol. To update a network of tinySDR devices, we use an AP with a LoRa radio to communicate with each device sequentially. In order to propagate updates throughout a testbed or to specific tinySDR nodes, we design a MAC layer for the LoRa PHY. We pre-program a timer on the MCU to periodically turn off the FPGA and switch from IQ radio mode to the backbone radio to listen for new firmware updates. If there is an update, the AP sends a programming request as a LoRa packet with specific device IDs indicating the nodes to be programmed along with the time they should wake up to receive the update. Upon processing this packet and detecting its ID, the tinySDR node switches into update mode and sends a ready message to the AP at the scheduled time. Then, the AP transmits the firmware update as a series of LoRa packets with sequence numbers. Upon receiving each packet, the tinySDR node checks the sequence number and CRC. For a correct packet it writes the data to its flash memory and transmits an ACK to indicate correct reception. In the case of failure no ACK is sent and the AP re-transmits the corrupted packet after a timeout. After sending all the firmware data, the AP sends a final packet indicating the end of firmware update which tells the tinySDR node to reprogram itself and switch back to normal operation.

To maintain the OTA timing, we use a programmable timer that operates with a 20 PPM low-frequency crystal oscillator source. This will result in timing drift between the tinySDR node and the AP over time. However, with each update we compensate the error by sending the correct time to the tinySDR.

Compressing and decompressing the bitstream. Our system compresses data to reduce update times, however this compression must be compatible with the resources available on tinySDR. We choose the miniLZO compression algorithm [34], which is a lightweight subset of the Lempel–Ziv–Oberhumer (LZO) algorithm. Our implementation of miniLZO only requires a memory allocation equal to the size of the uncompressed data. We perform compression on the AP. The compression ratio of bitstream file varies based on the content of the bitstream, and in the worst case the compressed file could have almost the same size of the original file. This would require a maximum memory allocation of 579 kB which we cannot afford on a low-cost MCU. Instead, we first divide the original update file into blocks of 30 kB that will fit in the MCU memory. Then we compress each block separately and transmit them to the tinySDR node one by one. Considering the LoRa radio takes more power than the MCU, we immediately write the data to our dedicated programming flash memory using an SPI interface.



(a) **LoRa Modulator**

(b) **LoRa Demodulator**

Figure 6: **LoRa Implementation Block Diagrams.**

After receiving all the data we turn off the LoRa radio and decompress data. First, we allocate memory on the MCU's SRAM equal to the block size and load a block of data from flash. Next, we perform decompression and write the data in the allocated SRAM memory. Finally, we write the decompressed data back to the flash beginning at the corresponding address of the programming boot file. We repeat these steps until we decompress the full firmware update.

Over-the-air FPGA programming. After storing uncompressed programming data in flash memory, we program the FPGA. We use the MCU to set the FPGA into programming mode. When the FPGA switches to programming mode, it automatically reads its firmware directly from the flash memory using a 62 MHz quad SPI interface and programs itself. Reading from flash using quad SPI achieves programming times of 22 ms which is similar to FPGAs with embedded flash memory and results in minimal system down time. After programming is complete, it resumes operation and begins running the new firmware.

3.5 TinySDR's Architectural Considerations

We design tinySDR to achieve three main goals: i) low-power ii) low-cost and iii) over-the-air programmability. To do this, we use a low-power I/Q radio with lower bandwidth support compared to previous platforms. Since this radio is optimized for low bandwidths and in turn low sampling rates, it consumes less power during TX/RX operations. Previous platforms [31, 56] use hardware architectures that support high-bandwidth protocols such as Wi-Fi. However, we use a low-power radio and build our hardware architecture for IoT protocols around it. In addition, we design a power management system to be able to power cycle different parts of tinySDR's architecture in each operation to further reduce the power consumption. To achieve this, we use an MCU chip that enables full control of the tinySDR's blocks and power domains. We achieve minimum power consumption during sleep mode by turning off power-hungry components.

In contrast, previous architectures such as uSDR [56] use an MCU integrated with an FPGA which forces the FPGA to be always ON and increases the power consumption during sleep mode. This is because uSDR is not designed for IoT endpoints but for gateways and hence does not provide any of the architectural optimizations required for the low-power sleep operation required by IoT devices. Furthermore, by using a lower bandwidth radio and also a low-cost SRAM-based FPGA, we minimize the cost compared to platforms such as uSDR [56], Pluto SDR [31] and LimeSDR [2]. Finally, we design tinySDR to operate completely standalone on battery without the need for a wired connection to a network or a computer. To do this, we design an OTA system on tinySDR to be able to re-program the FPGA and MCU on tinySDR wirelessly. This capability does not exist on prior SDR platforms.

4 Case Studies: LoRa and BLE Beacons

4.1 LoRa Protocol with tinySDR

We choose LoRa as it is gaining popularity for IoT solutions due to its long range capabilities. Since LoRa is a proprietary standard, we begin by describing the basics of its modulation and packet structure followed by the implementation details of our modulator, demodulator and MAC protocol.

LoRa Protocol Primer. LoRa achieves long ranges by using Chirp Spread Spectrum (CSS) modulation. In CSS, data is modulated using linearly increasing frequency upchirp symbol. Each upchirp symbol has two main features: Spreading Factor (SF) and Bandwidth (BW). SF determines the number of bits in each upchirp symbol [44, 47, 69] and BW is the difference between upper and lower frequency of the chirp which together with SF determines the length of an upchirp symbol. SF and BW trade data rate for range. Data is modulated by 2^{SF} cyclic-shifts of an upchirp symbol. The starting point of the symbol in frequency domain, which is the cyclic shift of the upchirp symbol, determines its value [16]. LoRa uses SF values from 6 to 12 and BW values from 7.8125 KHz to 500 KHz to achieve PHY-layer rates of $\frac{BW}{2^{SF}} \times SF$.

Fig. 5 shows the LoRa packet structure which begins with a preamble of 10 zero symbols (upchirps with zero cyclic-shift). This is followed by the Sync field with two upchirp symbols. Next, a sequence of 2.25 downchirp symbols (chirp symbol with linearly decreasing frequency) indicate the beginning of the payload. The payload then consists of a sequence of upchirp symbols which encode a header, payload and CRC.

LoRa Modulator. Fig. 6a shows the block diagram of our LoRa modulator. We use our FPGA to implement a LoRa modulator in Verilog and stream data to AT86RF215 in I/Q mode. The modulator begins with the *Packet Generator* module which reads data either from FPGA memory for transmitting fixed packets or from the MCU, as well as LoRa configuration parameters such as SF, coding and BW. This module determines each symbol value and its corresponding cyclic-shift. Next, the *Packet Generator* sends these parameters along with the symbol values to the *Chirp Generator* module, which generates the I/Q samples of each chirp symbol in the packet using a squared phase accumulator and two lookup tables for *sine* and *cosine* function [69]. We then feed these I/Q samples into an I/Q Serializer to stream them over the LVDS interface to the I/Q radio. We generate the 64 MHz transmission clock using internal PLL of the FPGA.

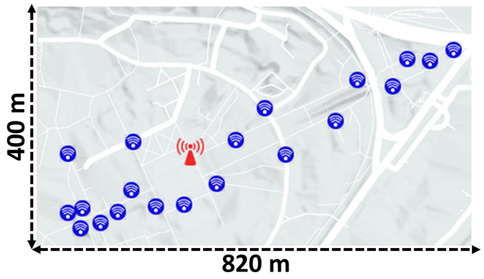


Figure 7: **Evaluation Testbed Map.**

LoRa Demodulator. Fig. 6b shows the block diagram of our LoRa demodulator. It begins by reading data from the I/Q radio into the *I/Q Deserializer* module on the FPGA which converts the serial I/Q stream to parallel I/Q for further signal processing. Next, we run the data through a 14 tap FIR lowpass filter to suppress high frequency noise and interference. We store the filtered samples in a buffer implemented using the FPGA's memory blocks. To decode the data, we use the *Chirp Generator* module from the *LoRa Modulator* described above to generate a baseline upchirp/downchirp symbol, and then we multiply that with the received chirp symbol using our *Complex Multiplier* unit. The output of the multiplication then goes to an FFT block implemented using a standard IP core from Lattice. Finally the *Symbol Detector* scans the output of the FFT for peaks and records the frequency of the peak to determine the symbol value. To detect the chirp type (upchirp/downchirp), we multiply each chirp symbol with both an upchirp and downchirp and then compare the amplitudes of their FFT peaks. The higher peak in the FFT shows higher correlation which indicates the chirp type.

LoRa MAC Layer. To demonstrate that our LoRa implementation on tinySDR is compatible with existing LoRa networks such as the LoRa Alliance's [26] The Things Network (TTN) [38], we adopt their LoRa MAC design from TTN's Arduino libraries [39] and implement it on tinySDR's MCU. TTN uses two methods for device association; Over-the-air activation (OTAA) and activation by personalization (ABP). In OTAA, each node performs a join-procedure during which a dynamic device address is assigned to a node. However, in ABP we can hard-code the device address in the device which makes it simpler since the node skips the join procedure. Our platform can support both OTAA and ABP methods.

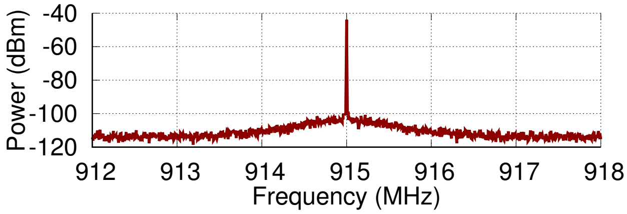


Figure 8: **TinySDR Single-Tone Frequency Spectrum.**

Table 4: **Different Operation Timing for TinySDR.**

Operation	Duration (ms)
Sleep to Radio Operation	22
Radio Setup	1.2
TX to RX	0.045
RX to TX	0.011
Frequency Switch	0.220

4.2 BLE Beacons with tinySDR

To demonstrate tinySDR's 2.4 GHz capabilities we implement Bluetooth beacons which are commonly used by IoT devices.

BLE Beacon Primer. We implement non-connectable BLE advertisements (ADV_NON_CONN_IND) which are broadcast packets used for beacons. These packets allow a low power device to broadcast its data to any listening receiver within range without the power overhead of exchanging packets to setup a connection. These packets have a bit rate of 1 Mbps in Bluetooth 4.0 or up to 2 Mbps in Bluetooth 5.0 and are generated using GFSK with a modulation index of 0.45-0.55. The GFSK modulation is binary frequency shift keying (BFSK) with the addition of a Gaussian filter to the square wave pulses to reduce the spectral width.

Generating a BLE Packet. Bluetooth advertisements consist of 6-37 octets, beginning with fixed preamble and access address fields indicating the packet type set to 0xAA and 0x8E89BED6 respectively. This is followed by the packet data unit (PDU) beginning with a 2 byte length field and followed by a manufacturer specific advertisement address and data. The final 3 bytes of the packet consist of a CRC generated using a 24-bit linear feedback shift register (LFSR) with the polynomial $x^{24}+x^{10}+x^9+x^6+x^4+x^3+x+1$. The LFSR is set to a starting state of 0x555555 and the PDU is input LSB first. The final LFSR state after inputting the PDU becomes the CRC. Data whitening is then performed over the PDU and CRC fields to eliminate long strings of zeros or ones within a packet. This is also done using a 7-bit LFSR with polynomial x^7+x^4+1. The LFSR is initialized with the lower 7 bits of the channel number the packet will be transmitted on, and each byte is input LSB first. We implement both these blocks in Verilog on the FPGA.

Packet Transmission and MAC Protocol. From this bitstream, we need to generate the I/Q samples to feed to the I/Q radio. First, we upsample and apply a Gaussian filter to the bitstream. This gives us the desired changes in frequency which we integrate to get the phase. We then feed the phase to *sine* and *cosine* functions to get the final I and Q samples, which are passed to I/Q serializer and sent to the I/Q radio. BLE divides the 2.4 GHz band into channels, each spaced 2 MHz apart, but BLE beacons are only transmitted on three advertising channels without carrier sense, typically in sequential order separated by a few hundred microseconds. This sequence is re-transmitted every advertising interval [37].

5 Evaluation

We deploy a testbed of 20 tinySDR devices across our institution's campus as shown in Fig. 7. To see if tinySDR meets the requirements for IoT endpoint devices, we characterize its power, computational resource usage, delays and cost when operating in different modes and running different protocols.

5.1 Benchmarks and Specifications

Sleep mode power. Many IoT nodes perform short, simple tasks allowing them to be heavily duty cycled which allows them to achieve battery lifetimes of years. We design tinySDR with this critical need in mind such that the MCU can actively toggle on and off power consuming components such as the radio, PAs, and FPGA to enter a low power sleep mode.

We do this by first turning off the the I/Q transceiver and LoRa radios. To reduce the static power consumption of the FPGA, we shut it down by disabling the voltage regulators that provide power to its I/O banks and core voltage. Similarly, we also turn off the PAs. Finally, we put the MCU in sleep mode LPM3 running only a wakeup timer. The measured total system sleep power in this mode was 30 uW.

The low sleep power allows for significant power savings, but also introduces latency. Table 4 shows the time required to wake up from sleep mode until the radio is active. Because we can perform the I/Q radio setup in parallel with booting the FPGA, the total wakeup time for RX and TX is 22 ms. The I/Q radio setup takes 1.2 ms, so the wakeup time is dominated by booting up the FPGA which itself takes 22 ms. We compare this to a SmartSense Temperature sensor [14] and find that tinySDR has only a 4x longer wakeup time even though it requires programming unlike commercial products that use a custom single protocol radio. Additionally many IoT devices operate at low duty cycles waiting in sleep mode for seconds or more making tinySDR's wakeup latency insignificant.

Switching delays. We also measure the switching delays for different operations on the I/Q radio as this is an important parameter for meeting MAC and protocol timing requirements. Table 4 shows that it takes 45 μs and 11 μs to switch from TX to RX mode and RX to TX mode respectively. As we see later, this is sufficient to meet the timing requirements of IoT packet ACKs and MAC protocols. Further, the delay for switching between different frequencies is only 220 us. To measure this number, we switch between 2.402 GHz, 2.426 GHz and 2.480 GHz. This switching delay is again sufficient to meet the requirements of frequency

Table 5: **TinySDR Cost Breakdown for 1000 Units.**

Components		Price
DSP	FPGA	$8.69
	Oscillator	$0.9
IQ Front-End	Radio	$5.08
	Crystal	$0.53
	2.4 GHz Balun	$0.36
	Sub-GHz Balun	$0.3
Backbone	Radio	$4.5
	Crystal	$0.4
	Flash Memory	$1.6
MAC	MCU	$3.89
	Crystals	$0.68
RF	Switch	$3.14
	Sub-GHz PA	$1.54
	2.4 GHz PA	$1.72
Power Management	Regulators	$3.7
Supporting Components	–	$4.5
Production	Fabrication [22]	$3
	Assembly [22]	$10
Total	–	**$54.53**

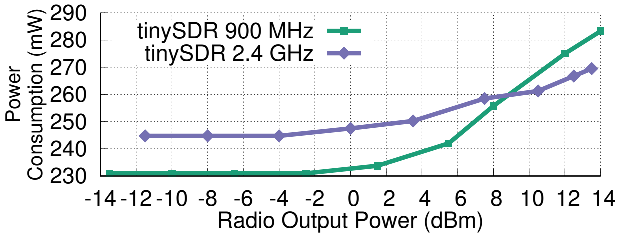


Figure 9: **Single-Tone Transmitter Power Consumption.** We show the total power consumption of tinySDR including I/Q radio, FPGA, MCU and regulators at different transmitter output power. This is 15-16 times lower power consumption than the USRP E310 embedded SDR.

hopping during Bluetooth advertising.

Transmitter performance. First, we implement a single-tone modulator on the FPGA that generates the appropriate I/Q samples and streams them over LVDS to the radio. We connect the output to an MDO4104b-6 [11] spectrum analyzer and observe a single tone, shown in Fig. 8, with no unexpected harmonics introduced by the modulator.

Next we measure the end-to-end DC power consumption of our system including the I/Q radio, FPGA, MCU and regulators to see how it scales with RF output power. We vary our radio output power while transmitting a single tone and use a Fluke 287 multimeter to measure its DC power draw. Fig. 9 shows the power consumption of tinySDR for 900 MHz and 2.4 GHz operation. Interestingly, we observe the DC power is constant at low RF power but increases as expected beyond some RF power level. TinySDR consumes 231 mW when transmitting at 0 dBm, and for comparison the end-to-end power consumption of the USRP E310 is 16x higher under the same conditions. Similarly tinySDR consumes 283 mW at its 14 dBm setting while the USRP E310 is 15x higher. In addition, we measure the peak current consumption of tinySDR while transmitting a single-tone on the I/Q radio. The peak current consumption is 105 mA when tinySDR boots up the FPGA and then starts transmitting using the I/Q radio. This current is less than the maximum current supported by a typical LiPo 1200 mAh battery [13].

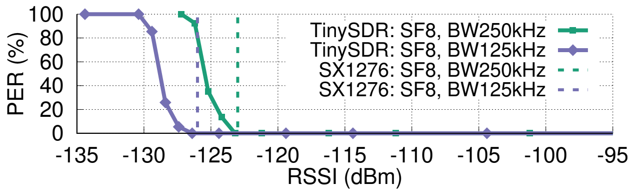


Figure 10: **LoRa Modulator Evaluation.** We evaluate our LoRa modulator in comparison with Semtech LoRa chip.

Cost. We also analyze the cost which is an important practical consideration for real world deployment at scale. Table 5 shows a detailed breakdown of cost including each component as well as PCB fabrication and assembly based on quotes for 1000 units [22], where the overall cost is around $55 each.

5.2 Evaluating the Case Studies

LoRa using tinySDR. We evaluate various different components of tinySDR using LoRa as a case study.

LoRa modulator. To evaluate this, we use our LoRa modulator to generate packets with three byte payloads using a spreading factor of $SF = 8$ and bandwidths of 250 kHz and 125 kHz which we transmit at -13 dBm. We receive the output of tinySDR on a Semtech SX1276 LoRa transceiver [35] which we use to measure the packet error rate (PER) versus RSSI and plot the results in Fig. 10. We compare our LoRa modulator to transmissions from an SX1276 LoRa transceiver. The plots show that we can achieve a comparable sensitivity of -126 dBm which is the LoRa sensitivity for $SF = 8$ and $BW = 125kHz$ configuration. This is true for both configurations, which shows that our low-power SDR can meet the sensitivity requirement of LPWAN IoT protocols.

LoRa demodulator. Next we evaluate our LoRa demodulator on tinySDR. To test this, we use transmissions from a Semtech SX1276 LoRa transceiver and use tinySDR to receive these transmissions. The LoRa transceiver transmits packets with two configurations using a spreading factor of 8 and bandwidths of 250 kHz and 125 kHz. We record the received RF signals in the FPGA memory and run them through our demodulator to compute a chirp symbol error rate. Note that the Semtech LoRa transceiver does not give access to its symbol error rate, but since we have access to I/Q samples, we can compute it on our platform. We plot the results in Fig. 11 as a function of the LoRa RSSI values. Our LoRa demodulator can demodulate chirp symbols down to -126 dBm which is LoRa protocol sensitivity at $SF = 8$ and $BW = 125kHz$. Both the LoRa modulator and demodulator run in real-time.

Resource allocation. Next, we evaluate the resource utilization of our LoRa PHY implementation on the FPGA. Table 6 shows the size for implementing the modulator and demodulator on our FPGA using different SFs. Our LoRa modulator supports all LoRa configurations with different SF with no additional cost. However, in the LoRa demodulator, we need

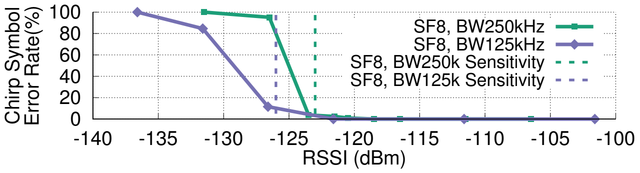


Figure 11: **LoRa Demodulator Evaluation.** We evaluate our LoRa demodulator by demodulating chirp symbols at different RSSI.

Table 6: **FPGA Utilization for LoRa Protocol.**

SF	LoRa TX (LUT)	LoRa RX (LUT)
6	976 (4%)	2656 (10%)
7	976 (4%)	2670 (10%)
8	976 (4%)	2700 (11%)
9	976 (4%)	2742 (11%)
10	976 (4%)	2786 (11%)
11	976 (4%)	2794 (11%)
12	976 (4%)	2818 (11%)

FFT blocks with different sizes to support different SF configurations. This table shows that our FPGA has sufficient resources to support multiple configurations of LoRa and still leave space for other custom operations.

LoRa MAC. We implement the LoRa MAC based on TTN's Arduino libraries [39]. TTN protocol together with control for the I/Q radio, backbone radio, FPGA, PMU and decompression algorithm for OTA take only 18% of MCU resources. Also, as shown in Table 4, our timings are well within the requirements for LoRaWAN specifications [26].

We also measure the power consumption of our platform for LoRa packet transmission and reception. LoRa packet transmission with $SF = 9$ and $BW = 500\ kHz$ and radio output power of 14 dBm consumes a total power of 287 mW from which 179 mW is for the radio and the rest is from the FPGA and MCU. LoRa packet reception consumes 186 mW with radio taking 59 mW.

BLE using tinySDR. Next, we evaluate tinySDR using BLE beacons as a case study. First, we measure the impact of our BLE beacons transmitted from tinySDR using the TI CC2650 [21] BLE chip as a receiver. We do this by configuring tinySDR to transmit BLE beacons at a rate of 1 packet per second. We transmit 100 packets and set the CC2650 BLE chip to report bit error rate (BER). Fig. 12 shows the BER as a function of the received RSSI as reported by the CC2650 BLE chip. The plot shows that we achieve a sensitivity of -94 dBm. This is within 2 dB of the CC2650 BLE chipset's sensitivity, defined by a BER threshold of 10^{-3}.

Next we evaluate the latency of our BLE implementation as BLE beacons are typically transmitted in sequence by hopping between three different advertising channels. We measure the minimum time tinySDR takes to switch between these frequencies by connecting its output to a 2.4 GHz envelope detector and using an MDO4104B-6 oscilloscope to measure the time delay between transmissions. Fig. 13 plots the envelope of three BLE beacons in the time-domain transmitted on the different advertising channels and shows that our system can transmit packets with as little as 220 us delay between beacons. The corresponding result when a iPhone 8 transmits beacons is 350 us. Finally, generating BLE beacons requires only 3% of the FPGA resources on the tinySDR and it could run for over 2 years on a 1000 mAh battery when transmitting once per second.

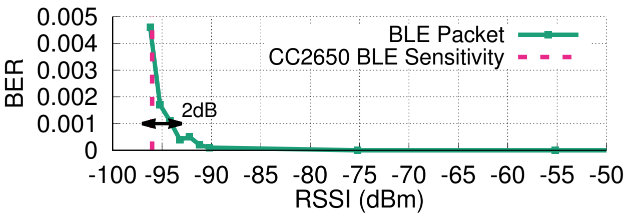


Figure 12: **BLE evaluation.** BLE beacons at different power levels.

5.3 Over-the-Air Programming

An effective OTA programming system should both minimize use of system resources such as power as well as network downtime. Considering the time to reprogram the FPGA and microcontroller from flash is fixed, the downtime for programming a node depends on the amount of data sent and the throughput which varies with SNR.

Raw programming files for our FPGA are 579 kB, however we compress our data using miniLZO. While the exact compression ratio depends on FPGA utilization, our LoRa program compresses to 99 kB and BLE to 40 kB. Our microcontroller programs for both LoRa and BLE are approximately 78 kB and are both compressed to 24 kB. When dividing the files into packets, we would ideally minimize the preamble length and maximize packet length to reduce overhead, however long packets with short preambles lead to higher PER. We choose a preamble of 8 chirps and packets of 60 B which we find balances the trade-off of protocol overhead versus range in our experiments.

To see the impact on a real deployment, we evaluate the time required to program tinySDR nodes in our 20 device testbed shown in Fig. 7. We set up a LoRa transceiver configured with $SF = 8$, $BW = 500\ kHz$ and $CodingRate = 6$ connected to a patch antenna transmitting at 14 dBm as an AP and measure the time it takes to program the tinySDR devices at each location, according to our protocol. We transmit the compressed FPGA and MCU programming data for LoRa and BLE and plot the results as a CDF in Fig. 14. The plots show that the LoRa FPGA requires an average programming time of 150 s while BLE, FPGA, and MCU require 59 s and 39 s respectively due to their smaller file size. Decompressing these received files only takes a maximum of 450 ms. The variation of the programming time between different nodes is caused by the variation in the wireless channel and hence the number of re-transmissions for each tinySDR node is different.

Our OTA programming system components, backbone radio and MCU, consume an average energy of 6144 mJ for receiving a LoRa FPGA update and 2342 mJ for a BLE FPGA update when using 14 dBm output power. Using a 1000 mAh

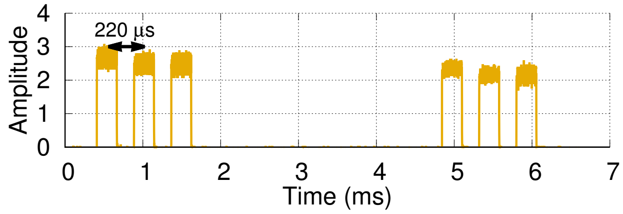


Figure 13: **BLE Beacons Signal.** We show BLE beacon transmissions on three advertising channels from tinySDR using an envelope detector.

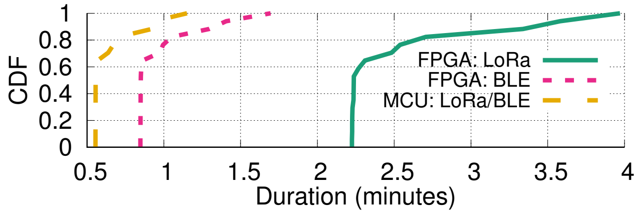


Figure 14: **OTA Programming Time.** We show CDF of OTA programming time for programming LoRa and BLE implementations on tinySDR.

LiPo battery, we could OTA program each tinySDR node with LoRa 2100 times and BLE 5600 times. Assuming OTA programming of once per day, the average power consumption would be 71 uW and 27 uW respectively for LoRa and BLE.

6 Research Study: Concurrent Reception

An SDR designed for IoT endpoints that can provide I/Q transmission and reception capability opens up opportunities for addressing multiple research questions in IoT networks.

In this section, we focus on the following question: Can a *low-power IoT endpoint* device decode multiple concurrent LoRa transmissions at the same time? LoRa supports long range communication for IoT devices and is gaining popularity as a low-power wide area networking (LPWAN) standard. Supporting long ranges introduces new challenges since it increases the probability of collisions in large scale city-wide deployments. While recent works [44, 47] have explored the feasibility of enabling concurrent LoRa transmissions, they have been designed for decoding on a gateway-style USRP device. In fact, most concurrent transmission techniques in our community [44, 45, 52] have been prototyped on USRPs and it is unclear if a low-power IoT endpoint device can decode concurrent transmissions in real-time within its stringent power and resource constraints. TinySDR enables us to explore such questions and design MAC protocols for decoding concurrent transmissions on IoT endpoints.

Using orthogonal LoRa codes. Here we explore a specific way of enabling concurrent transmissions in LoRa: using orthogonal codes. Specifically, to allow multiple LoRa nodes to communicate at the same time, we exploit LoRa's support for orthogonal transmissions [16] which can occupy the same frequency channel without interfering with each other. Two chirp symbols are orthogonal when they have a different chirp slope. For a chirp with a spreading factor of SF and bandwidth of BW, the chirp slope is given by: $\frac{BW^2}{2^{SF}}$ [47].

Decoding concurrent transmissions on tinySDR. In order to receive concurrent LoRa transmissions, tinySDR must be able to demodulate LoRa upchirp symbols with different slopes. Suppose we have two LoRa transmissions that use different spreading factor and bandwidth configurations: SF_1, BW_1 and SF_2, BW_2. To decode them concurrently, we implement decoders similar to Fig. 6b for each chirp configuration in parallel on our FPGA. Specifically, we first generate a corresponding downchirp symbol for each configuration in real-time using our chirp generator. Note that we generate each chirp with its corresponding configuration on the FPGA and we do not use pre-generated chirps on the FPGA. We then correlate the received signals with their corresponding downchirp symbols using time domain multiplication. After correlation, we take the appropriate length FFT of the result.

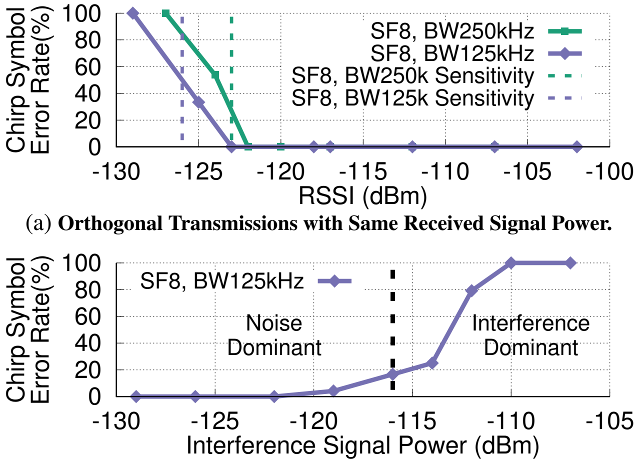
Evaluation. We evaluate three key aspects of our design: 1) the platform's effectiveness in decoding concurrent transmissions across a range of RSSI values, 2) the power consumption at the endpoint device while decoding concurrent transmissions and 3) the computational resources required.

We use two SX1276 LoRa transceivers as our transmitters and set them to transmit continuously at two different settings: they both use a spreading factor of $SF = 8$ but have two different bandwidth setups, $BW_1 = 125kHz$ and $BW_2 = 250kHz$. We set the two to send random chirp symbols. The tinySDR platform decodes these two concurrent transmissions and computes the chirp symbol error rate for each transmission. We evaluate two scenarios: 1) when the two transmitters have a similar power level at the receiver, 2) fix the power of one of the transmitters and increase the power of the other one.

Fig. 15a shows the results when the two transmissions have similar power at the receiver. We lose around 2 dB and 0.5 dB sensitivity for concurrent demodulation of LoRa configurations with $BW_1 = 125kHz$ and $BW_2 = 250kHz$. This is because while in theory the two chirps are orthogonal, in practice, the chirps are created in the digital domain with discrete frequency steps which introduces some non-orthogonality.

Fig. 15b shows the results when the first LoRa transmitter $BW_1 = 125kHz$ is received near its sensitivity of -123 dBm and and the second LoRa transmitter changes its power. Here, the chirp symbol error rate is affected when the other transmission's power is higher than -116 dBm. When two concurrent transmissions are present, one acts as an interferer when decoding the other. The combined power of noise and the interferer, $P_{I,N}$, determines the error rate. When sweeping the power of interferer, at first the $P_{I,N}$ is dominated by noise and we should not see much effect on error rate. Then at some point their power would be equal which results in a 3 dB increase of $P_{I,N}$ and hence 3 dB sensitivity loss after which the error rate is determined by the interferer power. This demonstrates the need for power control for concurrent transmissions to be received on IoT endpoints.

Our parallel demodulation implementation, uses only 17% of the FPGA resources. This concurrent demodulation imple-



(a) **Orthogonal Transmissions with Same Received Signal Power.**

(b) **Orthogonal Transmissions with Different Received Signal Power**. We set the power of LoRa transmission with $BW_1 = 125kHz$ to -123 dBm and increase the power of the other one.

Figure 15: **Orthogonal LoRa Demodulation Evaluation**

mentation consumes 207 mW. Note that Semtech gateway solutions such as the SX1308 [29] can receive multiple transmissions. But, to the best of our knowledge we are the first to show that concurrent LoRa transmissions can be decoded on an IoT endpoint while meeting its power and computational requirements. This would have been difficult to do without tinySDR.

7 Conclusion and Research Opportunities

This paper presents the first SDR platform specifically tailored to the needs of IoT endpoints that can be used for large scale IoT network deployments. The goal of tinySDR is to provide a platform that can catalyze research in IoT networks.

Research on PHY/MAC protocols. TinySDR presents an opportunity for researchers to avoid the time consuming endeavor of building their own custom hardware and instead focus on PHY/MAC protocol innovations across the stack: What is the trade-off between packet length and overall throughput? Are there benefits of rate adaptation? What about concurrent transmissions from IoT devices? One could also create multi-hop IoT PHY/MAC innovations, which have not been explored well given the lack of a flexible platform.

Research on IoT localization. TinySDR could also be used to build localization systems as it gives access to I/Q signals and therefore phase across the 2.4 GHz and 900 MHz bands, which forms the basis for many localization algorithms [62]. One could also explore distributed localization solutions that combine the phase information across a distributed set of sensors to create a large MIMO sensing system.

Machine learning on IoT devices. The FPGA on tinySDR opens up exciting opportunities [42] for exploring machine learning algorithms on-board. This would allow researchers to explore trade-offs between the power overhead of running an on-board classifier versus sending data to the cloud. This could also enable use of high bandwidth sensors such as cameras and microphones where the power bottleneck may be communication rather than sensing.

Low power backscatter readers. Recent work on ambient backscatter [51, 53, 54, 59, 71] aims to achieve ultra-low power communication for IoT devices. Many of these proposals require either a single-tone generator [54] or a custom receiver to decode the backscatter transmissions [49, 50, 61, 65]. TinySDR can be used as a building block to achieve a battery-operated backscatter signal generation and receiver.

Better programming interface and protocols. In addition to IoT research opportunities, we can also improve our platform in multiple ways. TinySDR currently requires users to write Verilog or VHDL to program the FPGA and C code for programming the microcontroller. Future versions can incorporate a pipeline to use high level synthesis tools or integrate with GNUradio for easy prototyping. Further, tinySDR uses a simple MAC protocol for programming with a focus on using minimal system resources to allow for other custom software; however we could explore modified MAC protocols that simultaneously broadcast the updates across the network to reduce programming time.

Acknowledgments. We thank Aaron Schulman, Mohamad Katanbaf, and the anonymous reviewers for their helpful feedback on the paper. This work was funded in part by NSF awards CNS-1812554, CNS-1452494, CNS-1823148 and Google Faculty Research Awards.

References

[1] bladerf 2.0 micro. https://www.nuand.com/bladeRF_2_micro-brief.pdf.

[2] Limesdr. https://myriadrf.org/projects/limesdr/.

[3] Limesdr-mini. https://wiki.myriadrf.org/LimeSDR-Mini.

[4] An overview of lvds technology. http://www.ti.com/lit/an/snla165/snla165.pdf.

[5] Rtlsdr rtl2832u dvb-t tuner dongles. https://www.rtl-sdr.com/buy-rtl-sdr-dvb-t-dongles/.

[6] Usrp b200mini. https://www.ettus.com/content/files/USRP_B200mini_Data_Sheet.pdf.

[7] Usrp e310. https://www.ettus.com/content/files/USRP_E310_Datasheet.pdf.

[8] 2.45 ghz balun, filter combination, 2003. https://www.mouser.com/datasheet/2/611/JTI_Balun-Filter-2450FB15A050_2006-09-242325.pdf.

[9] Max5189 datasheet, 2003. https://datasheets.maximintegrated.com/en/ds/MAX5186-MAX5189.pdf.

[10] Max2831 datasheet, 2011. https://datasheets.maximintegrated.com/en/ds/MAX2831-MAX2832.pdf.

[11] Mdo4000b series datasheet, 2013. http://www.testequipmenthq.com/datasheets/TEKTRONIX-MDO4104B-6-Datasheet.pdf.

[12] Ad9364 transceiver datasheet, 2014. https://www.analog.com/media/en/technical-documentation/data-sheets/AD9364.pdf.

[13] Li-polymer battery technology specification, 2014. https://cdn-shop.adafruit.com/product-files/258/C101-_Li-Polymer_503562_1200mAh_3.7V_with_PCM_APPROVED_8.18.pdf.

[14] Smartsense temp/humidity manual, 2014. https://support.smartthings.com/hc/en-us/articles/203040294-SmartSense-Temperature-Humidity-Sensor.

[15] 3.5mhz, 500ma synchronous step down dc-dc regulator, 2015. https://www.semtech.com/uploads/documents/sc195.pdf.

[16] Lora modulation basics, 2015. https://www.semtech.com/uploads/documents/an1200.22.pdf.

[17] Ad9361 transceiver datasheet, 2016. https://www.analog.com/media/en/technical-documentation/data-sheets/AD9361.pdf.

[18] Ad9363 transceiver datasheet, 2016. https://www.analog.com/media/en/technical-documentation/data-sheets/AD9363.pdf.

[19] Adg904 datasheet by analog devices, 2016. http://www.analog.com/media/en/technical-documentation/data-sheets/ADG904.pdf.

[20] At86rf215 datasheet, 2016.

[21] Cc2650 simplelink datasheet by ti, 2016. http://www.ti.com/lit/ds/symlink/cc2650.pdf.

[22] Pcbminions inc., 2016. https://pcbminions.com/.

[23] Se2435l power amplifier datasheet, 2016. http://www.skyworksinc.com/uploads/documents/SE2435L_202412I.pdf.

[24] Sx1276 datasheet by semtech, 2016. https://www.semtech.com/uploads/documents/sx1276.pdf.

[25] Lms7002m datasheet, 2017. https://limemicro.com/app/uploads/2017/07/LMS7002M-Data-Sheet-v3.1r00.pdf.

[26] Lora alliance, 2017. https://lora-alliance.org/sites/default/files/2018-04/lorawantm_specification_-v1.1.pdf.

[27] Msp432p401r, msp432p401m simplelink mixed-signal microcontrollers datasheet, 2017. http://www.ti.com/lit/ds/symlink/msp432p401r.pdf.

[28] Sky66112 power amplifier datasheet, 2017. http://www.skyworksinc.com/uploads/documents/SKY66112_11_203225L.pdf.

[29] Sx1308 datasheet by semtech, 2017. https://www.semtech.com/uploads/documents/sx1308.pdf.

[30] Ad9228 datasheet, 2018. https://www.analog.com/media/en/technical-documentation/data-sheets/ad9228.pdf.

[31] Adalm-pluto overview, 2018. https://wiki.analog.com/university/tools/pluto.

[32] Atmel at86rf215 868/915/928 mhz impedance matched balun + lpf, 2018. https://www.mouser.com/datasheet/2/611/0896BM15E0025-1518705.pdf.

[33] Lfe5u fpga family datasheet, 2018. http://www.latticesemi.com/view_document?document_id=50461.

[34] minilzo implementation, 2018. http://www.oberhumer.com/opensource/lzo/#abstract.

[35] Sx1257 datasheet by semtech, 2018. https://www.semtech.com/uploads/documents/DS_SX1257_V1.2.pdf.

[36] Usrp x-300, 2018. https://www.ettus.com/product/details/X300-KIT.

[37] Bluetooth core specification v5.1, Jan. 2019. https://www.bluetooth.com/specifications/bluetooth-core-specification.

[38] The things network, 2019. https://www.thethingsnetwork.org/.

[39] The things network arduino library, 2019. https://github.com/TheThingsNetwork/arduino-device-lib.

[40] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. R. Cavallaro, and A. Sabharwal. Warp, a unified wireless network testbed for education and research. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, pages 53–54. IEEE, 2007.

[41] N. Anand, E. Aryafar, and E. W. Knightly. Warplab: a flexible framework for rapid physical layer design. In *Proceedings of the 2010 ACM workshop on Wireless of the students, by the students, for the students*, pages 53–56. ACM, 2010.

[42] J. Chan, A. Wang, A. Krishnamurthy, and S. Gollakota. Deepsense: Enabling carrier sense in low-power wide area networks using deep learning. *CoRR*, abs/1904.10607, 2019.

[43] P. Dutta, Y.-S. Kuo, A. Ledeczi, T. Schmid, and P. Volgyesi. Putting the software radio on a low-calorie diet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 20. ACM, 2010.

[44] R. Eletreby, D. Zhang, S. Kumar, and O. Yağan. Empowering low-power wide area networks in urban settings. SIGCOMM '17.

[45] S. Gollakota and D. Katabi. Zigzag decoding: Combating hidden terminals in wireless networks. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, 2008.

[46] Y. Guddeti, R. Subbaraman, M. Khazraee, A. Schulman, and D. Bharadia. Sweepsense: Sensing 5 ghz in 5 milliseconds with low-cost radios. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 317–330, 2019.

[47] M. Hessar, A. Najafi, and S. Gollakota. Netscatter: Enabling large-scale backscatter networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 271–284, 2019.

[48] R. Hildebrandt. The pseudo dual-edge d-flip-flop, 2011.

[49] V. Iyer, J. Chan, I. Culhane, J. Mankoff, and S. Gollakota. Wireless analytics for 3d printed objects. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 141–152, New York, NY, USA, 2018. ACM.

[50] V. Iyer, J. Chan, and S. Gollakota. 3d printing wireless connected objects. *ACM Trans. Graph.*, 36(6), Nov. 2017.

[51] V. Iyer, V. Talla, B. Kellogg, S. Gollakota, and J. Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Proceedings of the 2016 ACM SIGCOMM Conference*.

[52] S. Katti, S. Gollakota, and D. Katabi. Embracing wireless interference: Analog network coding. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 397–408. ACM, 2007.

[53] B. Kellogg, A. Parks, S. Gollakota, J. R. Smith, and D. Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *Proceedings of the 2014 ACM Conference on SIGCOMM*.

[54] B. Kellogg, V. Talla, S. Gollakota, and J. R. Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *NSDI 16*.

[55] A. Khattab, J. Camp, C. Hunter, P. Murphy, A. Sabharwal, and E. W. Knightly. Warp: a flexible platform for clean-slate wireless medium access protocol design. *ACM SIGMOBILE Mobile Computing and Communications Review*, 12(1):56–58, 2008.

[56] Y.-S. Kuo, P. Pannuto, T. Schmid, and P. Dutta. Reconfiguring the software radio to improve power, price, and portability. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 267–280. ACM, 2012.

[57] Y.-S. Kuo, T. Schmid, and P. Dutta. A compact, inexpensive, and battery-powered software-defined radio platform. In *Proceedings of the 11th international conference on Information Processing in Sensor Networks*, pages 137–138. ACM, 2012.

[58] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. Soda: A low-power architecture for software radio. *ACM SIGARCH Computer Architecture News*, 34(2):89–101, 2006.

[59] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith. Ambient backscatter: Wireless communication out of thin air. SIGCOMM '13.

[60] G. J. Minden, J. B. Evans, L. Searl, D. DePardo, V. R. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, et al. Kuar: A flexible software-defined radio development platform. In *2007 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks*, pages 428–439. IEEE, 2007.

[61] S. Naderiparizi, M. Hessar, V. Talla, S. Gollakota, and J. R. Smith. Towards battery-free hd video streaming. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[62] R. Nandakumar, V. Iyer, and S. Gollakota. 3d localization for sub-centimeter sized devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, 2018.

[63] R. Narayanan and S. Kumar. Revisiting software defined radios in the iot era. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. ACM, 2018.

[64] M. C. Ng, K. E. Fleming, M. Vutukuru, S. Gross, H. Balakrishnan, et al. Airblue: A system for cross-layer wireless protocol development. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, page 4. ACM, 2010.

[65] A. Saffari, M. Hessar, S. Naderiparizi, and J. R. Smith. Battery-free wireless video streaming camera system. In *2019 IEEE International Conference on RFID (RFID)*, pages 1–8. IEEE, 2019.

[66] C. Shepard, A. Javed, and L. Zhong. Control channel design for many-antenna mu-mimo. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15.

[67] C. Shepard, H. Yu, N. Anand, E. Li, T. Marzetta, R. Yang, and L. Zhong. Argos: Practical many-antenna base stations. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12.

[68] P. D. Sutton, J. Lotze, H. Lahlou, S. A. Fahmy, K. E. Nolan, B. Ozgul, T. W. Rondeau, J. Noguera, and L. E. Doyle. Iris: an architecture for cognitive radio networking testbeds. *IEEE communications magazine*, 48(9):114–122, 2010.

[69] V. Talla, M. Hessar, B. Kellogg, A. Najafi, J. R. Smith, and S. Gollakota. Lora backscatter: Enabling the vision of ubiquitous connectivity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2017.

[70] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker. Sora: high-performance software radio using general-purpose multi-core processors. In *6th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 09)*, 2009.

[71] A. Wang, V. Iyer, V. Talla, J. R. Smith, and S. Gollakota. FM backscatter: Enabling connected cities and smart fabrics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.

[72] H. Wu, T. Wang, Z. Yuan, C. Peng, Z. Li, Z. Tan, B. Ding, X. Li, Y. Li, J. Liu, et al. The tick programmable low-latency sdr system. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 101–113. ACM, 2017.

[73] J. Zhang, X. Zhang, P. Kulkarni, and P. Ramanathan. Openmili: a 60 ghz software radio platform with a reconfigurable phased-array antenna. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 162–175. ACM, 2016.