Bfree

Enabling battery-free sensor prototyping with python

Kortbeek, Vito; Bakar, Abu; Cruz, Stefany; Yildirim, Kasim Sinan; Pawełczak, Przemysław; Hester, Josiah

DOI
10.1145/3432191

Publication date
2020

Document Version
Final published version

Published in
Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies

BFree: Enabling Battery-free Sensor Prototyping with Python

VITO KORTBEEK, Delft University of Technology, The Netherlands
ABU BAKAR, Northwestern University, USA
STEFANY CRUZ, Northwestern University, USA
KASIM SINAN YILDIRIM, University of Trento, Italy
PRZEMYSŁAW PAWEŁCZAK, Delft University of Technology, The Netherlands
JOSIAH HESTER, Northwestern University, USA

Building and programming tiny battery-free energy harvesting embedded computer systems is hard for the average maker because of the lack of tools, hard to comprehend programming models, and frequent power failures. With the high ecologic cost of equipping the next trillion embedded devices with batteries, it is critical to equip the makers, hobbyists, and novice embedded systems programmers with easy-to-use tools supporting battery-free energy harvesting application development. This way, makers can create untethered embedded systems that are not plugged into the wall, the desktop, or even a battery, providing numerous new applications and allowing for a more sustainable vision of ubiquitous computing. In this paper, we present BFree, a system that makes it possible for makers, hobbyists, and novice embedded programmers to develop battery-free applications using Python programming language and widely available hobbyist maker platforms. BFree provides energy harvesting hardware and a power failure resilient version of Python, with durable libraries that enable common coding practice and off the shelf sensors. We develop demonstration applications, benchmark BFree against battery-powered approaches, and evaluate our system in a user study. This work enables makers to engage with a future of ubiquitous computing that is useful, long-term, and environmentally responsible.

CCS Concepts: • **Human-centered computing** → **Ubiquitous and mobile computing**; **Ubiquitous and mobile computing systems and tools**; • **Computer systems organization** → **Embedded and cyber-physical systems**;

Additional Key Words and Phrases: Energy Harvesting, Intermittent Computing, Making

ACM Reference Format:
Vito Kortbeek, Abu Bakar, Stefany Cruz, Kasım Sinan Yıldırım, Przemysław Pawełczak, and Josiah Hester. 2020. BFree: Enabling Battery-free Sensor Prototyping with Python. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 4, Article 135 (December 2020), 39 pages. https://doi.org/10.1145/3432191

1 INTRODUCTION

The maker and hobbyist movement has brought computing and programming to the masses [55, 69, 84]. Hobbyists can now build functional embedded computing systems, such as temperature sensors, motion-controlled actuators, interactive lighting systems, or simple robotic platforms, that used to be the purview of experts only. Building

Authors' addresses: Vito Kortbeek, Delft University of Technology, Delft, The Netherlands, v.kortbeek-1@tudelft.nl; Abu Bakar, Northwestern University, Evanston, IL, USA, abubakar@u.northwestern.edu; Stefany Cruz, Northwestern University, Evanston, IL, USA, stefanycruz2024@u.northwestern.edu; Kasım Sinan Yıldırım, University of Trento, Trento, Italy, kasimsinan.yildirim@unitn.it; Przemysław Pawełczak, Delft University of Technology, Delft, The Netherlands, p.pawelczak@tudelft.nl; Josiah Hester, Northwestern University, Evanston, IL, USA, josiah@northwestern.edu.

2474-9567/2020/12-ART135 $15.00
https://doi.org/10.1145/3432191

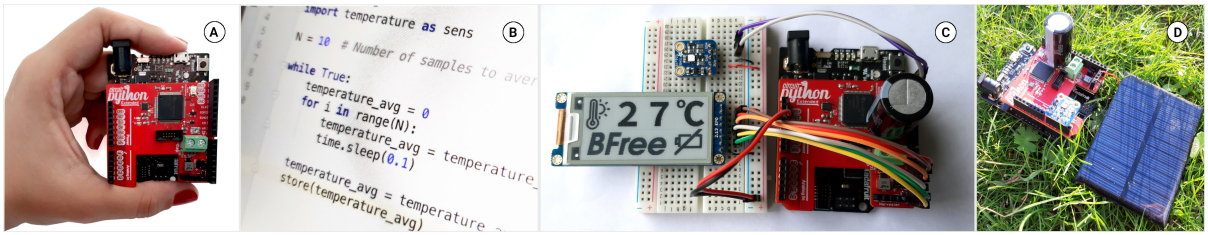


Fig. 1. BFree shield connected to an embedded hobbyist-grade embedded computer (Adafruit Metro M0 board [3]) enables developing battery-free applications powered by ambient energy **(A)**. Makers and technology hobbyists can for the first time program a battery-free platform in Python **(B)**, and can easily connect sensors to BFree for fast prototyping **(C)**. BFree can be deployed indefinitely, supporting application domains where untethered and long-term sensing is desired **(D)**.

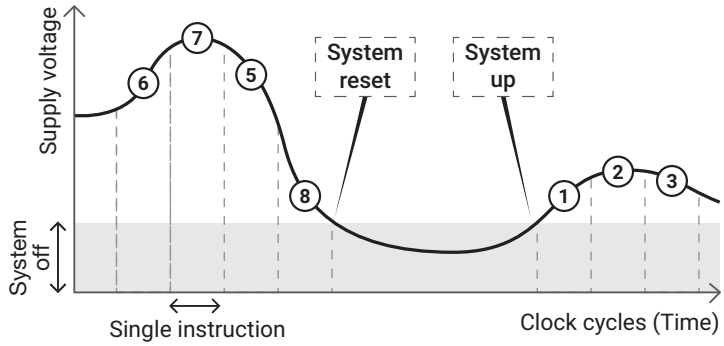
with Arduino [8], and more recent platforms such as CircuitPython [4], MBed [11], Micro:bit [23], or Microsoft MakeCode [67], has empowered novice developers and makers to think beyond the traditional computing constraints with a desktop or laptop and what can be accomplished with computing everywhere. All these platforms allow for quick prototyping and programming of complex embedded systems, reducing the time to demo.

Concurrently, concerns about the sustainability of computing within wireless sensor networks [42, 77], and more generally, the Internet of Things [87] have arisen. The proliferation of batteries and the difficulties of disposal, along with the huge growth in the number of devices, has led to efforts to develop *battery-free* embedded devices that can be powered by ambient energy (vibrations, radio frequency transmissions, light, etc.). These devices are likely the future of the Internet of Things [33, 40, 78]. Simply, these devices have a lower environmental impact, are cheaper, and can be deployed maintenance-free for much longer than their battery-powered counterparts. Battery-free devices allow very unique applications recently demonstrated and deployed in consumer-grade products like phones [91], in space [17], in implantables [59, 83], in machine learning [51], in handheld gaming consoles [19], and even underwater [38]. Despite this emerging technology trend, the majority of hobbyist and maker projects are still plugged into the wall or laptop, or battery-powered, meaning that the hobbyist programmer is *unprepared for (or not even aware of) a battery-free future* [32, 74] and does not have the ability to create novel and exciting untethered applications.

Currently, developing battery-free applications powered by ambient energy only is in the realm of experts, as a combination of system-level difficulties that span hardware, software, and design make it difficult to work with these devices. The main difficulty comes from the frequent power failures caused by energy scarcity; energy is harvested from solar panels, radio frequency, kinetic, or other ambient sources, stored briefly in "small" capacitors and then used to power computational, sensing, and communication tasks. Because of this interrupted operation (in extreme cases, sometimes multiple power failures a second) [82, 92], the field of *intermittent computing systems* has arisen to create systems, architecture, programming models, and tools for masking or handling these power failures gracefully [32, 33, 57].

The de facto programming language for programming battery-free systems is the C programming language. Developers must program in C and use specialized software (i.e., runtimes) and hardware (i.e., non-volatile memory) to allow a regular program to run correctly and consistently on intermittent power. In general, these runtimes allow programs to automatically (or manually) checkpoint[1] state before a power failure and then restore that state to allow for continuation of the program [16, 58, 60, 82, 102]. However, using these runtimes requires in-depth knowledge

[1]By *checkpoint* we denote the act of saving volatile state to non-volatile memory to allow for later restoration and therefore the continuation from that point onward.




```
1  import time, board, sense
2  N = 10
3  while True:
4      temp_avg = 0
5      for i in range(N):
6          temp_avg = temp_avg + sense.temp
7          time.sleep(0.1)
8      temp_avg = temp_avg / N
9      store(temp_avg)
```

Fig. 2. An illustrative power supply trace of a battery-free device executing a simple Python program. The program never reaches line 9, where it stores the result. With energy harvesting devices, power failures can occur at any time between any two lines of code.

of tools like LLVM, GCC, Make, and custom APIs—again, things that are standard for experts but *arcane for the novice*.

If we are to enable hobbyists to program these devices and participate in the future of sustainable computing, we must streamline this process. We foresee that Python is the strongest programming language candidate to enable this. It is one of the fastest-growing and most popular languages currently ranked as the number one language in 2020 based on IEEE Spectrum multi-metric multi-source study [14]. It is the most searched language on online language tutorials [13] and one of the top three languages measured through StackOverflow and GitHub data mining and developer surveys [72]. Python is an interpreted language popular with beginners, hobbyists, and advanced users for myriad applications from machine learning to embedded programming [95]. Its simplicity and ubiquity have already made it an ideal language for electronic hobbyists and makers with the advent of MicroPython and AdaFruit's CircuitPython [4].

However, using Python to develop battery-free applications is not trivial—an illustration of a battery-free device executing a simple Python code is shown in Figure 2. Power failures cause the program to restart from the top of the program, keeping the entire program from finishing, wasting time, resources, and essentially making the device useless. These power failures also change the peripherals state (such as a connected sensor or radio) and cause delays (as the time it takes the battery-free device to restart) to be too long and too energy-intensive for the device to function on harvested energy only.

In this paper, we propose an end-to-end system, BFree, shown in Figure 1 that seeks to *fill the systems gap preventing hobbyists and makers from participating in the battery-free energy-harvesting future of ubiquitous*

computing. We tackle the technical hurdles of implementing a power failure resilient Python interpreter on low power and ultra-constrained embedded systems. In particular, we are concerned with making the tools for resilient, useful, in-the-wild computation to be build-able (make-able) by the average person. We try to integrate as closely as possible in the existing workflows of Python developers and hobbyist maker communities; for this reason, we adapt the most popular Python runtime for embedded systems—MicroPython [25] (or specifically its fork, CircuitPython [4], targeting ease of learning and use for hobbyist microcontroller users). Because CircuitPython is actively developed by Adafruit (one of the major providers of hardware to makers and hobbyists [98]), we start from the base CircuitPython runtime and implement numerous additions that enable CircuitPython to function effectively without a battery or tethered to a power outlet. These additions are invisible to the programmer; they do not change the existing CircuitPython workflow taught by Adafruit[2]. In our system, makers and hobbyists can develop untethered, battery-free computers that can do interesting tasks like read a humidity sensor, transmit information through LoRa radios, take a thermal image snapshot, or recognize human movements.

Contributions: We make various innovations across hardware and software to make this happen, rewriting and extending the CircuitPython interpreter (written in C) to checkpoint and restore state automatically such that Python programs can run despite frequent power failures on a very constrained computing device (an ARM Cortex-M0 [10]). Reworking the initialization of CircuitPython to enable fast reboots, supporting general peripherals like SPI and I2C for interesting applications using sensors, and rewriting system libraries to support restarting. We build custom hardware based on Adafruit's CircuitPython device lineup that functions as a shield and allows for a plug-and-play way to add energy harvesting and battery-free operation to the standard CircuitPython microcontroller board [3]. The shield and new runtime work together to ensure power failure resilient operation—enabling, for the first time, battery-free computation for hobbyists, makers, and early-stage embedded programmers.

In summary, the specific contributions of this paper are:

a) Introducing power failure resilience to an embedded Python runtime for ARM microcontrollers;
b) The integration of re-initialization of active peripherals;
c) The design and development of a hardware module that enables off-the-shelf maker platforms to be used for battery-free development and deployment;
d) Integrating both of these into an existing workflow supported by the hobbyist and makers community.

BFree is the first general-purpose platform for battery-free, energy harvesting devices, that runs Python. We release all code and hardware designs as open source to the makes and hobbyists community via [96].

2 DEVELOPING A BATTERY-FREE INTERNET OF THINGS

Increasingly, battery-powered computing's negative impacts have become more apparent [35, 48, 103]. This is worrying as the future of computing, following current trends, is one where likely trillions of computational devices [50, 87] will exist invisibly and visibly around us. If the future is wearable, implantable, and battery-powered, this future has trillions of dead batteries in it, with the associated ecological costs. For this reason, ubiquitous computing must leave the batteries behind (or at least our traditional notion of what a battery is) and harvest energy from the ambient to survive. These new devices must last for decades without maintenance and be resilient to power failures and changing conditions. Unfortunately, building systems like this is a complex task, not easily done by a *hobbyist*[3], and not lending itself to the maker movement [69, 84]. If these devices are the future, and everyone will be a programmer, then these devices must be made accessible to the novice, hobbyist, and maker movement.

[2]The authors are not affiliated with, or funded by, Adafruit.
[3]By *hobbyist* we refer to people that have at least minimum experience in any of classical (popular) programming languages such as C, Java or Python; not hobbyist in a sense of being exposed only to extremely simplified (almost kids-accessible) languages such as Scratch [68].

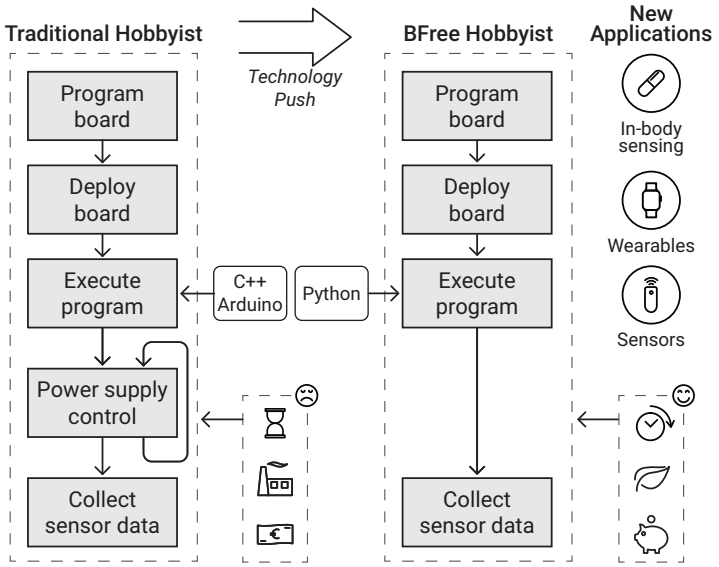


Fig. 3. The vision of computing we hope to enable by allowing hobbyists to build and then deploy battery-free devices easily and quickly for years: (i) reducing time in development (*easy-to-use Python versus C/C++*), (ii) reducing carbon footprint and saving money (*elimination of batteries*) and (iii) making program execution understandable despite frequent power failures (*guaranteed forward progress despite interrupts*).

2.1 Maker Platforms

The first steps towards this vision were enabled by platforms like Arduino [8], MicroPython [25], and CircuitPython [4]. CircuitPython is a Python interpreter written in C that can run on a microcontroller like an ARM Cortex-M0 [10]. Python programs are written on the desktop computer, sent over USB to the microcontroller (MCU), compiled into bytecode on the MCU, then executed. CircuitPython wraps common hardware functions like digital I/O, analog, and serial protocols into libraries that are accessible to the programmer. Other libraries that support things like cameras and radios can be created with Python. CircuitPython has a large community of users, hundreds of libraries for sensors and radios, stable releases, and a large number of hardware devices to use. We chose to build off CircuitPython because the Python programming language has seen a surge in popularity [14, 72], especially with low skill or hobby programmers, and to maintain access to this motivated existing user base. CircuitPython currently *cannot* support battery-free energy harvesting applications because of a multitude of reasons centered around the frequent power failures caused by requiring the device to live off energy harvesting only and not have a battery.

2.2 Challenges of Battery-free Programming

When a battery is removed from a microcontroller such as an Arduino Uno [9] or Adafruit Metro M0 [3], and the device opportunistically uses energy harvested from the ambient (like solar radiation), power failures become the norm. Once started, the device begins a race to get things done before a power failure. Once the power failure occurs, all the device's volatile state is lost (for example, memory content or register values). Then the process begins again.

Without some type of system-level handling of these power failures, programs could easily get stuck repeating old tasks and never completing all the tasks required (see again Figure 2), never getting to the end of the program. Recent work has explored instrumenting C programs with checkpoints [58, 82] or partitioning C programs into task graphs [60, 102] to enable easier checkpoint and restore cycles so that forward progress can be maintained. These techniques are useful, but only to the expert. In this work, we rethink the problems in battery-free, intermittent computing in terms of an interpreted and easy to use language—Python—aiming at the vision depicted in Figure 3. Unlike C/C++, which is compiled directly to machine code with each instruction executed by the CPU, interpreted languages have a runtime system that interprets the bytecode of the language into machine code. Allowing the runtime to handle high-level programming models and concepts, and even compile run code on the fly without compilation. Our choice to focus on an interpreted language instead of already very well explored C, stems not only from the fact that Python is probably the most popular programming language at the time of writing this article [13], but also due to its code compactness, fast extensibility, easier comprehension, and its simple and forgiving syntax. Quoting from a study published in 2000 [80]: "Designing and writing the program in Perl, *Python*, Rexx, or Tcl takes no more than half as much time as writing it in *C, C++*, or Java-and the resulting program is only half as long." Naturally, both C and Python have features that are represented better in one of these languages [63], but the fact that Python has *not yet* been used in the context of intermittent execution called for action.

Unique challenges come from trying to build and program battery-free and energy harvesting "things" that were addressed for the C language, and must now be explored and overcome in any interpreted language, such as the Python runtime in this work.

Power Failures: Energy needs usually outweigh the energy availability, meaning that even with consistent energy harvesting, power failures are the norm as the supply becomes depleted. Programs can be interrupted mid-execution at any code line, which damages program consistency and frustrates the programmer.

Long Reboot Time: Initializing a system, especially a sophisticated runtime, takes time and energy. Any upfront energy cost for rebooting takes away from valuable user application time and, in some cases, can cause a power failure before the completed reboot. This long reboot happens because the expectation is that these systems will almost always have continuous power (via a battery or USB plug) and rarely, if ever, need to reboot, so the reboot does not need to be optimized for speed.

Peripherals: Interfaces such as SPI and I2C have their own state stored in volatile registers. These interfaces connect to external peripherals (like a temperature sensor or a radio), which also have a volatile state. Both the interface and the peripheral will lose their configured settings when a power failure occurs. Developers have to explicitly handle the loss of the state of both the interface and the peripheral.

This list of challenges is not exhaustive (please refer to a broad discussion of these challenges in [57]). Indeed the area of intermittent computing for performance-driven embedded systems is an active research area. However, the fundamental limits listed above must be overcome once again, rethought, and re-imagined to enable an interpreted and easy to use language for hobbyist programmers.

2.3 State-of-the-Art in *Programming* for Batteryless Systems

With the multitude of challenges for batteryless development, two prevailing programming models have arisen, *task*- and *checkpoint*-based, both focusing on the C language, which we compare and contrast here. Figure 4 shows a visual comparison of these two programming models against Python, and each model's pros and cons.

Like in any area, the best programming language to use depends on the context, user skill, and available tools. Task-graph-based models like Mayfly [35], Alpaca [60], and InK [102], require a list of tasks that are strung together in a task-graph. The graph specifies the order and branching/control of execution. Tasks themselves must be atomic and idempotent (i.e., have no side effect, so they must bring up and tear down peripherals like a radio if

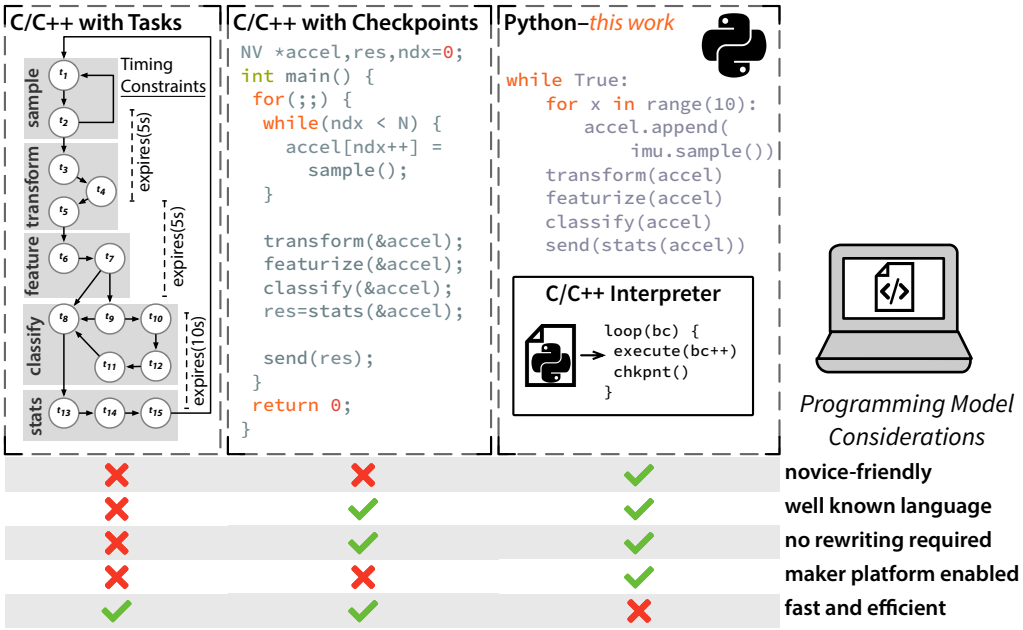


Fig. 4. Current and future programming models for batteryless systems. Two left-most dashed rectangles shown the state-of-the-art, expert-focused programming models using either *task graphs* where each task is a focused C/C++ fragment, to define a program (as in InK [102]), or automatically (with some caveats) checkpointing a C/C++ program (as in TICS [44]). These models run bare-metal, as in the CPU directly executes the machine code. A right-most dashed rectangle shows a novice-focused new model for batteryless systems programming, using Python. While Python requires an interpreter which executes bytecode, and is therefore much much slower when executing on an MCU compared to other approaches, the benefit of a familiar, forgiving (no C-pointers) programming model combined with automatic handling of power failures is far more important.

used in the task). The runtime system that executes tasks commits the results after task completion to non-volatile memory, preserving forward progress. Most task-graph-based languages allow annotations on the edges to define time constraints and data handling. Together these task-based programming models are by far the fastest and most energy-efficient methods for intermittent computing, but they require significant attention and expert rewriting to implement correctly.

The other major programming model is to take regular C code and automatically instrument it with checkpoints at compile time [44, 61, 97] or runtime [12]. The key research challenge here is to reduce the size of the checkpoint, such that the execution time becomes feasible and usable. C/C++ is not known for being novice-friendly, with its bare metal execution, use of pointers, and requirement (especially in the embedded context) for the programmer to have in-depth familiarity with computer organization concepts like memory, addressing, and types, to be useful.

The third option, newly presented by this work, is to use Python, as shown in Figure 4, which compares all three programming models. This option is radically different as it suggests an entirely new way to write software for batteryless systems, using a programming paradigm (interpreted)—*never used on batteryless devices before*. Python does not require the programmer to port existing code to a niche programming model like InK (which is task-based) and does not require developers to know C/C++ and have intimate knowledge of computer organization—a non-starter for many programmers with non-traditional introductions to computing. Python is also directly in line with

exciting, novice focused maker platforms that already have large communities and hardware. Notably, an interpreted language like Python will be slower, as an intermediate step exists between the Python code and the machine code, and is likely the primary reason up untill now Python was not used on batteryless, intermittently-powered devices. However, we posit that the flexibility and ease of using the language are worth that performance hit in many contexts. Finally, Python (and, more specifically, the interpreter) provides a ready mechanism for seamless—and invisibly to the programmer—checkpointing for power failure resilience.

2.4 State-of-the-Art in *Building* for Batteryless Systems

Researchers in intermittent computing have turned to novel hardware to make programming and building batteryless devices easier. C-based task graphs and checkpointed programming models are used on a multitude of hardware that all have one common characteristic, TI MSP430 FR series microcontrollers [94], which have built in FRAM, a byte addressable, non-volatile memory, that makes checkpointing state quick and cheap. Platforms like WISP [86], Flicker [31], Capybara [17] or Botoks [18] all use MSP430s. The key problem with these platforms is they are research platforms built for other researchers, not for novice developers or makers trying to build fun applications and learn things. Using a platform like WISP, an active RFID device, requires access to expensive RFID readers and custom programming modules. Flicker, Capybara, and Botoks all were built to explore specific hardware/circuit concepts in intermittent computing, including federating energy storage to reduce power failures (Flicker, Capybara), enabling rapid prototyping (Flicker), and using RC circuits for robust timekeeping across power failures (Botoks). Flicker is most closely related, claiming to enable novices to rapidly prototype. However, Flicker is a hardware platform *only* and does not enable a new programming model for batteryless systems, but supports existing ones. This is at once the best part, and great flaw of Flicker, as the programming models supported are all C-based, and as we discussed in Section 2.3, these expert level programming models are not sufficient for many novice programmers. This can be seen as Flicker was not readily adopted by the novice community, even now.

Finally, and most importantly, all of these platforms are highly constrained, comprising less than 256 KB of code memory, and 4 KB of SRAM for scratch space. Modern ARM Cortex M microcontrollers, such as [10], have up to 32 times the scratch space, and four times more memory, with processors ten times faster or more. The resource constraints of the TI MSP430 microcontroller preclude running a sophisticated interpreter with large memory footprint like Python (or JavaScript). Moreover, these platforms are bespoke, research prototypes, which are not part of any existing large community or ecosystem. To be truly effective and broadly adopted, a future platform targeting makers and novices must integrate closely with common hardware, like Adafruit's Metro M0 [3] boards.

3 BFREE SYSTEM DESIGN

We have developed BFree (shown in Figure 5) for novice developers, makers, or prototypers who want to program battery-free and energy harvesting Internet of Things applications easily with Python. To enable this vision, BFree's goals are:

(1) build a power failure resilient version of the Python runtime that can execute arbitrary Python programs;
(2) design hobbyist-usable hardware that can harvest ambient energy;
(3) enable a rich set of built-in functionality with sensors and libraries, and
(4) integrate (1–3) into a complete platform focused on entire stack (software and hardware) usability that can be used with common hobbyist platforms.

A key idea to enable the above goal number (4) is to build on and modify CircuitPython, and leverage the hardware ecosystem surrounding it. With an active user base and support by the maker-oriented company [3], this integration, though more difficult than developing a custom solution as in previous work [17, 31, 102], will finally enable the adoption of battery-free computing with BFree. Instead of designing everything from scratch, we seek to build on the existing maker and hobbyist electronics communities' momentum and enable them to go *batteryless*.

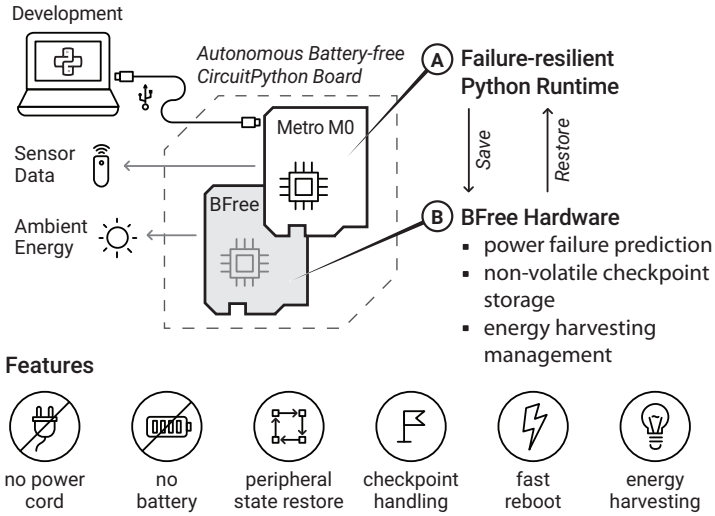


Fig. 5. BFree system overview. Programmers develop Python applications on their PC and upload them to the Adafruit Metro M0 board so that their application is interpreted by a power failure-resilient Python interpreter. The necessary power-failure management is performed via communicating with the BFree hardware that harvests energy, handles checkpoints, keeps track of peripherals, and ensures fast reboot.

3.1 Executing Python Code

As shown in Figure 5, BFree is split across the *BFree runtime* and *BFree hardware shield*, which sits on top of the Adafruit Metro M0 board (see again Figure 1). Python programs can be executed on the Adafruit Metro M0, a simple Arduino-style breakout board that has power circuitry, USB, LEDs, and pins broken out for easy prototyping, build around an ARM Cortex-M0 microcontroller [10]. Novice developers write Python code on their laptop, send it to the Metro M0 over USB, where it is then compiled and executed.

3.2 Energy Harvesting

The Metro M0 board is not equipped with energy harvesting circuitry for battery-free operation, nor does it have any fast and durable non-volatile memory (it only has FLASH—slow, energy-intensive, and not very durable). BFree hardware provides access to the energy available in the ambient; solar, RF, kinetic, or any other type (solar being a default one). The BFree hardware sits on top of the Adafruit Metro M0 board while still exposing pins for prototyping. Energy is harvested, stored in a capacitor, then made available to the Metro M0 board to execute the Python program embedded in the core. When the Metro M0 and BFree shield is disconnected from power (battery, USB, or wall socket) the energy harvesting circuitry takes over, enabling battery-free and untethered operation.

3.3 Checkpoints and Progress

The BFree shield and our Python runtime are co-designed to enable resilience to power failures. Before a power failure, the progress of the Python program is checkpointed. The checkpoint operation takes all the volatile memory and data stored on the Adafruit Metro M0 that is required to resume execution and saves it to the fast non-volatile storage (FRAM) on the BFree shield. When enough energy is stored in the BFree shield for the whole device to turn on, the Metro M0 turns on, downloads and restores the checkpoint of past progress from the BFree shield, and

then resumes executing the Python program from where it left off. Doing so keeps the program from wasting cycles re-executing old code, keeps memory and progress consistent, and makes it easier for the novice programmer since they do not have to figure out what to do in the face of power failures. These checkpoints are carefully managed between the BFree shield and Metro M0 so that the Python code can be safely and consistently executed despite power failures.

3.4 Libraries and Sensing

As shown in Figure 5, programmers can attach sensors via a breadboard and then import built-in or third-party libraries to use with their Python program. Some built-in libraries provide access to hardware functions on the Metro M0 and are heavily used by MicroPython/CircuitPython programmers, for example, the `time`, `digitalio`, and `busio` libraries. These enable time delays and measurements, usage of the digital pins, and interaction with I2C and similar communication protocols (which enable use of external 'breadboarded' sensors), respectively. The BFree runtime provides modified versions of these libraries, building in features to enable intermittent computation, reducing surprises for the novice programmer who wants to move to battery-free operation. For example, the I2C and digital pin state are saved in a checkpoint so that on reboot the correct pin direction and output, as well as I2C configuration is restored. These system design points together enable a broad range of built-in functionality and external sensors and peripherals.

3.5 Deployment

Programmers bring this all together to deploy real-world applications with BFree. Once programmed through the laptop or desktop, the Metro M0 equipped with the BFree shield is disconnected and deployed in the wild, harvesting energy and performing computation and sensing despite power failures.

4 IMPLEMENTATION

We now describe the implementation challenges and details stemming from the system requirements necessary to port a significant (CircuitPython) codebase to intermittent operation. Our core implementation requirement is to enable power failure resilient operation for interpreted Python programs over bare metal programming environments such as C/C++. To achieve this (i) the progress of computation and memory consistency against power failures should be ensured; (ii) the system reboot procedure should be optimized so that the system recovers from power failures faster; (iii) peripheral state should be restored so that peripheral interaction continues from a consistent state; and (iv) system level libraries `time`, `digitalio`, and `busio` must be adapted for persistent and reliable operation. The BFree additions to the CircuitPython interpreter include over 5500 lines of low-level code, written in C and assembly, split across the ARM Cortex-M MCU on the Metro, and the MSP430FR MCU on the BFree shield. This represents a substantial addition to the core codebase of CircuitPython to enable intermittent operation and checkpointing.

4.1 Hardware

We designed and built the BFree shield capable of energy harvesting, power-failure detection, managing checkpoints in non-volatile memory, and keeping track of time. This shield sits on top of the Adafruit Metro M0 board—shown in Figure 6. The details of the hardware design are as follows.

Energy Harvesting Circuitry: The energy harvesting circuitry on the BFree shield is the source of energy for all the components on both the BFree shield as well as the Adafruit Metro M0 board during battery-free operation. The BFree shield can be equipped with any energy harvesting source, e.g., a solar panel through a connector to harvest ambient energy. The energy harvesting circuitry accumulates the harvested energy from this energy harvesting

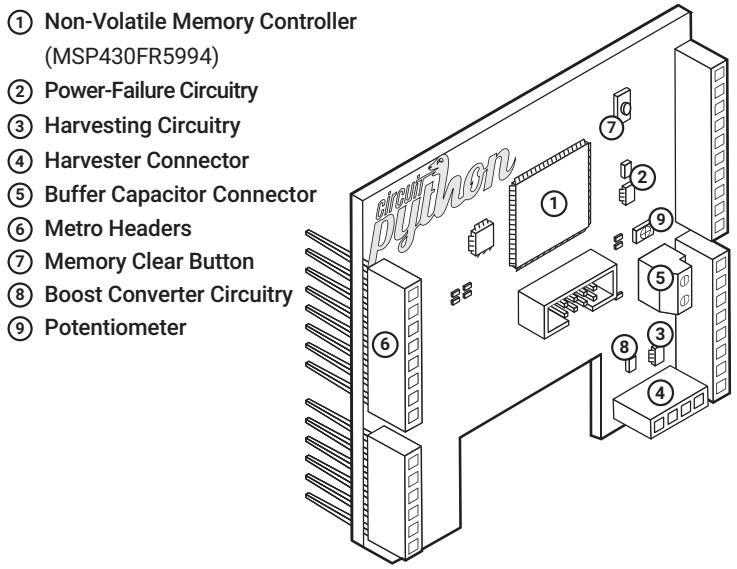


Fig. 6. BFree hardware with annotated components and functions. The non-volatile memory controller captures and logs checkpoint data. Power-failure circuitry allows for a configurable power-failure signal when the energy in the storage capacitor is running low. Harvesting circuitry accumulates the harvested energy from the harvester connector into the buffer capacitor and provides hysteresis control. The memory-clear button provides the user with an easy way to restart the application (i.e., delete the checkpoints). Headers connect the shield to the Adafruit Metro M-1 board.

source into a capacitor, which is user-selectable[4]. A hysteresis control (via MIC841 voltage comparator [66]) is implemented to enable operation when the stored energy in this capacitor is above a predefined threshold.

Power Failure Prediction: The BFree shield is equipped with a user-configurable voltage comparator—using a potentiometer and a nanopower comparator (TI TLV3691 [93])—that is used to signal the BFree runtime that the storage capacitor voltage, and therefore the remaining energy, is running low. This threshold voltage is made configurable because the ideal setting highly depends on the capacitor's discharge speed. In turn, this is a relation between the current draw of the system, the incoming energy from the harvester, and the size of the capacitor. As BFree aims to support many different applications with wildly different requirements, this needs to be configurable to fit the applications' needs. This power failure prediction signal can optionally be used by the BFree runtime to change the checkpoint scheduling, reducing the checkpointing overhead when the system's remaining energy is not critically low.

Checkpoint Storage: The BFree shield is composed of non-volatile memory (FRAM) hosted by a Texas Instruments MSP430FR5994 microcontroller [94] that has 256 KB FRAM and 4 KB SRAM. The software on this microcontroller implements a map-based file system to store the checkpoints and takes care of the required double-buffering to keep checkpoints from becoming corrupted. The microcontroller on the Adafruit Metro M0 board triggers a checkpoint operation by sending commands to the BFree shield over an SPI bus. The MSP-based FRAM MCU was chosen to alleviate the BFree runtime from the checkpoint management burden. However, it

[4]We will report the exact values of both capacitor and the type of energy harvester while discussing results assessing the impact of energy trace on BFree program execution in Section 6.3.

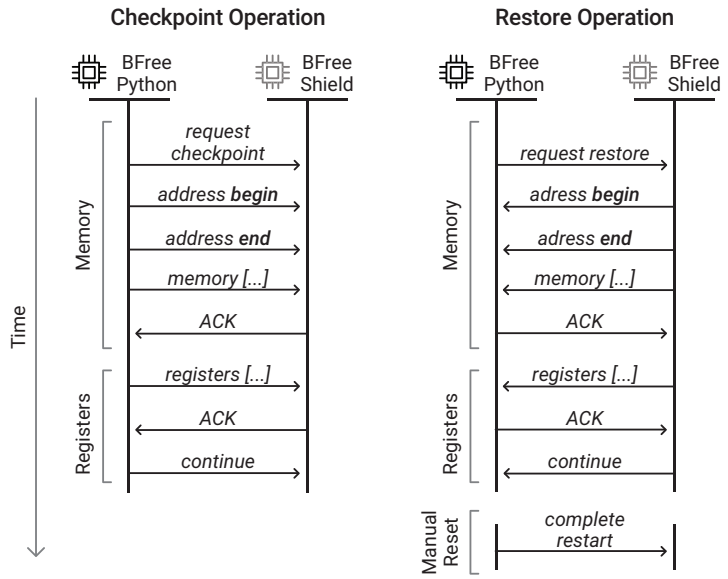


Fig. 7. Checkpoints ensure the progress of computation and memory consistency. The BFree Python interpreter sends the checkpoint data (volatile memory contents and register values) over the SPI bus to the BFree shield so that this data is stored in non-volatile memory. Upon recovery from a power failure, the stored checkpoint data is used to restore the state of computation and the memory contents.

would be easy to equip the system with a memory-only module and modify the BFree runtime to perform the data management (e.g., double-buffering), provided that the Adafruit Metro M0 board would have on-board FRAM.

4.2 Software

We modified the CircuitPython runtime interpreter so that the necessary operations to ensure the progress of computation and memory consistency are taken.

Checkpoints capture the volatile state of the ARM Cortex-M0 microcontroller on the Metro M0 board that executes the Python interpreter. The volatile state of the ARM Cortex-M0 includes the contents of the volatile main memory, where the global and local variables are stored, as well as the contents of the registers and the state of any initialised interface such as I2C. Capturing this state before a power failure, then restoring this state on reboot ensures the progress of computation. Computation continues from where it left just after the recovery from a power failure. However, not all state has to be restored when the system restores a checkpoint. Some states such as USB and the file system should be reinitialized from scratch. The CircuitPython codebase has been explored and divided into two parts. One part that can be safely restored and one part that requires re-initialization when the system attempts a reboot. We instrumented the Python interpreter with *potential* (i.e., not forced) checkpoint locations. Most notably, we added them to the main interpretation loop. The locations are potential because whether a checkpoint is performed at these locations is decided by the current checkpoint strategy (e.g., by the period-based strategy). An alternative is to trigger checkpoints via power-failure prediction notification (which is possible with BFree hardware).

Checkpoint Content: The checkpoint contains all the necessary information for the system to continue where it left off. As CircuitPython itself is written in C, this contains: (i) the registers, (ii) global variables, (iii) the stack, and (iv) any dynamically allocated memory. The C programming language allows for custom memory schemes as all the memory can be accessed freely through pointer manipulation. A custom memory allocator is, therefore, not uncommon within embedded software development. This is also the case in CircuitPython by means of a garbage collector. The garbage collection sub-system occupies all the remaining memory in the system after the global variables and the stack are reserved, the size of which is determined within the linker script during compilation. The garbage collector is also responsible for all the dynamically allocated memory in the system, and all the Python interpreter specific stacks—as CircuitPython is a stack-based interpreter. Our checkpointing implementation allows for the dynamic specification of memory regions that are required to be checkpointed.

Excising C frameworks that support intermittent execution, such as [102], expect that the whole system needs to be restored. In real-world applications, such as CircuitPython, this does not hold. There are parts of the system that need to be reinitialized every reboot. In CircuitPython, this mainly consists of the USB and flash file system-related data and variables. Excluding this data from the checkpoint is not only to reduce the checkpoint size—and therefore the checkpoint time—but also having these values at anything other than zero can (and will) cause bugs during the initialization of these subsystems. We inspected the CircuitPython code base and selected all global variables that require to be checkpointed. These are put into a special configuration file, and the compiler will place these special memory regions that are excluded from the checkpoint.

Checkpoint Operation: The left-hand side of Figure 7 depicts the steps taken during the checkpoint save and restore operations. The checkpoint data is sent over the SPI bus from the ARM Cortex-M0 microcontroller on the Metro M0 board to the MSP430FR5994 microcontroller on the BFree shield. The checkpoint includes the start and end addresses of the volatile memory region on the ARM Cortex-M0 microcontroller and the contents of this memory space. This step is repeated for the different memory sections in CircuitPython. Moreover, the contents of the general-purpose and special registers should also be saved within the checkpoint context. Special attention is given to the registers, as these need to be checkpointed last, and their content must not be altered during the checkpoint procedure. Otherwise the state of the program will be corrupted when a restore is performed. To guarantee this, the register checkpointing is written primarily in assembly language.

All this information is sent over SPI and stored in non-volatile FRAM of the MSP430FR5994 on the BFree shield. It is worth mentioning that the checkpointed data is stored in a double-buffered memory region in FRAM to ensure memory consistency: the checkpoint is stored in a temporary memory region where the original memory region holds the data of the previous checkpoint taken successfully. After the checkpoint data is entirely saved in the temporary buffer, an atomic variable is modified to swap the temporary and original buffers, thereby committing the checkpoint. If the preexisting checkpoint would be directly overwritten, a power failure during a checkpoint would lead to a corrupted state and would therefore require a complete restart of the application.

Restore Operation: Figure 7 also depicts the restore operation. After sufficient energy is harvested and the system reboots to start operating again, the latest successful checkpoint needs to be restored so that the computation continues from where it left. For checkpoint recovery, the Metro M0 board communicates with the MSP430FR5994 on the BFree board over SPI. It reads the contents of the checkpointed memory regions and the values of the registers and peripherals. The recovery is completed by jumping to the next instruction to be executed. The successful restoration of the volatile state captured in the checkpoint ensures the progress of computation and memory consistency. To enable a manual hard reset—restarting the application from the beginning—an additional reset button is provided on the BFree board. Pressing this button while powering up the system will delete the existing checkpoint. Additionally, this can be achieved by uploading a new Python script to the Metro M0 or by restarting the current Python script using the standard Python Read-Evaluate-Print-Loop (REPL) language shell made available over the serial interface.

Reducing System Restart Burden: Unmodified CircuitPython takes an entire second to boot, which wastes a significant amount of energy. The Metro M0 board has a bootloader to easily update the CircuitPython binary. This increases the bootup time because some time is reserved for the user to notify the system to mount it for a CircuitPython update. Additionally, approximately 700 ms is introduced to let the user enter a so-called safe mode, shown in Figure 16. We remove this delay and optimize other delays during the reboot in our modification of the runtime (in-depth discussion of this matter is provided in Section 6.5). We do not alter the bootloader code, as we want hobbyists to be able to freely change their CircuitPython image to our intermittent version (BFree) and back without the need for an external programmer.

Peripherals and Libraries: For the proper operation of battery-free hardware platforms, the states of peripherals, including digital pins, I2C, and time, should be restored after a power failure. As an example, during a particular I2C communication between the microcontroller and a sensor (i) first the dedicated I/O ports of the microcontroller are configured for I2C operation, (ii) then the sensor is configured for the desired operation, (iii) and finally the command for the desired operation is sent, e.g., sampling or actuation. Upon power failure, if the command is sent without re-configuring the I/O ports of the microcontroller and/or sensor, the sensor sampling will not work correctly.

In order to require minimal changes to existing CircuitPython applications that make peripherals and external sensors, a balance was struck between automated restoration and programmer aided restoration. The states of the peripherals are automatically restored during the restore operation. However, the initialization of different sensors can wildly differ on a case by case basis. Therefore any re-initialization required by the sensor must be performed by the programmer. In traditional CircuitPython libraries, the split between peripheral and sensor initialization is already present, therefore it is often only needed to relocate the sensor initialization library calls to just before the reading. The effects and impact of this trade-off will be further showcased in Section 5.

Checkpoint Strategies: We designed three checkpoint scheduling strategies that use the information provided by the power-failure prediction signal from the BFree shield (see the description of power failure prediction in Section 4.1). These strategies, named (i) *periodic*, (ii) *trigger*, and (iii) *hybrid*, have different overhead since they generate a different number of checkpoints during program execution. The periodic strategy generates a checkpoint of the system every p milliseconds, without considering the power-failure signal. Alternatively, the trigger strategy generates a checkpoint every p milliseconds *only* when the power-failure signal is active. The hybrid strategy is a combination of the other two strategies: it generates a checkpoint either every p milliseconds when the energy level is high (i.e., the power-failure signal is not active) or every q milliseconds when the power-failure signal is active (i.e., low energy operation).

Among the checkpoint strategies, the trigger strategy eliminates all checkpoints when the energy level is not low, a desirable property. However, when the power-failure signal active, this strategy requires enough energy (i.e., active-time) to perform a checkpoint—which might not be guaranteed due to varying energy harvesting conditions and different capacitor sizes. Therefore, if the storage capacitor discharges faster than checkpoint generation, periodic checkpoints are ideal. Under varying energy harvesting conditions (e.g., solar energy harvesting frequently distracted by clouds), the hybrid strategy can perform the best. In our system, the programmer can switch between these checkpoint strategies as well as change their parameters (i.e., p) at runtime using a builtin Python library. This enables dynamic and configurable checkpointing with respect to the characteristics of the energy harvesting environment during the application's execution.

5 BFREE DEPLOYMENT AND USE CASES

The key question for BFree is how useful it is for building battery-free applications. In this section, we engage in proof by demonstration, showing a range of battery-free applications that BFree enables for novice programmers. We build *useful*, sustainable, and battery-free applications around BFree with *unmodified* CircuitPython. Figure 8

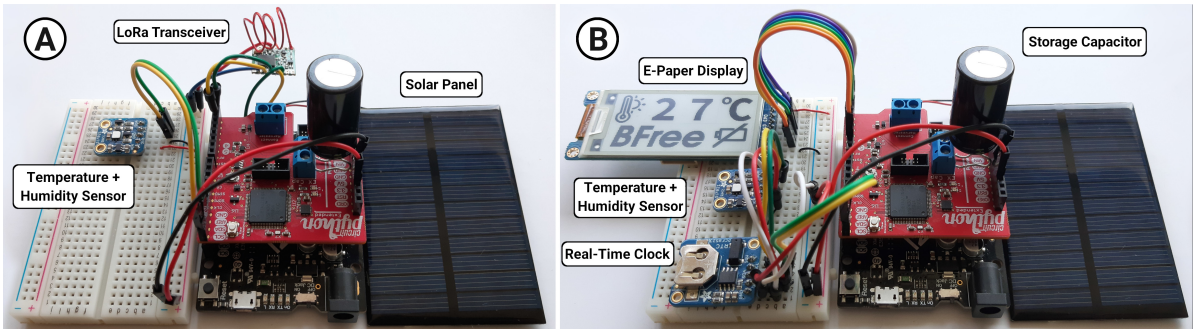


Fig. 8. Assembled hardware to demonstrate two hobbyist-grade applications written in unmodified CircuitPython running on intermittent energy with BFree: **(A)** LoRa sensor mote, and **(B)** electronic paper temperature display.

shows hardware prototypes of two example applications, (i) LoRa[5] sensor mote and (ii) electronic paper temperature display. For each application, we design an experimental plan to explore how battery-free operation affects design and deployment. We proceed with discussing each prototype in detail.

5.1 BFree LoRa Sensor Mote

The classic use case for embedded systems is to measure environmental factors long term, with seminal examples deploying 'motes' like the TelosB [77] for applications including volcano monitoring [100], habitat, and wildlife monitoring [62], wildfire detection [28], and precision agriculture [46], among many others. BFree enables these types of environmental monitoring applications programmed in Python, without relying on batteries. All of these applications have similar functions typical of an edge computing system; they opportunistically measure some aspect of the environment, process and summarize the collected data, and (when a specific condition is met) share this information wirelessly. We prototyped an environmental monitoring system hardware on a breadboard (as is typical of a hobbyist) and programmed using Python software. The sensor senses temperature and humidity, averages multiple readings, and then sends that data wirelessly to a base station. We used an unmodified Adafruit SI7021 [1] breakout board, which continuously measures temperature and humidity and sends the measured data to the Adafruit Metro M0 board for further processing through the I2C bus. When a predefined number of samples are collected, BFree averages them and broadcast this value using an Adafruit RFM95W [5] LoRa radio transceiver breakout board connected to the Adafruit Metro M0 board via the SPI port. To receive and verify the messages broadcast by BFree, a dedicated LoRa messages collector operating on constant power that continuously listens for LoRa packets has also been built using a second Metro M0 board running vanilla CircuitPython.

For both the SI7021 sensor and the RFM95W LoRa module, we used the *unaltered* Python libraries provided by Adafruit. The application Python code we altered only by moving the LoRa initialization to the transmission code, and disabling checkpoints before the LoRa initialization and transmission and enabling them after, which is provided in an API for BFree programmers. Doing so is required as the LoRa module has an internal state that needs to be configured at boot, which will be lost after every power failure. The SI7021 has no internal state and is therefore automatically handled by the BFree runtime. So *only two* additional single-line statements from the programmer are required to fully unplug the USB cable, leave the battery behind, and survive off energy harvesting only. With BFree, programming in Python, and using hobbyist electronics, this relatively complex application is easily transformed into a battery-free system resilient to power failures.

[5]LoRa is a low power, wide area, low data rate network protocol. It is increasingly common in distributed sensor networks because of unlicensed operation and very long communication ranges. More details on LoRa can be found in many academic surveys, e.g., [79].

Table 1. Duty cycle to on/off relation.

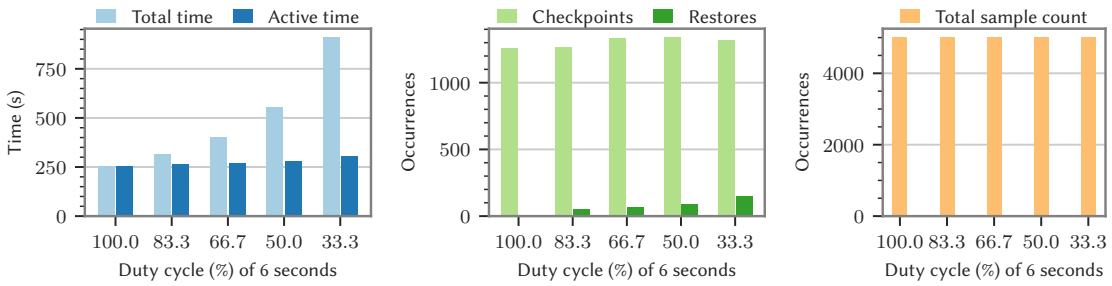
Duty cycle (%)	**On time** (s)	**Off time** (s)
100	∞	0 (i.e., continuous power)
83.3	5	1
66.6	4	2
50	3	3
33.3	2	4

Experimental Setup: To demonstrate that BFree works as expected, we run a series of benchmarking tests on the LoRa application that exercises the checkpointing and restore functionality of a complex, peripheral enabled application. We limited the number of LoRa packet broadcasts to 50, and the number of samples collected (both the temperature and the humidity) before a broadcast to 100. These numbers were chosen such that the benchmark on continuous power completes in approximately 250 seconds. The payload of each LoRa packet contains three main fields coded in plain text: (i) broadcast packed ID, and (i) average temperature measurement, and (iii) average humidity measurement (comma-separated two decimals per each of the two measurements). To make our evaluation repeatable and remove any potential energy harvesting variability, we connect a controlled square wave (low state of zero volts and high state of positive voltage) to the harvester input. This way, we can experiment with the on/off time relation (i.e., duty cycle) of the system in a controlled way and more accurately synchronize the start of the application with the start of an on period. Additionally, by varying only the duty cycle and not the time it takes to perform one on/off cycle we can directly compare the results from run to run. The only limitation we encountered using this controlled setup is the lack of power failure detection, as the square wave causes an immediate power fault, limiting us to investigate only the periodic checkpointing strategy. We configured the system with a checkpoint period of 200 ms, i.e., when the system is active every 200 ms a checkpoint is created. Additionally, we chose a period of six seconds and ran the application using five different duty cycles shown in Table 1. Note that an on-time of one second is too short to be feasible due to the limitations imposed by the setup, initialization time of the LoRa module, and broadcast time of the LoRa module.

Summary of Results: We experiment with different levels of intermittency, i.e., different levels of energy availability. As the duty cycle, we selected (see again Table 1) decreases from 100% (i.e., continuous power) down to 33.3%, the *total time* to complete the benchmark increases. However the *active time* (i.e., the time the system is on and executing code) remains somewhat constant, see Figure 9(a). The difference between the active times is the accumulation of all the reboot procedures, checkpoint restores, and re-execution of code executed before a power failure and after the last checkpoint. As shown in the same figure, this only accounts for a small amount of the active time. The total number of samples, as can be seen in Figure 9(c), stays constant throughout all the runs since our application performs 50 LoRa packet broadcasts and broadcasts data only after 100 samples are taken and averaged. The number of restorations in Figure 9(b) is equal to the number of off time occurrences throughout the run. The number of checkpoints in this configuration is approximately equal to the active time divided by the checkpoint period.

5.2 BFree Electronic Paper Display

The second demonstration system we developed is a BFree electronic paper display that updates a display output every n seconds with the average temperature since the last display refresh. These displays are becoming more common for electronic shelf labels in automated supermarkets and grocery stores, as they can automatically update values and pricing without human intervention. Of course, these systems suffer from the use of a battery. With

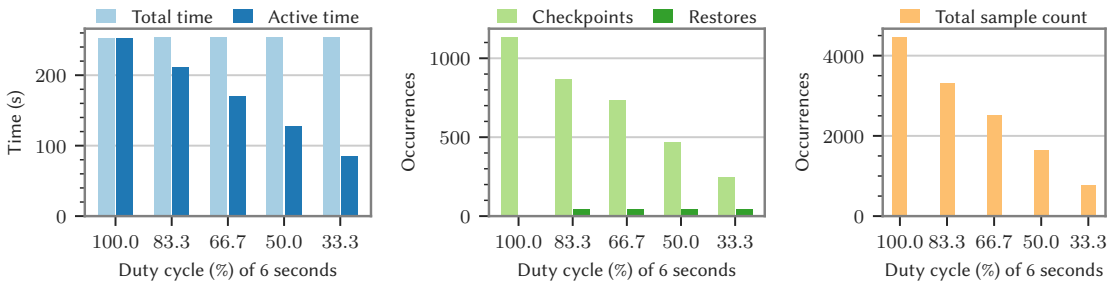


(a) LoRa Execution time (b) LoRa checkpoints and restores (c) LoRa total number of samples

Fig. 9. Evaluation of the LoRa sensor mote application. This application sends a total of 50 LoRa broadcast packets that carry the average temperature and humidity of 100 samples collected. The intermittency is varied using different duty cycles within 6 second period (see Table 1). The period checkpoint strategy with a checkpoint period of 200 ms is used. The results indicate that (a) as the off time increases, the total time increases as it takes longer to complete the benchmark application. The constant active time (the time the system is actually on) shows that the overhead caused by re-execution and restores is minimal; (b) the number of checkpoints stays constant due to the periodic checkpoint strategy; (c) as the benchmark is completed after 5000 samples are collected (50 times 100 samples), the total number of samples stays constant irrespective of varying duty cycles.

BFree, a shelf label can be energy harvesting and auto-updating. For this demonstration, we breadboard a BFree system that takes average temperature readings over time and displays it on e-paper at a set update rate. We use the same temperature sensor as used in the LoRa application, i.e., Adafruit SI7021 (see Section 5.1). For the display, we use an electronic paper (e-paper) display. E-paper displays are ideal for intermittent applications as they retain their display state even if there is no power. A limitation with e-paper displays is the low maximum refresh rate and (in some cases) the requirement to perform a time-consuming update cycle whenever a section of the display requires changes. We set to show the average temperature every ten seconds, so the maximum update rate is not an issue. On the other hand, to overcome the need to update the whole screen (which can take between two and ten seconds depending on the model of the display), we chose a Wemos Electronics 2.13 inch 250×122 e-paper display [99] that supports partial updates. Partial updates allow us to only write to a small section of the screen, reducing the update time to around 0.7 seconds. As there are no e-paper libraries for CircuitPython that allow for partial updates, we wrote a custom library in C as a demonstration that BFree also works for built-in libraries written in C. Additionally, the e-paper application demonstrates a different kind of intermittent application. For LoRa, the number of samples and broadcasts were fixed, so a certain active time is required to complete the benchmark. In contrast, for the e-paper application, we added a real-time clock (RTC) and fixed the refresh-period of the e-paper display. The RTC is an Adafruit PCF8523 module [2] and is used without modifying the provided Python library. Note that the RTC module supports a coin cell battery by default (which was used in the experiment), but any (super-)capacitor can also be used to keep BFree completely battery-free (for example 220 mF capacitor should keep the RTC running for at least a month assuming a startup charge of 3.3 V, see [71, Page 34], simple circuitry could additionally be used to charge the RTC capacitor from harvested energy).

Experimental Setup: We chose to refresh the e-paper screen every ten seconds. To complete the benchmark, a total of 25 screen refreshes must be performed. These numbers were chosen to (i) not damage the e-paper display by exceeding the refresh rate and (ii) limit the total benchmark time on continuous power to approximately



(a) E-paper Execution time (b) E-paper checkpoints and restores (c) E-paper total number of samples

Fig. 10. Evaluation of the e-paper application benchmark refreshing the screen a total of 25 times with the average temperature collected during a ten second period between refreshes. The intermittency is varied using different duty cycles of a six second period (see Table 1). The period checkpoint strategy is used with a checkpoint every 200 ms. The results indicate that (a) total time stays constant and the active time decreases (in contrast to Figure 9(a)), this is because of the time-sensitive nature of the application; (b) the number of checkpoints decrease when the duty cycle increases because it is directly tied to the active time when the period checkpoint strategy is used. The restores are constant for the same reason; (c) because the benchmark is completed after 250 seconds (25 times 10 seconds) the total number of samples decreases as the active time also decreases.

250 seconds. For the same reasons mentioned in Section 5.1, we use a square wave with different duty cycles (see Table 1) to power the board and a checkpoint period of 200 ms to evaluate the applications.

Summary of Results: In contrast to the LoRa application presented earlier, the total time of the e-paper application remains constant (see Figure 10(a)) as the duty cycle decreases from 100% down to 33.3%. This is due to the fact that we fixed the amount of time the benchmark takes (using the RTC and a fixed number of screen updates) instead of fixing the number of samples required to complete the benchmark. Consequently, the active time, the total number of samples (Figure 10(c)) and the number of checkpoints (Figure 10(b)) decrease. Since the number of times the system turns off during the benchmark is also constant—except for the case of continuous power—the number of restores is constant.

6 EVALUATION

The main design goals of BFree are to (i) match the runtime performance of CircuitPython on continuous energy, (ii) enable progress of computation during intermittent operation that relies only on harvested energy, (iii) introduce a minimal burden on baseline CircuitPython in terms of resources. In order to see if we meet these goals, we evaluate our BFree prototype implementation by comparing its performance and runtime overhead to unmodified baseline CircuitPython/MicroPython and regular C. In particular, we perform the following experimental evaluation:

(1) Running benchmark programs in order to measure the performance versus baseline, and verify the correctness of execution despite multiple power failures;
(2) Demonstrating the effects of using harvested energy and different checkpoint strategies;
(3) Measuring the cold boot (startup time) of the system;
(4) Profiling the resource requirements such as the main and non-volatile memory overhead;

Experimental Setup: For each experiment, we use an Adafruit Metro M0 with the BFree shield connected on top. We control power delivery in one of two ways: (i) via a microcontroller that gates power using a MOSFET from a Digilent power supply to the BFree device under test, allowing us to repeatably simulate intermittency rates, and (ii) via a lamp from which the BFree shield harvests light energy. We use a Keysight DSOX3014A oscilloscope and

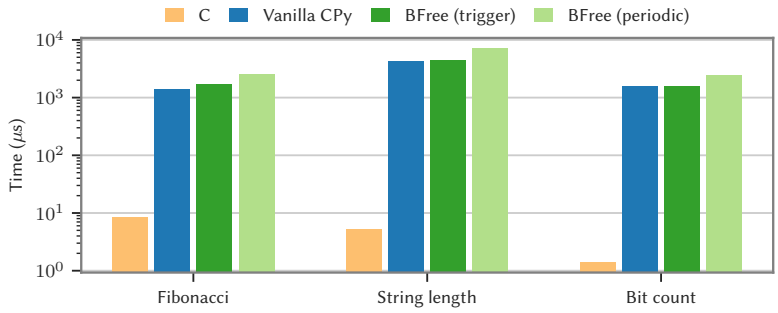


Fig. 11. Comparison of execution time of three applications: (i) generator of Fibonacci sequence for the value of 40 (denoted as *Fibonacci*), (ii) program that outputs length of a predefined string with a length of 40 characters (denoted as *String length*), and (iii) a program that counts the bits of the number 0x7FFFFFFF (denoted as *Bit count*) written with (i) vanilla C language, (ii) vanilla CircuitPython (denoted as *CPy*), and (iii) CircuitPython running under BFree runtime. Two cases of checkpoining were used: (i) *trigger*, with checkpoint triggered by an on-board BFree comparator and (ii) *periodic*, with checkpoint every 200 ms. Each application is run 10000 times and the result presented is the average of all the runs. Applications implemented in CircuitPython with and without BFree shield take up to thousand a times more time to complete than simple C implementations. On the other hand, BFree introduces small overhead over vanilla CircuitPython.

Saleae Logic Pro 8 Logic Analyzer to perform time and electrical power (P/I/V/E) measurements. When making comparisons, we use baseline CircuitPython firmware (version 4.1.0) running on the same experimental setup (i.e., software and hardware).

6.1 Comparing Execution Time: C, CircuitPython, BFree

An interpreted programming language, such as CircuitPython, will, in most cases, be slower (in terms of code execution duration) than a compiled language, such as C—a language of choice for professional development with embedded microcontrollers. Python (thus also CircuitPython) will trade-off execution speed for code extensibility, code clarity, and multi-purpose features. In this experiment, we measure the difference in execution time between interpreted CircuitPython and bare-metal C compiled to machine code. Additionally, we compare the execution time of BFree (as in our modified CircuitPython runtime coupled with the BFree shield) compared to vanilla (i.e., unmodified) CircuitPython.

Experiment: We measured CircuitPython's execution speed for selected three simple programs, (i) a Fibonacci sequence generator (for one specific value), (ii) a program that outputs the length of a predefined string, and (iii) a program that counts bits of a predefined bit sequence. Each of these programs was implemented in simple Python, and in C. All programs were written such that they did not rely on any internal or high-level functions (of either C or Python). All three programs (for C and for Python), which source code is available in [96], were executed on Adafruit Metro M0 board with and without BFree shield. The same programs were also run on two versions of BFree: (i) *trigger*, when BFree on-board comparator triggered a checkpoint (refer to Section 4.1 (**Power Failure Prediction**)) and (ii) *periodic* when checkpoint was triggered periodically every 200 ms, irrespective of supply voltage level. We compare all programs with continuous power to not confound execution time with delays from power failures and only compare the actual time computing.

Results: The result of the experiment is presented in Figure 11. Our results confirm our intuitive hypothesis that compiled C programs are faster than interpreted CircuitPython. In terms of actual values, compiled C demonstrates thousands of times faster execution than CircuitPython. On the other hand, we see that BFree gives only a small

overhead compared to vanilla CircuitPython, which is already widely used by the maker community, proving that the end user of BFree will not experience significant performance degradation compared to vanilla CircuitPython. Our final observation is that event-driven checkpointing reduces the overhead of the intermittent runtime, which is clearly seen comparing BFree (triggered) and BFree (periodic) for all three programs in Figure 11.

6.2 Benchmarking for Correctness and Power Failure Resilience

Next, we measured the execution time and correctness of the same benchmarks written in Python as the one used in the previous experiment (see Section 6.1) running on BFree. Each benchmark was executed on intermittent power of a varying duty cycle, the same way as done in Section 5.1, i.e., different duty cycles of a total period of six seconds, where a duty cycle of 100% equals constant power (see Table 1). Also similar to Section 5.1, a periodic checkpoint strategy was used with a period of 200 ms. The results of the evaluation are given in Figure 12.

The key takeaway from Figure 12 is that BFree allows these benchmark programs to complete and make progress despite intermittent power failures. This would not be possible for this task using normal CircuitPython, as the benchmark will never complete. The time it takes for the benchmark to complete, including the time the system is off due to a power failure, is denoted as the *total time*. The *active time* is the time spent executing code, i.e., the total time minus all the off-times/time during power failures. Additionally, the active time (denoted as dark blue in Figure 12) only slightly increases when the number of power failures increases, signifying that our boot and restore overhead is small. Because we used a periodic checkpoint strategy, the number of checkpoints remain constant, and the number of restores is equal to the number of power failures. If a trigger-based strategy were used, the number of checkpoints would grow with the number of restorations. This is because the number of checkpoints created during low energy operation is determined by the programmer through the potentiometer setscrew on the BFree shield, the incoming energy and the consumed energy (Section 14).

We verified the outcome of each benchmark for correctness (for example, by comparing the values generated by the Fibonacci function). Since the programs running intermittent power had multiple power failures before the final result was computed, the intermediate results and checkpointing must work for the correct result. By getting the correct result for each of the benchmarks, this demonstrates that the BFree method preserves program consistency and correctness through power failures.

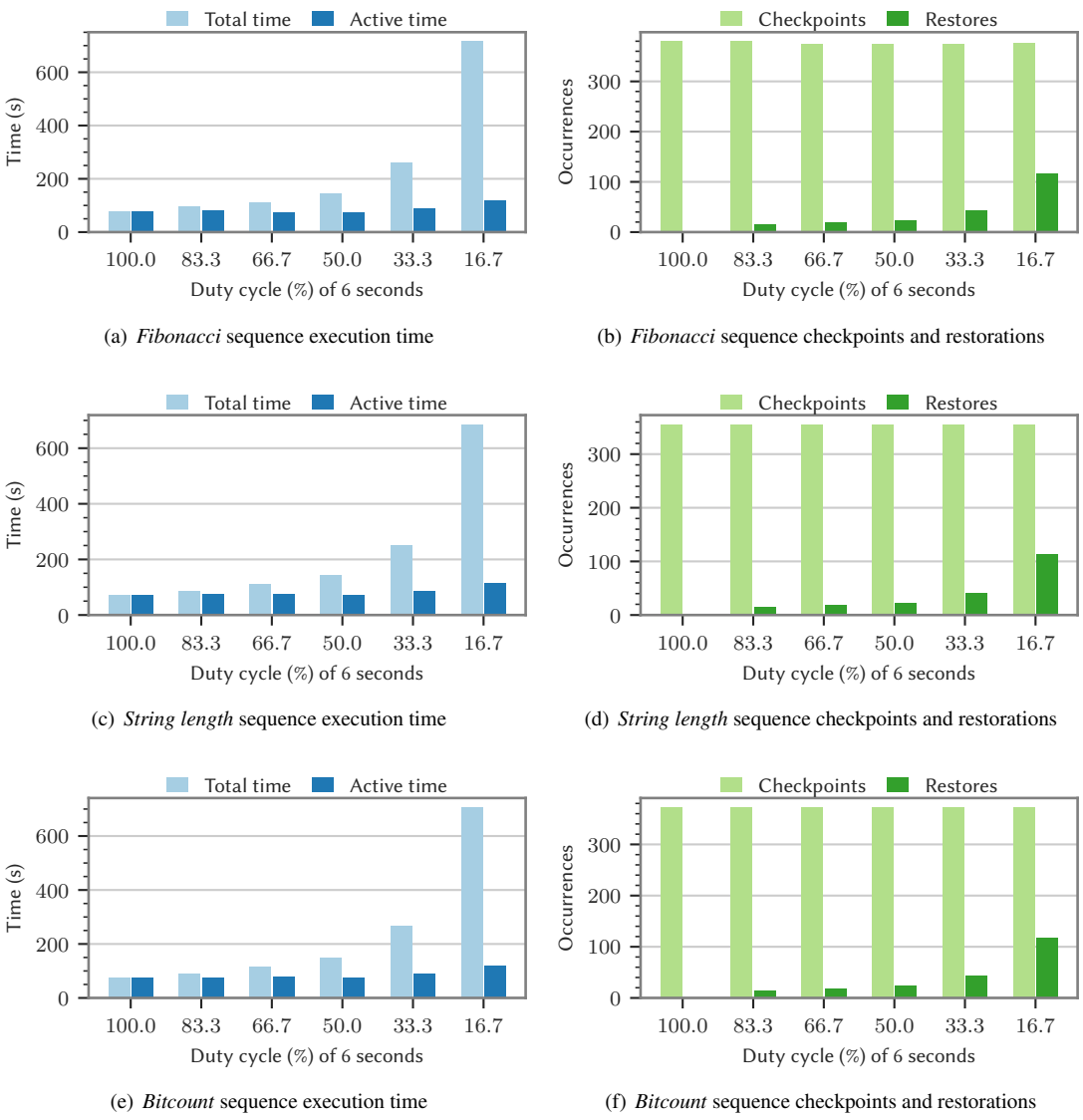
6.3 Execution in Varied Energy Harvesting Environments

To evaluate the power-management and harvesting subsystems of the BFree shield, we directly connected a solar panel to the BFree shield and recorded a trace of the system. The Python code running during this trace is the *Fibonacci* benchmark used in Section 6.2. We used a 6 V 0.6 W 80×55 mm off-the-shelf solar panel [70] and recorded the system's operation during an 80 second time window for two different light intensity levels. These levels are 1000 lux and 3000 lux and correspond, approximately, to an overcast day and shade during a sunny day, respectively. Note that no maximum power point tracking was performed as the solar panel was directly connected to the BFree shield; a 15 mF storage capacitor was used.

The results are shown in Figure 13. As can be seen the active time (light blue) in Figure 13(a) is significantly shorter than in Figure 13(b), because of this the low energy operation (dark blue) is extremely short. We also clearly see the intermittent operation for both cases of light intensities.

6.4 Measuring the Effect of Checkpoint Strategies

Different energy scenarios require different checkpoint strategies (refer to Section 4.2). To observe the performance of each of the introduced strategies, we recorded a single active period for each of these strategies (periodic, trigger, and hybrid), each with two different configurations. The system configuration is identical to the one used



(a) *Fibonacci* sequence execution time

(b) *Fibonacci* sequence checkpoints and restorations

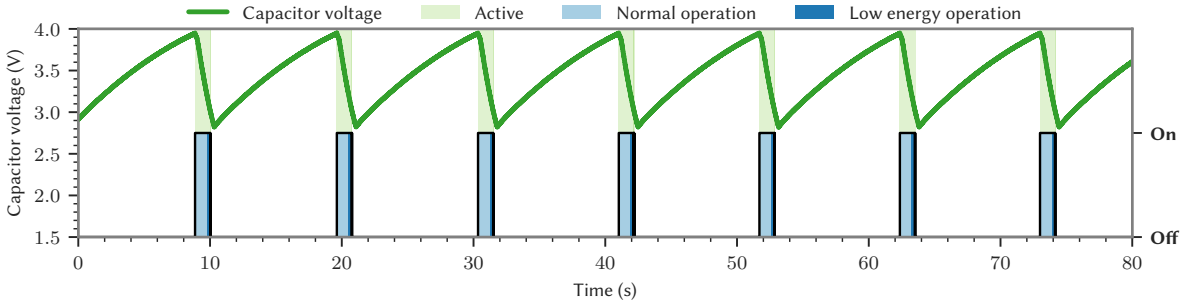
(c) *String length* sequence execution time

(d) *String length* sequence checkpoints and restorations

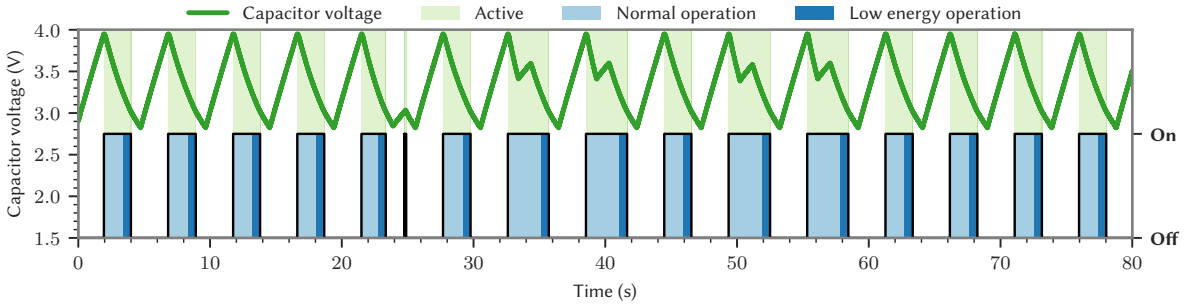
(e) *Bitcount* sequence execution time

(f) *Bitcount* sequence checkpoints and restorations

Fig. 12. Evaluation of the *Fibonacci*, *String length*, and *Bit count* benchmarks written in Python (the same ones as used in Figure 11), using the same input as mentioned in Section 6.1. The benchmarks are executed in a loop of 30000 iterations for Fibonacci and bit count, and 10000 iterations for the string length application. The intermittency rate is varied using different duty cycles of a six second period (Table 1) with the addition of the 16.7% duty cycle. The period checkpoint strategy is used with a checkpoint every 200 ms. (figures (a, (c), and (e)). As with the LoRa demonstration app (Section 5.1), these apps are computational and not time-based, therefore (again) the total time increases with the increase of the off-time. The slight increase in active time is due to the re-execution of code that happens after the last successful checkpoint, and restore operations (figures (b), (d), and (f)). Due to the period checkpoint strategy, the number of checkpoints remains the same. The number of restores is equal to the number of reboots of the system, and therefore is also equal to the number of off periods.



(a) 1000 lx light intensity



(b) 3000 lx light intensity

Fig. 13. Recording of the energy trace (the voltage on the capacitor) when running the *Fibonacci* benchmark. The light intensity used was (a) 1000 lx, and (b) 3000 lx. Both traces used a 15 mF storage capacitor and the power failure signal (signaling low energy operation) is configured at 3.25 V on the storage capacitor. The low energy operation is nearly imperceptible at 1000 lx as the voltage of the storage capacitor is falling too fast.

in generating Figure 13(b). The results are presented in Figure 14. The impact of the different strategies on the *Fibonacci* benchmark can be seen in Figure 15, with the summary presented in the caption of the figure.

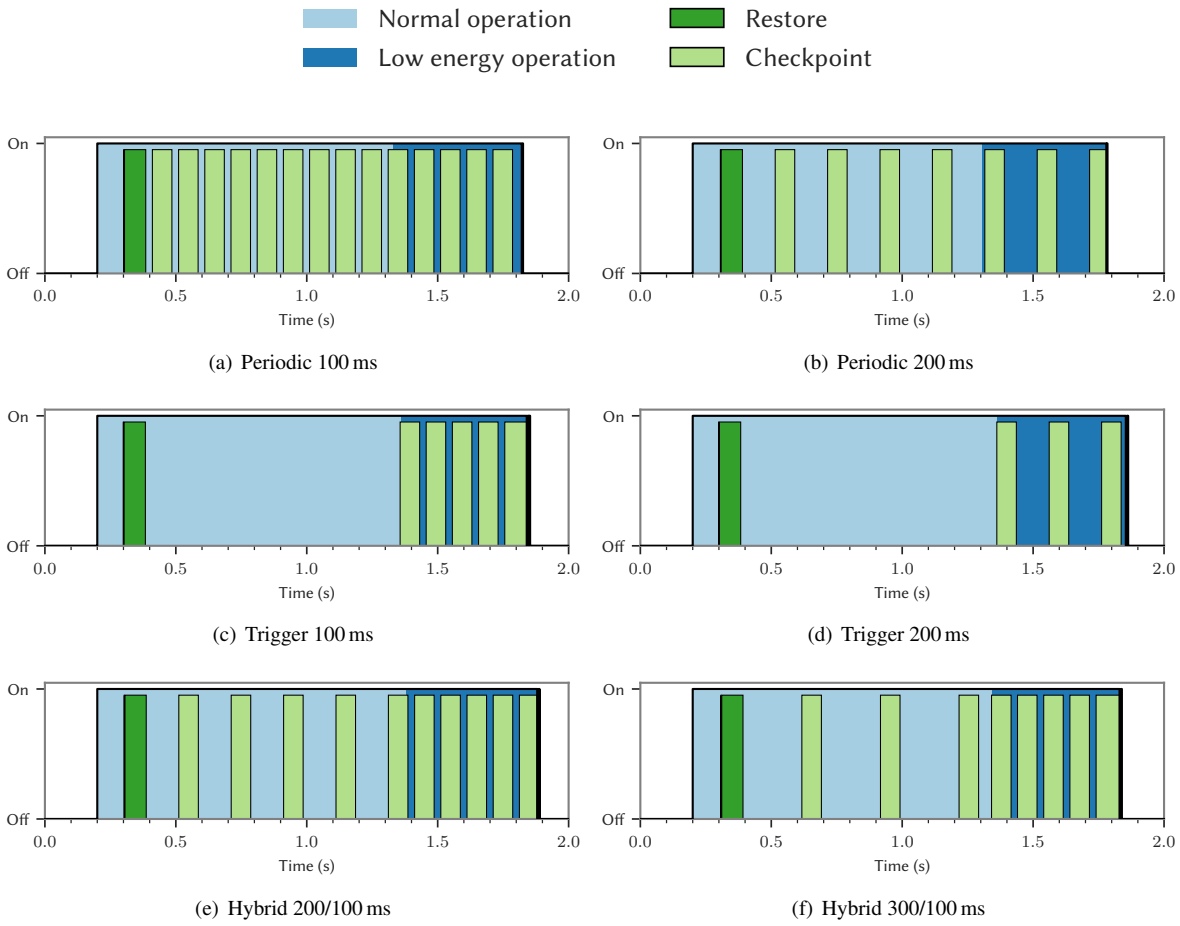
6.5 Startup Time

We measured the cold boot time of the baseline CircuitPython system to be 700 ms. By applying the techniques described in Section 4.2, we reduced the boot time of BFree considerably. A full boot of the BFree until the point the system is ready to restore a checkpoint is now approximately 270 ms. A detailed result of this comparison is presented in Figure 16.

6.6 Resource Usage

We measure the overhead in memory and speed incurred from our checkpointing routines.

Checkpoint Content: BFree either periodically or on-demand checkpoints the volatile system state to the BFree shield over SPI. The transfer speed and the checkpoint size are dominating factors when it comes to the overhead introduced by BFree. The current SPI frequency is dictated by the TI MSP430FR5994 microcontroller [94] and is set to 3 MHz. The checkpoint size is constant and dominated by the size of the garbage collector and is therefore



(a) Periodic 100 ms

(b) Periodic 200 ms

(c) Trigger 100 ms

(d) Trigger 200 ms

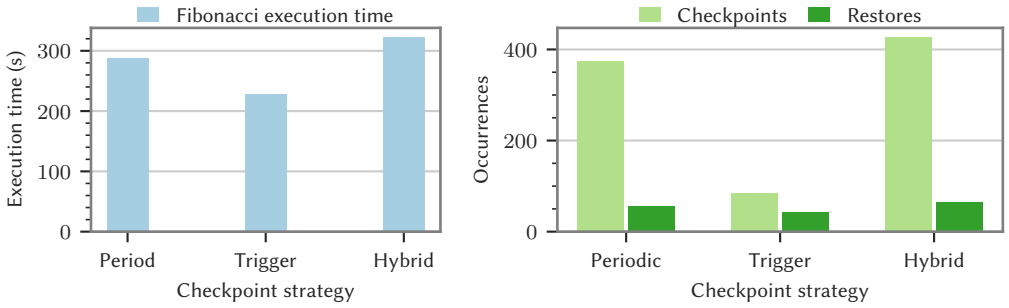
(e) Hybrid 200/100 ms

(f) Hybrid 300/100 ms

Fig. 14. Recordings of the checkpoint strategies—*period*, *trigger*, and *hybrid*—each with two different configurations that can all be configured at runtime by the user. The period based strategy, figure (a) and (b), does not take low energy into account when scheduling a checkpoint. When using the trigger based checkpoint strategy, figure (c) and (d), checkpoints are only performed when the energy level is low. Lastly the hybrid approach, figure (c) and (f), has two configuration periods: (i) one during normal operation, and (ii) one during low energy operation.

always 27.6 kB. The distribution of the memory regions making up the checkpoint are shown in Table 2. Therefore, the time spent performing a checkpoint is also constant and measured it to be approximately 75 ms. A restoration is slightly slower, averaging around 80 ms. The SPI frequency can be increased to 4 MHz to improve the checkpoint time to approximately 52 ms, but this can leave the system vulnerable to interference on the SPI bus. Therefore, all the experiments were performed at the slower clock speed of 3 MHz.

Checkpoint Code Overhead: The memory footprint of BFree compared to CircuitPython is presented in Table 3. Therein, it is seen that the FLASH footprint, representing the additional code, is almost the same for both CircuitPython and BFree due to the scale of the CircuitPython project. The additional mechanisms to allow for



(a) *Fibonacci* sequence execution time

(b) *Fibonacci* sequence checkpoints and restorations

Fig. 15. Evaluation of the Fibonacci benchmark from Section 6.2 running on intermittent power harvested using the same hardware setup as in Section 6.3. The chosen light intensity was 3000 lx; a 15 mF storage capacitor was used. We tried to keep the light intensity—and therefore the active time—constant, but in practice it varied from 2.1 to 2.6 seconds. The checkpoint strategies where configured as in Figure 14 (b), (d), and (f), respectively. The results show that the total execution time, see figure (a), not only depends on the chosen checkpoint strategy, but also the configuration parameters of the strategy. In theory, the total time of the hybrid strategy can be lower than the period strategy. In this case, the number of checkpoints taken, see figure (b), is lowest for the trigger strategy and highest for the hybrid strategy. The hybrid strategy can be improved by either (i) lowering the checkpoint frequency of any of the the energy operation modes, or (ii) by tuning at what voltage of the storage capacitor the low energy operation starts (see Section 4.1 (**Power Failure Prediction**)). The number of restorations, see figure (b), directly translate to the number of off periods encountered during operation.

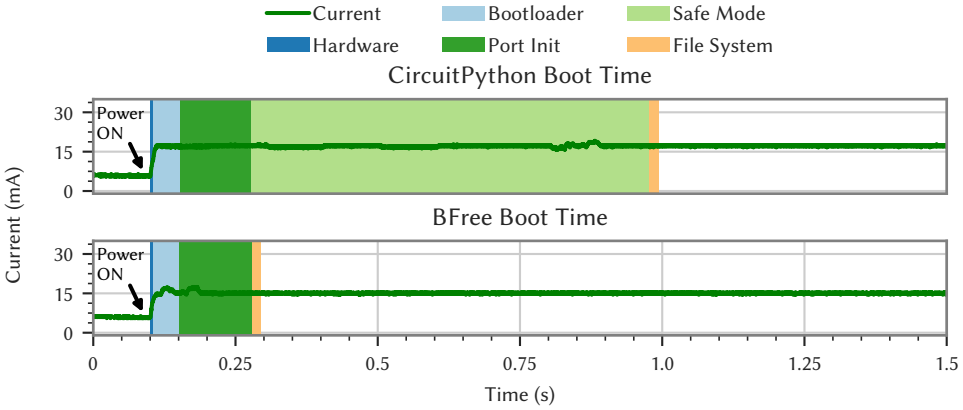


Fig. 16. Measurement of the boot time of pure CircuitPython running on Adafruit Metro M0 board only (top graph), and CircuitPython running on top of Adafruit Metro M0 board, connected to BFree shield running BFree runtime (bottom graph). Current measurement was taken with a digital oscilloscope connected to a shunt resistor connected in series to Adafruit Metro M0 power supply port (powered by 3.3 V from external source). Bottom figure clearly shows that we have optimized BFree boot time considerably by eliminating the introduced delay to enter 'safe mode' and by disabling the on-board RGB LED (used to show the current stage of operation on Metro M0 board), reducing current consumption and voltage fluctuation during startup. BFree current consumption (shield and runtime) is is approximately 2.2 mA and is the difference in current consumption between the top and the figure.

Table 2. BFree checkpoint content and its respective size in Bytes.

Checkpoint Content	**Size** (Bytes)
registers	68
.data	16
.bss	1320
stack	5280
garbage collector	20916
total	27600

Table 3. CircuitPython and BFree memory footprint. For ease of comparison the increase in FLASH and RAM use between CircuitPython and BFree is directly noted. The remaining memory for the CircuitPython application is roughly 24000 bytes when using BFree. To put these number into perspective, the RTC library used in Section 5.2 uses approximately 8000 bytes of memory, almost *seven* times more than the additional memory required by BFree.

CircuitPython (Bytes)		**BFree** (Bytes)		*Increase* (%)	
FLASH	RAM	FLASH	RAM	FLASH	RAM
193488	6840	204336	8000	5.61	16.96

checkpoint creation and restoration introduced by BFree increase the RAM used by almost 17%, reducing the memory left for CircuitPython applications by 7.3%. To put these number into perspective, the RTC library used in Section 5.2 that was written by the CircuitPython community (in other words: not the authors of this paper), requires almost seven times more memory than the additional memory required by BFree. Most of the increase in memory consumption is caused by additional data structures introduced to keep track of the peripheral state during execution and a 512 Byte 'safe stack' used to execute the checkpoint routines without changing any of the memory in use by the Python interpreter.

6.7 Discussion of Results

We consider different aspects of the evaluation results.

Performance: As shown with the benchmarks' performance, the additions made to the CircuitPython runtime system do not significantly hamper the operation of programs while plugged in. When running on harvested and intermittent power, the performance is mostly determined by the amount of energy that can be harvested and the length of power failures. We note that without careful recovery mechanisms like those implemented in BFree, CircuitPython programs will go into inconsistent states, corrupt memory, or crash when harvesting energy and running intermittently.

Overhead: The current mechanism for checkpointing takes nearly the entire volatile memory and saves it. This takes a significant amount of time to checkpoint versus bare-metal, C-based embedded runtime systems. However, this level of overhead is often acceptable for reactive, human speed, sensing-based, and low numerical complexity programs run by makers and hobbyists.

Hardware Limitations: We note that the CircuitPython Metro board is not necessarily designed for low power operation or untethered operation, as the 10–15 mA operating range is quite high (see again Figure 16), and the selection of circuitry for USB communication, power regulation, and others are designed for ease of manufacture

and cost. Additionally, peripherals are connected in such a way that they use power even when off. A careful redesign of the Metro M0 board would easily reduce total power consumption by an order of magnitude by using lower power ARM microcontrollers, e.g., [7], selectively gating peripherals, and redesigning the power conditioning circuitry. Despite this, we have shown that application development with BFree is still possible and performant.

Fair Comparisons: We do not compare against state-of-the-art intermittent runtime systems like InK [102], Alpaca [60] or Chain [16], as these are written for professional programmers in C. Neither BFree nor CircuitPython/MicroPython can approach these runtimes' low overhead or performance (as shown in Section 6.1 comparing Python to C), as they spend significant clock cycles interpreting Python bytecode, managing the Python runtime, and facilitating operations that make the novice programmers life easier (such as handling USB connections and file systems). However, these systems are not in competition as they present radically different programming models, with the former serving expert developers. In contrast, we propose to serve novice and hobbyist developers seeking entrance into the ubiquitous and untethered computing world.

Harvesting Tradeoffs: During our evaluation, we used a single storage capacitor, and we only measured the trace using one solar panel (although with different light intensities). Changing these will affect energy harvesting and, potentially, the operation of the application. If we consider an application with a certain energy requirement, increasing the capacitor size will lead to a longer active time. This means that a different checkpoint strategy might be optimal. If the harvester output energy is increased (e.g., by using a different solar panel), the active time will also increase. However, increasing the capacitor size will also affect the off-time of the system, as the capacitor also takes longer to charge. Because BFree is intended for hobbyists and can be applied in various ways, in many different configurations, we did not exhaustively evaluate all the combinations. Instead, we attempted to provide an overview of the possible combinations and the significant number of parameters that can be tuned in BFree—both in the hardware and the software—to gracefully handle all these different harvesting scenarios considered in Section 6.3 and Section 6.4.

7 USER STUDIES

To complement the evaluation results of BFree presented in Section 6, we conducted two user studies[6], both approved by the Human Research Ethics committee of Delft University of Technology, that ask the following **research questions**:

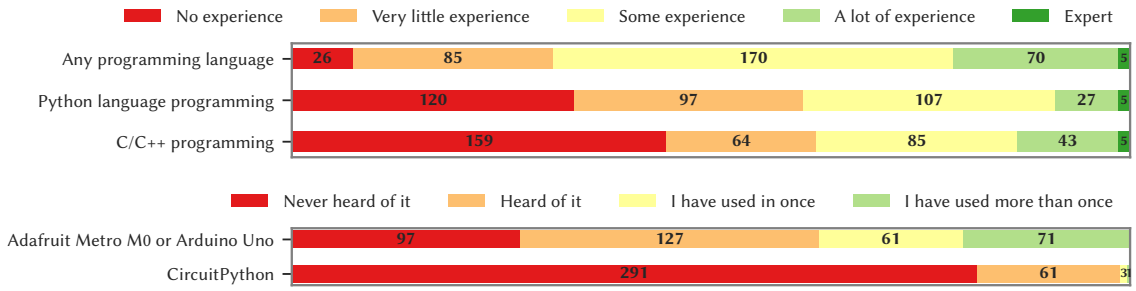
- **RQ1:** Would programming considering intermittency support in (Circuit) Python be *easier* than with state of the art intermittent programming runtimes (written for the C language)?
- **RQ2:** Is the system we built *usable and useful* with regard to making battery-free electronics (powered by ambient sources) for low skill, inexperienced makers?

These research questions help us to understand how novice programmers would best be able to program systems previously only used by experts.

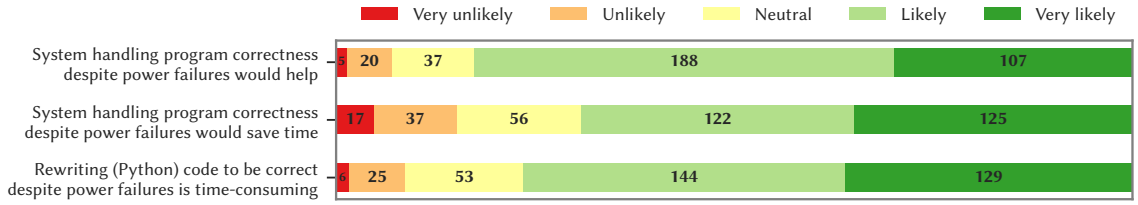
7.1 BFree Language Comparative Study

To answer question **RQ1**, we conducted a study on a large group of computer science students of the Delft University of Technology. We asked them to compare Python and C/C++ in the context of application development for intermittent computing. The purpose of this study is by no means to evaluate the BFree programming language (i.e., CircuitPython) or its sub-components, but to verify the hypothesis that (Circuit) Python programming language (with intermittent execution support engine invisible to the programmer) is easier to use for a novice programmer, compared to state of the art software solutions targeting expert programmers, such as [16, 44, 60, 102] based on the C language.

[6]Details about the study, including questionnaire and the detailed answers given by participants of both studies are available in [96].



(a) Technical capabilities self-assessment of BFree language comparative study participants



(b) BFree usability questionnaire responses after exposure to task of finding a bug in a Python code transformed to handle power intermittency

Fig. 17. BFree language comparative study results (356 participants; average age of the participant of the study was 19 years, with the youngest participant being 17 years old and the oldest—35 years old). Figure (a) presents study participants technical capabilities self-assessment, while figure (b) presents answers to questionnaire provided after participants were exposed to a simple task of finding a bug in a Python code transform to handle power intermittency. A large group of participants with little programming experience and exposure to hobbyist embedded systems unanimously agreed that system that handles program correctness despite power interrupts would help a programmer and save development time. Note: number on each bar in both figures represent the number of responses to each question.

7.1.1 Participants. As noted earlier, our study was performed among a large pool of students of computer science of Delft University of Technology (details on the student cohort are given in the caption of Figure 17). We aimed at students of varying levels of experience with computer programming and with hobbyist-level microcontrollers. Specifically, we have presented the questionnaire during a break of one lecture of the first-year undergraduate "Object Oriented Programming" class, second-year undergraduate "Digital Systems" class, and graduate-level "Analytics and Machine Learning for Software Engineering" class.

Participants have self-assessed their technical abilities and the knowledge of Adafurit Metro M0 and CircuitPython by answering a short questionnaire at the end of the study, which will be described in the subsequent section. Results of the self-assessment are given in Figure 17(a), with the raw data accessible in [96].

Based on the outcome of the self-assessment, we conclude that majority of students never heard of CircuitPython and have very little experience with using Arduino Uno [9] or Adafruit Metro M0 [3]. Moreover, we conclude (see again Figure 17(a)) that the majority of students have little or no programming experience (especially in C/C++). This level of experience is representative of inexperienced hobbyists beginning with active use of embedded electronics platforms.

7.1.2 Design and Execution of the Study. Firstly, we presented the study participants with a short overview of the challenges of program execution on battery-free devices running intermittently, followed by the link to the questionnaire. The questionnaire started with a short textual introduction to CircuitPython and small embedded microcontrollers, followed by a short description of the intermittent computing problem. Neither during the overview, nor in the introduction to the questionnaire, it was revealed that the authors are the designers of BFree hardware and BFree Python, to avoid any bias in giving answers.

We then proceeded with the exercise that exposed the respondents to the cognitive burden of intermittent computing. That is, we provided a short explanation on how to convert a Python code into one that can run intermittently. This conversion is done by the process of *task transformation*, shown earlier in Figure 4, e.g., in the same way as proposed by the state of the art runtimes for C language such as InK [102] or Alpaca [60]. Then the participants were asked to find a bug in the Python code of simple "variable swap" function, which does not use an extra temporary value, i.e., the code executed

```
a = a + b; b = a - b; a = a - b
```

.

After this step, we presented three different task-transformed implementations of the original Python code of this "variable swap" function. Two of these implementations were 50 lines long, while the third one was 52 lines long. Exact code listing is provided in [96]. Among these three intermittently-executable program options two contained a bug: one option had an incorrect task transition and the other option did not save the program state completely. Respondents had to chose the bug-free version one from the three choices. After a choice was made respondents were not allowed to change their answer.

7.1.3 Result. From all respondents 78.7% (292 out of 371 responses to this question) found the correct answer and spent approximately five minutes on this task. This implies that the respondents were educated enough to answer the core set of questions.

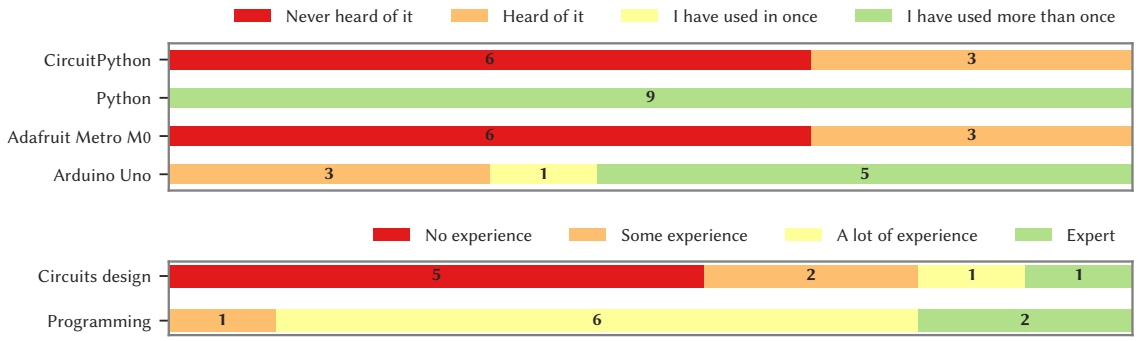
After the participants gave an answer to this exercise we then asked three core questions pertaining to the simplicity of BFree Python. Answers could be provided in the five-level Likert scale. Questions and the results are shown in Figure 17(b). Therein, we see that an overwhelming majority assessed that developing intermittent programs using BFree Python is *easier* (and also *faster*) than using existing systems (i.e. manual transformation of the code for intermittency protection) that lead to extensive cognitive burden. We consider this as a positive answer to the research question **RQ1**[7].

7.2 BFree User Experience Study

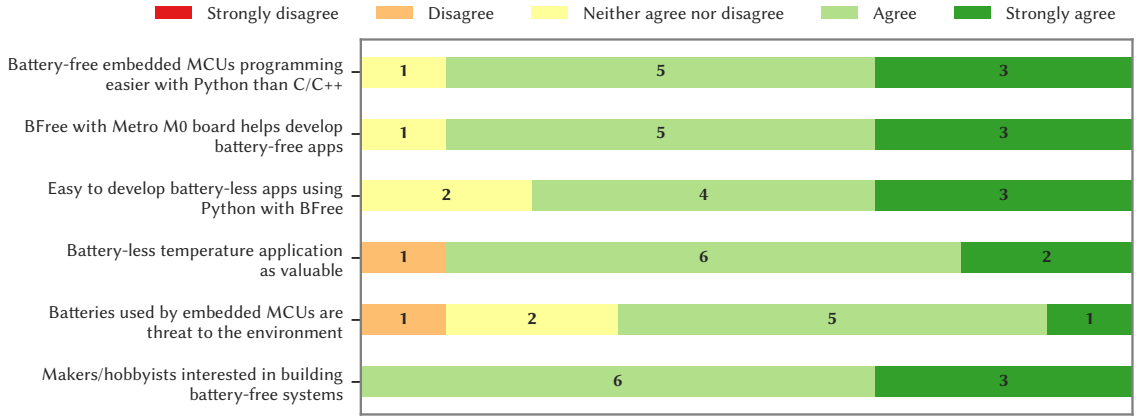
We proceed with the second study aiming at answering **RQ2**, by asking a small set of students to experiment with a real BFree platform (that runs a real battery-free application) and provide feedback on BFree's use.

7.2.1 Participants. For the BFree user experience study we have selected nine participants—all university students of either MSc or PhD level (details on this student cohort are given in the caption of Figure 18). The study participants have self-assessed their technical capabilities at the end of the study by answering a set of questions. Detailed result of this self-assessment is presented in Figure 18(a), with the raw data available in the BFree online repository [96]. Analyzing the responses in Figure 18(a) we see that a group was diverse and none of the students had any experience either with CircuitPython or with Adafruit Metro M0 board. This was a desired outcome, as this way the participants were not biased in assessing CircuitPython and the Adafruit Metro M0 board. At the same time all participants claimed to use a regular Python language and most study participants claimed to use another popular hobbyist-grade embedded microcontroller platform—Arduino Uno. Participants of the study were found

[7]We recall that the task transformation is one of the core ways of preparing code for intermittency for advance embedded systems programmers, see [60, 102].



(a) Technical capabilities self-assessment of BFree user experience study participants



(b) BFree usability questionnaire responses after 20 minute session of experimenting with battery-free temperature measurement app

Fig. 18. BFree user experience study results (with one female and eight male participant; two participants were 21 years old, one participant–24 years old, two participants–25 years old, three participants—26 years old, and one participant—29 years old). Figure (a) presents study participants technical capabilities self-assessment, while figure (b) presents answers to the BFree usability questionnaire after hands-on session with a real battery-free application. A diverse group of participants, having no practical experience with CircuitPython and Adafruit Metro M0 board, responded positively to BFree aiding in developing battery-free applications. Note: number on each bar in both figures represent the number of responses to each question.

via email announcements to student groups known to the authors of this paper. No direct professional connection was present between the participants and the authors of this study.

7.2.2 Design and Execution of the Experiment. Each participant was invited to the room specifically prepared for the BFree experience study, as not to influence the participant with the supervisor (or other people's) presence. The participant was asked to sit in front of a desktop PC, standing on a regular office table. This PC was connected via an USB port to Adafruit Metro M0 board, to which BFree board was attached to. On the PC's monitor a web browser was opened that contained a short description of the experiment. The participant was asked to read this

description first. This description was accompanied by the same short introduction to intermittent computing as given to the participants of the language comparative study presented earlier, see Section 7.1. After reading the description of the experiment, we explained the experiment to the participant again—this time in person—giving each participant the opportunity to ask questions about the experiment process.

The explanation itself was performed as follows. The same PC located in a room had also a pre-loaded program editor with prepared temperature measurement application written in CircuitPython (code of this program is available in [96]). We asked each participant during the explanation to upload this code to the Adafruit Metro M0 board and subsequently disconnect it from the USB port. Disconnection from the USB port effectively made Adafruit Metro M0 board intermittently-powered by ambient indoor light. Additionally, a table on which the PC and BFree board was located, was also equipped with a light bulb (to imitate strong light source) which participant could turn on and off, controlling the rate of intermittent power supply. Participants were then asked to connect the board back to the PC (making it again continuously powered) to read-out the temperature measurement, which continued from the last moment the Metro M0 with BFree was powered (indicated on a terminal window with an increasing counter).

After this explanation the participant was ready to perform an actual experiment. For this we asked each participant to redo the process we demonstrated, asking first to modify (in any way the participant deemed interesting) original code of the temperature measurement and then upload in to BFree. In other words we asked the participant to simply "play" with the code and BFree (by increasing or decreasing light of the light bulb, covering solar panels with hands, plugging USB cable in then reading the measurement and unplugging it again as many times as possible, etc.). We then left the participant alone in the room. Each participant had about 20 minutes for this task. After completing the experiment the participant was asked to answer a short questionnaire related to the experience with BFree. The questionnaire was password-protected and the password was shared only after the participant completed the experiment.

7.2.3 Result. Answers to closed and open questions provided after the completion of the experiment are provided in Figure 18(b) and Table 4, respectively.

Based on the individual experience session with BFree majority of participants agreed that BFree helps to develop battery-free applications, making a development of such application easy (see Figure 18(b)). Participants found the environmental impact of batteries is existent and makers/hobbyists would be interested in using BFree.

At the end of the study participants also suggested a set of applications, deployment scenarios, listed strong and weak points of BFree, as well as provided remarks on BFree they briefly experimented with. Succinct list of answers is given in Table 4. These comments also point us to potential areas for future work.

Based on the above outcome we conclude that BFree is *usable and useful*, which positively answers research question **RQ2**.

7.2.4 Limitations of BFree Experience Study. The experience study has limitations. The main one being the lack of access to the real makers community that would provide matched and in-depth assessment of our developed platform. Also, larger participants pool and longer time provided to the participants would result in more expressive evaluation of BFree. Longer experience would also enable users to develop more sophisticated applications, gaining more knowledge on the limitations of BFree. Finally, participants of this experience study were using the first version of BFree, with many, still unresolved at that time, bugs. We note that the results provided in this paper (Section 6) are based on the much newer version of BFree shied and BFree runtime.

Finally, we remark that this user experience study was executed in mid-February 2020. Since then we could not redo the experience study with a new version of BFree due to COVID-19 restrictions regarding people's presence at our university. Simply, it was logistically hard to control flow of people in and out of the room where the experiment was performed.

Table 4. Selected responses to open questions regarding BFree usability made by participants of BFree user experience study. Exact response text from participants were compressed to fit this table. For exact and complete set of answers the reader is referred to [96].

Open question	Responses
What apps would you *develop* with BFree?	• Event counter powered by harvested energy • Weather sensors/wildlife counters in non-populated regions • Crypto-currency mining • Aerospace (when power cuts out due to vibrations rocket state machines could continue right where left off)
What apps would you *deploy* with BFree?	• I would not use this to deploy a product: it feels too much 'hobbyistic' • Cheap wireless sensors everywhere to measure temperature, humidity, light, motion
What are the *strong points* about BFree?	• Easy to use, does not need complicated setup, easy to compile code • No need to think about keeping state in your code • If succeeded making it 100% seamless[1] developers would not need to learn anything to use battery-free feature
What are the *weak points* about BFree?	• Unknown what the approximate power limitations are and how these combine with different hardware • Unclear which operations are safe and what are the risks if the power dies during a task • Harder to debug/test than a standard microcontroller • Doubt about use cases: typically device would either run all the time or it would not matter if it completely resets
Do you have any *remarks* about BFree?	• Stress to audience how it is different from any solar-powered computer • Better documentation on how to make sure power cuts are handled safely • Have smaller form factor BFree boards • People will be able to come up with cool applications for BFree

[1] User reported an issue with device resetting when connected to USB port of a PC, which was corrected with the latest revision of BFree board; refer to explanation in Section 7.2.4.

8 RELATED WORK

This paper merges two visions: the future of embedded computing —*battery-free intermittent computing*—and *sustainable and novice-oriented* programming environments. This work, built on expert-oriented intermittent computing systems, is the *first interpreted runtime for intermittent computation*, and the first such system targeting novice and hobbyist developers.

Intermittent Computing: Devices like the WISP [86], computational RFID tags which harvest energy from RFID reader transmissions, were the first attempts to enable battery-free, energy harvesting embedded computing. Programming for these devices was incredibly difficult and inefficient, so runtime systems that instrumented C programs with checkpoints [58, 82] or transformed these programs into tasks [16, 35, 102] were created. Platforms that leveraged these new runtime systems were developed to increase energy-efficiency and dependability of intermittent computing applications [17, 31]. Tools that enabled deeper introspection into the energy environment [30] and

brought command line debugging support [15] further enhanced the ability of *experts* to deploy battery-free systems. None of these systems have addressed novice developers, focusing on performance and energy-efficiency. Even platform such as Flicker [31] mentioned above, that proposes a modular hardware system (similar in spirit to BFree) where individual hardware modules (energy harvesters, computation engine, wireless communication modules, displays, sensors, etc.) can be mixed in various combinations to create an application-specific sensor, requires a very good knowledge of embedded C programming. But more importantly Flicker has no dedicated runtime that handles energy intermittency. Also, Flicker modules are not backward compatible with popular hobbyist-grade microcontroller boards (such as in the case of BFree: Adafruit Metro M0) and does not have enough of memory to store the complete BFree runtime.

Recent technology advances in non-volatile memory storage and ARM Cortex platforms [10] enable us to make the first interpreted language for battery-free devices, and the first focused on enabling the *novice* instead of focusing on performance as in previous approaches. Moreover, as the novice is not as concerned with minute optimizations and performance, but mostly with getting something working, the overhead of our interpreted language approach compared to the state-of-the-art is not detrimental.

Battery-free Applications: We can already list single purpose applications that eschew batteries designed by experts such as wearable health [81], environmental monitoring [34], pervasive or wearable displays [20, 26], wearable authentication and haptics [54], gesture recognition [53], rapidly prototype interactive objects [52, 88], making Skype calls [91], enabling smart spaces [22], and others. This set of applications are enabled by the range of energy available to harvest in various forms, gathering from sunlight, radio frequency transmissions, vibrations, heat [73], human movement [101], and even microbial communities [21]. These applications show the potential for our work, as all of these systems are research products made by experts, often combining advanced techniques spanning computing, electrical engineering, and physics. We believe that BFree will enable these applications to be developed by novices, and potentially ease the development process for experts.

Sensing Platforms: Platforms for wireless sensor networks (WSNs) research and deployment have been developed over the past two decades [36], most notably with the TelosB [77] which was one of the most successful general purpose sensing platforms, and Prometheus [39] the first energy harvesting sensor platform. Energy harvesting WSNs [6] have began to dominate the sensor world because of decreased maintenance cost and longer deployment lifetimes. Building on these works, synthetic sensors for general purpose embedded computing framed towards the HCI space [47] were developed, along with other platforms meant to increase the applicability and generality of sensors like Hamilton [42], and EcoMicro [49]. None of these platforms can run firmware in Python. Also, none of these platforms can enable battery-free deployment, being built on the assumption that power is continuous and reliable, even if constrained. BFree is specifically engineered to handle the various system difficulties when faced with frequent power failures, enabling development of robust battery-free, untethered applications by novices.

Novice-Oriented Programming: Developing tools and systems to increase access and applicability of computing to novices has a long history and inspires this work. Well known systems like Logo, Scratch, Processing, and Arduino represent programming environments designed with a low learning curve and directed toward makers, artists, designers, and inexperienced programmers. Platforms like Codeable Objects [37] extend programming to the physical world. Bifröst [64], WiFröst [65] and Scanalog [89] help with debugging complex hardware and software embedded systems. Python Tutor [27] and OmniCode [41] and similar work has sought to teach Python programming to the novice with interactive execution. All of these systems focus on an area where novice programmers are under-served, in this spirit, BFree opens up battery-free and energy harvesting programming to the novice. We believe that interactive programming environments are an interest area of future research to help novice programmers predict how their battery-free application will perform in the wild.

Computing for Conservation and Sustainability: BFree is motivated by the growing ecological concerns associated with climate change and planetary stewardship that has inspired significant work in computing, HCI, and sustainability [43, 76]. Systems that are designed to encourage energy conservation are tangentially related to this work [90]. For example, EnergyBugs [85] materialize the unseen energy into a tangible object. Persuasive displays [45], eco-feedback systems [24], and other battery-free systems are a response to the increasingly devastating ecological impact of a battery-powered Internet of Things. Other work has advocated for sustainable, responsible approaches to building the smart city [29] and more considered HCI in agriculture [56]. We view BFree as an attempt to democratize mobile and wearable computing with an ecologically responsible view, building on the intellectual underpinnings of work in sustainability. Computing with batteries has offered convenience, but has constrained the design space of ubiquitous computing. As a tool, BFree devices offer ways to visualize energy, show the power of responsible computing practice, and assist in novel computing applications.

9 DISCUSSION AND FUTURE WORK

This work is only the beginning, opening up the possibilities of battery-free devices, for *everyone, everywhere*. We anticipate major research directions to be taken that will enhance the workflow and systems presented here, so that novice developers can be a part of a sustainable future of ubiquitous computing.

A General Platform: Future work could allow the BFree shield to be used without Python, and instead with other languages, both compiled, such as C and Rust, to those interpreted such as JavaScript, and even MakeCode [67] block-based languages. This potential for a general platform that supports many languages is a future goal of BFree such that any person, at any skill level and with any past programming experience can engage in battery-free, energy harvesting prototyping and software development. However, enabling this is not trivial, as checkpointing routines, instantiation mechanisms, and memory management would need to be re-evaluated per language. For example, JavaScript has an even more flexible (often confusing) specification than Python, with more complex bytecode. We leave this for future work.

Alternative Solutions: Potentially one could write a Python program that stored data right after it was generated, and upon start would read that data and try to resume. In essence, this approach amounts to rolling a custom intermittency solution, running into all the problems described earlier in this paper. Manually keeping progress is not simple, as it is not known when a failure might happen, and in fact, a failure could happen before the Python runtime even starts or a single line of user code is executed. Another alternative is making your program small enough and simple enough that it will likely finish before the storage capacitor depletes. While this is possible, it is not ideal, as one must first guess how much energy one has and how much energy a line of code costs, becoming very constrained in what one can do. BFree allows makers to program just like they always do and not worry about power failures and recovery.

Garbage Collection: The garbage collector in CircuitPython is a black box: during a startup, required fixed size memory regions are allocated, and then the remaining memory is allocated for dynamic memory requested by the Python application. Because of the black box nature of the algorithm used by the garbage collector it is impossible to find the exact occupied memory regions. For this reason, we are forced in BFree to save all the memory available to the garbage collector during the checkpoint procedure, even when most of it is not actually used by the Python program. A future improvement would be to replace the current garbage collector with one that causes less fragmentation, or adapt the current one to both: (i) expose the memory used by the Python application and (ii) have an efficient compacting method which is called before each checkpoint to reduce the checkpoint size.

Performance: The performance of BFree, especially considering the speed of program resumption after restart, requires further work. Also, we acknowledge that Python code hand-instrumented (but optimized) for intermittent

operation *might* be faster than our non-optimized BFree implementation. This observation has been echoed by some of the respondents of our survey, quote:

> *(...) programmers are pretty adamant of being able to control every inch of their code if they need efficiency, and handwritten code will almost always be faster.*

Nonetheless, the aim of BFree it to enable hobbyist or maker programmers, for which the speed of execution is of secondary importance to the usability of the whole system and rapid prototyping ability. We hope to increase both usability and performance to ease access to the battery-free computing domain.

Checkpoint Strategies: BFree provides multiple entry points and adjustable settings for creating checkpoint strategies. This work has only explored the surface level of strategies with just-int-time, periodic, and hybrid methods. In many cases and applications, none of these strategies would be ideal. Significant research space exists to explore and understand checkpoint strategies for interpreted languages. Because an interpreted language has access to the bytecode and other information from the program, that is not always available for bare-metal machine code, more intelligent checkpointing could be envisioned. Moreover, with the larger memory space and compute power of the ARM-based BFree system, more sophisticated checkpoint methods could be explored that take into account history, trends, or even energy aware prediction. These methods are bound to increase the performance of BFree and are a rich area to explore.

Platforms: Our BFree shield is the first proof-of-concept of what is possible with battery-free development for hobbyists and makers. Further hardware extensions could include, *emulators for energy harvesting* (to test the performance of the application), *capacitor size adaptation shields* (to test the code under different energy supply reservoirs), or *multi-sensor shields* (to experiment with different battery-free applications without the need to buy new sensors for each new experiment).

Applications: The success of battery-free intermittent systems solely depends on the richness of applications it can execute. As the field matures, more and more interesting applications arise, from smart protective equipment, to space satellites, to wearable devices that never need charging. That said, not all applications will immediately benefit from BFree, and not all users are even aware of the potential applications or power that battery-free operation gives. As one of the respondents of one of our studies (see Section 7) said:

> *The only application I can really think of (...) is long term execution (of) programs. Something like a neural network or genetic algorithm which may run for 24+ hours. It seems kind of niche but I can see the appeal.*

Therefore, more work is needed to think what "killer" applications for battery-free hobbyist micro-controllers could be, and more work is needed to encourage hobbyists to think beyond traditional boundaries of computing and energy. We view this platform as an enabler for interaction research; such as that extending energy materially [75], exploration of novel interactions that engage with energy, engineering persuasive exhibits or displays, and allowing novel wearables. In the future we hope to see interesting and dynamic applications written and deployed with BFree.

Tooling: Finally, a set of tools is needed, helping makers in experimenting with BFree. These tools have not been a focus of this work. This includes, user interface enhancements (for code development, code optimization and code debugging) and developer community code management system for BFree hardware and software, and energy introspection.

10 CONCLUSIONS

BFree allows makers and low skill hobbyists to develop computing and sensing platforms that not only run perpetually on harvested ambient energy but also make them free from batteries and a tethered power supply—thereby reducing the ecological and maintenance cost of traditional embedded systems. BFree's core innovation lies in developing the first power failure resilient runtime for an interpreted language (CircuitPython) that runs on embedded systems. With BFree, novice programmers can develop CircuitPython applications that sense, compute, learn, communicate, and much more—all without needing a battery or access to an electrical socket. BFree invisibly handles the frequent power interrupts caused by scarce ambient energy and small energy reservoir so that the programmer does not need to account for them. We evaluated BFree against a battery-powered CircuitPython baseline and showed reasonable overhead of BFree. Our system was tested with actual users, confirming the usability of BFree. Further, we open sourced the code and hardware of BFree as a resource to the community. With BFree we open a new application area for makers and hobbyists, and new realms of possibilities as we build the sustainable Internet-of-Things.

ACKNOWLEDGMENTS

We would like to thank our anonymous reviewers for their insightful comments. In addition, we would like to thank all the volunteers that participated in both of our user studies.

This research was supported by the Netherlands Organisation for Scientific Research (NWO), partly funded by the Dutch Ministry of Economic Affairs, through TTW Perspective program ZERO (P15-06) within Project P4, and by the National Science Foundation trough grants CNS-1850496, CNS-2032408, and CNS-2038853. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Adafruit. 2016. Si7021 Temperature and Humidity Sensor Breakout Board. https://www.adafruit.com/product/3251. (Sept. 2016). Last accessed: Oct. 27, 2020.
[2] Adafruit. 2017. PCF8523 Real Time Clock Assembled Breakout Board. https://www.adafruit.com/product/3295. (Aug. 2017). Last accessed: Oct. 27, 2020.
[3] Adafruit. 2018. Adafruit Metro M0 Express - designed for CircuitPython - ATSAMD21G18. https://www.adafruit.com/product/3505. (April 2018). Last accessed: Sep. 14, 2019.
[4] Adafruit. 2019. Welcome to CircuitPython! https://learn.adafruit.com/welcome-to-circuitpython. (Sept. 2019). Last accessed: Oct. 27, 2020.
[5] Adafruit. 2020. RFM95W LoRa Radio Transceiver Breakout Board. https://www.adafruit.com/product/3072. (Jan. 2020). Last accessed: Oct. 27, 2020.
[6] Kofi Sarpong Adu-Manu, Nadir Adam, Cristiano Tapparello, Hoda Ayatollahi, and Wendi Heinzelman. 2018. Energy-harvesting Wireless Sensor Networks (EH-WSNs): A Review. *ACM Transactions on Sensor Networks* 14, 2 (July 2018), 10:1–10:50.
[7] Ambiq Micro. 2018. APOLLO Ultra-Low Power Microcontrollers and SoC Solutions. https://ambiqmicro.com/mcu. (2018). Last accessed: Oct. 27, 2020.
[8] Arduino. 2019. Arduino GitHub Repository. https://github.com/arduino/Arduino. (Sept. 2019). Last accessed: Oct. 27, 2020.
[9] Arduino. 2019. Arduino Uno Rev3. https://store.arduino.cc/arduino-uno-rev3. (March 2019). Last accessed: Oct. 27, 2020.
[10] ARM Limited. 2019. Cortex-M0. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0. (Sept. 2019). Last accessed: Oct. 27, 2020.
[11] ARM Limited. 2019. Mbed OS 5 Website. https://www.mbed.com/en. (2019). Last accessed: Oct. 27, 2020.
[12] Domenico Balsamo, Alex S. Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. 2016. Hibernus++: a Self-calibrating and Adaptive System for Transiently-powered Embedded Devices. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 35, 12 (2016), 1968–1980.
[13] Pierre Carbonnelle. 2020. PYPL: PopularitY of Programming Language. http://pypl.github.io. (Aug. 2020). Last accessed: Oct. 27, 2020.
[14] Stephen Cass. 2020. The Top Programming Languages 2020. https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020. (July 2020). Last accessed: Oct. 27, 2020.

[15] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson Sample. 2016. An Energy-interference-free Hardware/Software Debugger for Intermittent Energy-harvesting Systems. In *Proc. ASPLOS* (April 2–6). ACM, Atlanta, GA, USA, 577–589.
[16] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proc. OOPSLA* (Oct. 30 – Nov. 4). ACM, Amsterdam, The Netherlands, 514–530.
[17] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices. In *Proc. ASPLOS* (March 24–28). ACM, Williamsburg, VA, USA, 767–781.
[18] Jasper de Winkel, Carlo Delle Donne, Kasım Sinan Yıldırım, Przemysław Pawełczak, and Josiah Hester. 2020. Reliable Timekeeping for Intermittent Computing. In *Proc. ASPLOS* (March 16–20). ACM, Lausanne, Switzerland, 53âĂŞ–67.
[19] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. 2020. Battery-Free Game Boy. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 3 (2020), 111:1–111:34.
[20] Christine Dierk, Molly Jane Pearce Nicholas, and Eric Paulos. 2018. AlterWear: Battery-Free Wearable Displays for Opportunistic Interactions. In *Proc. CHI* (April 21–26). ACM, Montreal QC, Canada, 220:1–220:11.
[21] Conrad Donovan, Alim Dewan, Deukhyoun Heo, and Haluk Beyenal. 2008. Batteryless, Wireless Sensor Powered by a Sediment Microbial Fuel Cell. *Environmental Science & Technology* 42, 22 (Oct. 2008), 8591–8596.
[22] EnOcean. 2018. EnOcean Wall Mounted Occupancy Sensor. https://www.enocean.com. (April 2018). Last accessed: Oct. 27, 2020.
[23] Micro:bit Educational Foundation. 2016. BBC micro:bit. https://www.microbit.org. (Feb. 2016). Last accessed: Oct. 27, 2020.
[24] Jon Froehlich, Leah Findlater, Marilyn Ostergren, Solai Ramanathan, Josh Peterson, Inness Wragg, Eric Larson, Fabia Fu, Mazhengmin Bai, Shwetak N. Patel1, and James A. Landay. 2012. The Design and Evaluation of Prototype Eco-feedback Displays for Fixture-level Water Usage Data. In *Proc. CHI* (May 5–10). ACM, Austin, TX, USA, 2367–2376.
[25] Damien P. George. 2019. MicroPython Home Page. https://micropython.org. (Sept. 2019). Last accessed: Oct. 27, 2020.
[26] Tobias Grosse-Puppendahl, Steve Hodges, Nicholas Chen, John Helmes, Stuart Taylor, James Scott, Josh Fromm, and David Sweeney. 2016. Exploring the Design Space for Energy-Harvesting Situated Displays. In *Proc. USIT* (Oct. 16–19). ACM, Tokyo, Japan, 41–48.
[27] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In *Proc. SIGCSE* (March 6–9). ACM, Denver, CO, USA, 579–584.
[28] Carl Hartung, Richard Han, Carl Seielstad, and Saxon Holbrook. 2006. FireWxNet: A Multi-tiered Portable Wireless System for Monitoring Weather Conditions in Wildland Fire Environments. In *Proc. MobiSys* (June 19–22). ACM, Uppsala, Sweden, 28–41.
[29] Sara Heitlinger, Nick Bryan-Kinns, and Rob Comber. 2019. The Right to the Sustainable Smart City. In *Proc. CHI* (May 4–9). ACM, Glasgow, Scotland, UK, 317:1–317:13.
[30] Josiah Hester, Timothy Scott, and Jacob Sorber. 2014. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-Harvesting Sensors. In *Proc. SenSys* (Nov. 3–5). ACM, Memphis, TN, USA, 330–331.
[31] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things. In *Proc. SenSys* (Nov. 6–8). ACM, Delft, The Netherlands, 19:1–19:13.
[32] Josiah Hester and Jacob Sorber. 2017. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proc. SenSys* (Nov. 6–8). ACM, Delft, The Netherlands, 21:1–21:6.
[33] Josiah Hester and Jacob Sorber. 2019. Batteries not Included. *XRDS: Crossroads, The ACM Magazine for Students* 26, 1 (2019), 23–27.
[34] Josiah Hester and Lanny Sitanayah Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *Proc. SenSys* (Nov. 1–4). ACM, Seoul, South Korea, 5–16.
[35] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely Execution on Intermittently Powered Batteryless Sensors. In *Proc. SenSys* (Nov. 6–8). ACM, Delft, The Netherlands, 17:1–17:13.
[36] Jason Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy. 2004. The Platforms Enabling Wireless Sensor Networks. *Commun. ACM* 47, 6 (June 2004), 41–46.
[37] Jennifer Jacobs and Leah Buechley. 2013. Codeable Objects: Computational Design and Digital Fabrication for Novice Programmers. In *Proc. CHI* (Apr. 27 – May. 2). ACM, Paris, France, 1589–1598.
[38] Junsu Jang and Fadel Adib. 2019. Underwater Backscatter Networking. In *Proc. SIGCOMM* (Aug. 19–24). ACM, Beijing, China, 187–199.
[39] Xiaofan Jiang, Joseph Polastre, and David Culler. 2005. Perpetual Environmentally Powered Sensor Networks. In *Proc. IPSN* (April 24–27). ACM/IEEE, Los Angeles, CA, USA, 1–12.
[40] Pouya Kamalinejad, Chinmaya Mahapatra, Zhengguo Sheng, Shahriar Mirabbasi, Victor C.M. Leung, and Yong Liang Guan. 2015. Wireless Energy Harvesting for Internet of Things. *IEEE Commun. Mag.* 53, 6 (June 2015), 102–108.
[41] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-oriented Live Programming Environment with Always-on Run-time Value Visualizations. In *Proc. USIT* (Oct. 22–25). ACM, Québec City, QC, Canada, 737–745.
[42] Hyung-Sin Kim, Michael P. Andersen, Kaifei Chen, Sam Kumar, William J. Zhao, Kevin Ma, and David E. Culler. 2018. System Architecture Directions for Post-SoC/32-bit Networked Sensors. In *Proc. SenSys* (Nov. 4–7). ACM, Shenzhen, China, 264–277.
[43] Bran Knowles, Lynne Blair, Mike Hazas, and Stuart Walker. 2013. Exploring Sustainability Research in Computing: Where we Are and Where we go Next. In *Proc. UbiComp* (Sept. 8–12). ACM, Zurich, Switzerland, 305–314.

[44] Vito Kortbeek, Kasım Sinan Yıldırım, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. 2020. Time-sensitive Intermittent Computing Meets Legacy Software. In *Proc. ASPLOS* (March 16–20). ACM, Lausanne, Switzerland, 85–99.
[45] Stacey Kuznetsov and Eric Paulos. 2010. UpStream: Motivating Water Conservation with Low-cost Water Flow Sensing and Persuasive Displays. In *Proc. CHI* (April 10–15). ACM, Atlanta, GA, USA, 1851–1860.
[46] Koen Langendoen, Aline Baggio, and Otto Visser. 2006. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. In *Proc. International Parallel & Distributed Processing Symposium* (April 25–29). IEEE, Rhodes Island, Grece, 1–12.
[47] Gierad Laput, Yang Zhang, and Chris Harrison. 2017. Synthetic Sensors: Towards General-purpose Sensing. In *Proc. CHI* (May 6–11). ACM, Denver, CO, USA, 3986–3999.
[48] Dominique Larcher and Jean-Marie Tarascon. 2015. Towards Greener and More Sustainable Batteries for Electrical Energy Storage. *Nature Chemistry* 7, 19 (Jan. 2015), 19–29.
[49] Cheng-Ting Lee, Yun-Hao Liang, Pai H. Chou, Ali Heydari Gorji, Seyede Mahya Safavi, Wen-Chan Shih, and Wen-Tsuen Chen. 2018. EcoMicro: A Miniature Self-Powered Inertial Sensor Node Based on Bluetooth Low Energy. In *Proc. ISLPED* (July 23–25). ACM, Seattle, WA, USA, 30:1–30:6.
[50] Edward A. Lee, John D. Kubiatowicz, Jan M. Rabaey, Alberto L. Sangiovanni-Vincentelli, Sanjit A. Seshia, John Wawrzynek, David Blaauw, Prabal Dutta, Kevin Fu, Carlos Guestrin, Roozbeh Jafari, Doug Jones, Vijay Kumar, and Richard Murray. 2012. *The Terraswarm Research Center (TSRC): a White Paper*. Technical Report. University of California, Berkeley. Tech. Rep. UCB/EECS-2012-207.
[51] Seulki Lee, Bashima Islam, Yubo Luo, and Shahriar Nirjon. 2019. Intermittent Learning: On-Device Machine Learning on Intermittently Powered System. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 3, 4 (Dec. 2019), 141:1–141:30.
[52] Hanchuan Li, Eric Brockmeyer, Elizabeth J. Carter, Josh Fromm, Scott E. Hudson, Shwetak N. Patel, and Alanson Sample. 2016. PaperID: A Technique for Drawing Functional Battery-Free Wireless Interfaces on Paper. In *Proc. CHI* (May 7–12). ACM, San Jose, CA, USA, 5885–5896.
[53] Yichen Li, Tianxing Li, Ruchir A. Patel, Xing-Dong Yang, and Xia Zhou. 2018. Self-Powered Gesture Recognition with Ambient Light. In *Proc. USIT* (Oct. 14–17). ACM, Berlin, Germany, 595–608.
[54] Rong-Hao Liang, Meng-Ju Hsieh, Jheng-You Ke, Jr-Ling Guo, and Bing-Yu Chen. 2018. RFIMatch: Distributed Batteryless Near-Field Identification Using RFID-Tagged Magnet-Biased Reed Switches. In *Proc. USIT* (Oct. 14–17). ACM, Berlin, Germany, 473–483.
[55] Silvia Lindtner, Garnet Hertz, and Paul Dourish. 2014. Emerging Sites of CHI Innovation: Hackerspaces, Hardware Startups & Incubators. In *Proc. CHI* (Apr. 26 – May 1). ACM, Toronto, ON, Canada, 439–448.
[56] Szu-Yu (Cyn) Liu, Shaowen Bardzell, and Jeffrey Bardzell. 2019. Symbiotic Encounters: HCI and Sustainable Agriculture. In *Proc. CHI* (May 4–9). ACM, Glasgow, Scotland, UK, 317:1–317:13.
[57] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. In *Proc. SNAPL* (May 7–10). ACM, Alisomar, CA, USA, 8:1–8:14.
[58] Brandon Lucia and Benjamin Ransford. 2015. A simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proc. PLDI* (Aug. 13–17). ACM, Portland, OR, USA, 575–585.
[59] Yunfei Ma, Zhihong Luo, Christoph Steiger, Giovanni Traverso, and Fadel Adib. 2018. Enabling Deep-Tissue Networking for Miniature Medical Devices. In *Proc. SIGCOMM* (Aug. 20–25). ACM, Budapest, Hungary, 417–431.
[60] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. In *Proc. OOPSLA* (Oct. 22–27). ACM, Vancouver, BC, Canada, 96:1–96:30.
[61] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *Proc. OSDI* (Oct. 8–10). USENIX, Carlsbad, CA, USA, 129–144.
[62] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. 2002. Wireless Sensor Networks for Habitat Monitoring. In *Proc. International Workshop on Wireless Sensor Networks and Applications* (Sept. 28). ACM, Atlanta, GA, USA, 88–97.
[63] Clive Maxfield. 2016. Python is better than C! (Or is it the other way round?). https://www.embedded.com/python-is-better-than-c-or-is-it-the-other-way-round. (March 2016). Last accessed: Oct. 27, 2020.
[64] Will McGrath, Daniel Drew, Jeremy Warner, Majeed Kazemitabaar, Mitchell Karchemsky, David Mellis, and Björn Hartmann. 2017. Bifröst: Visualizing and Checking Behavior of Embedded Systems Across Hardware and Software. In *Proc. USIT* (Oct. 22–25). ACM, Québec City, QC, Canada, 299–310.
[65] William McGrath, Jeremy Warner, Mitchell Karchemsky, Andrew Head, Daniel Drew, and Bjoern Hartmann. 2018. WiFröst: Bridging the Information Gap for Debugging of Networked Embedded Systems. In *Proc. UIST* (Oct. 14–17). ACM, Berlin, Germany, 447–455.
[66] Microchip. 2017. MIC841/2 Comparator with 1.25% Reference and Adjustable Hysteresis. http://ww1.microchip.com/downloads/en/DeviceDoc/20005758A.pdf. (April 2017). Last accessed: Oct. 27, 2020.
[67] Microsoft. 2020. MakeCode: Hands on computing education. https://www.microsoft.com/en-us/makecode. (2020). Last accessed: Oct. 27, 2020.
[68] MIT Media Lab. 2002. Scratch Programming Language Official Website. https://scratch.mit.edu. (2002). Last accessed: Oct. 27, 2020.
[69] Iqbal Mohomed and Prabal Dutta. 2014. The Age of DIY and Dawn of the Maker Movement. *GetMobile* 18, 4 (Oct. 2014), 41—43.

[70] NeDRo. 2020. 6 V 0.6 W 80×55 mm Mini Solar Panel. https://etronixcenter.com/en/solar-panels-and-wind-turbines/8168876-al103-nedro-6v-06w-80x55mm-mini-solar-panel-7110218865414.html. (Aug. 2020). Last accessed: Oct. 27, 2020.
[71] NXP. 2015. User Manual for NXP Real Time Clocks PCF85x3, PCF85x63, PCA8565, PCF2123, and PCA21125. https://www.nxp.com/docs/en/user-guide/UM10301.pdf. (July 2015). Last accessed: Oct. 27, 2020.
[72] Stephen O'Grady. 2019. The RedMonk Programming Language Rankings: June 2019. https://redmonk.com/sogrady/2019/07/18/language-rankings-6-19. (July 2019). Last accessed: Oct. 27, 2020.
[73] Joseph A. Paradiso and Thad Starner. 2005. Energy Scavenging for Mobile and Wireless Electronics. *IEEE Pervasive Comput.* 4, 1 (Jan.–Mar. 2005), 18–27.
[74] Matthai Philipose, Joshua R. Smith, Bing Jiang, Alexander Mamishev, Sumit Roy, and Kishor Sundara-Rajan. 2005. Battery-Free Wireless Identification and Sensing. *IEEE Pervasive Comput.* 4, 1 (Jan.–Mar. 2005), 37–45.
[75] James Pierce and Eric Paulos. 2010. Materializing Energy. In *Proc. DIS* (Aug. 16–20). ACM, Aarhus, Denmark, 113–122.
[76] James Pierce and Eric Paulos. 2012. Beyond Energy Monitors: Interaction, Energy, and Emerging Energy Systems. In *Proc. CHI* (May 5–10). ACM, Austin, TX, USA, 2367–2376.
[77] Joseph Polastre, Robert Szewczyk, and David Culler. 2005. Telos: Enabling Ultra-low Power Wireless Research. In *Proc. IPSN* (April 24–27). ACM/IEEE, Los Angeles, CA, USA, 1–12.
[78] R. Venkatesha Prasad, Shruti Devasenapathy, Vijay S. Rao, and Javad Vazifehdan. 2014. Reincarnation in the Ambiance: Devices and Networks with Energy Harvesting. *IEEE Commun. Surveys Tuts.* 11, 1 (First Quarter 2014), 195–213.
[79] Jothi Prasanna Shanmuga Sundaram, Wan Du, and Zhiwei Zhao. 2020. A Survey on LoRa Networking: Research Problems, Current Solutions, and Open Issues. *IEEE Commun. Surveys Tuts.* 22, 1 (First Quarter 2020), 371–388.
[80] Lutz Prechelt. 2000. An Empirical Comparison of Seven Programming Languages. *Computer* 33, 10 (Oct. 2000), 23–29.
[81] Damith C. Ranasinghe, Roberto L. Shinmoto Torres, Alanson P. Sample, Joshua R. Smith, Keith Hill, and Renuka Visvanathan. 2012. Towards Falls Prevention: a Wearable Wireless and Battery-less Sensing and Automatic Identification-Tag for Real Time Monitoring of Human Movements. In *Proc. EMBC* (Aug. 28 – Sep. 1). IEEE, San Diego, CA, USA, 6402–6405.
[82] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. In *Proc. ASPLOS* (March 5–11). ACM, Newport Beach, CA, USA, 159–170.
[83] Saul Rodriguez, Stig Ollmar, Muhammad Waqar, and Ana Rusu. 2016. A Batteryless Sensor ASIC for Implantable Bio-impedance Applications. *IEEE Trans. Biomed. Circuits Syst.* 10, 3 (June 2016), 533–544.
[84] Paulo Rosa, Federico Ferretti, Ângela Guimarães Pereira, Francesco Panella, and Maximilian Wanner. 2017. *Overview of the Maker Movement in the European Union*. Technical Report. European Union. https://core.ac.uk/download/pdf/132627140.pdf.
[85] Kimiko Ryokai, Peiqi Su, Eungchan Kim, and Bob Rollins. 2014. EnergyBugs: Energy Harvesting Wearables for Children. In *Proc. CHI* (Apr. 26 – May 1). ACM, Toronto, ON, Canada, 1039–1048.
[86] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. 2008. Design of an RFID-based battery-free programmable sensing platform. *IEEE Trans. Instrum. Meas.* 57, 11 (Nov. 2008), 2608–2615.
[87] Philip Sparks. 2017. *The Route to a Trillion Devices: The Outlook for IoT investment to 2035*. Technical Report. ARM Limited. https://pages.arm.com/rs/312-SAX-488/images/Arm-The-route-to-trillion-devices_2018.pdf.
[88] Andrew Spielberg, Alanson Sample, Scott E. Hudson, Jennifer Mankoff, and James McCann. 2016. RapID: A Framework for Fabricating Low-Latency Interactive Objects with RFID Tags. In *Proc. CHI* (May 7–12). ACM, San Jose, CA, USA, 5897–5908.
[89] Evan Strasnick, Maneesh Agrawala, and Sean Follmer. 2017. Scanalog: Interactive Design and Debugging of Analog Circuits with Programmable Hardware. In *Proc. USIT* (Oct. 22–25). ACM, Québec City, QC, Canada, 321–330.
[90] Valerie Sugarman and Edward Lank. 2015. Designing Persuasive Technology to Manage Peak Electricity Demand in Ontario Homes. In *Proc. CHI* (April 18–23). ACM, Seoul, Republic of Korea, 1975–1984.
[91] Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua R. Smith. 2017. Battery-Free Cellphone. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 2 (June 2017), 25:1–25:20.
[92] Jethro Tan, Przemysław Pawełczak, Aaron Parks, and Joshua R. Smith. 2016. Wisent: Robust Downstream Communication and Storage for Computational RFIDs. In *Proc. INFOCOM* (April 10–15). IEEE, San Francisco, CA, USA, 1–9.
[93] Texas Instruments Inc. 2015. TLV36910.9-V to 6.5-V, Nanopower Comparator. https://www.ti.com/lit/ds/symlink/tlv3691.pdf. (Nov. 2015). Last accessed: Oct. 27, 2020.
[94] Texas Instruments Inc. 2017. MSP430FR59xx Mixed-Signal Microcontrollers (Rev. F). http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf. (March 2017). Last accessed: Oct. 27, 2020.
[95] The Economist. 2018. And Now for Something Completely Different. https://www.economist.com/science-and-technology/2018/07/19/python-has-brought-computer-programming-to-a-vast-new-audience. (July 2018). Last accessed: Oct. 27, 2020.
[96] TU Delft Sustainable Systems Lab. 2020. BFree GitHub Repository. https://github.com/tudssl/bfree. (Oct. 2020). Last accessed: Oct. 27, 2020.
[97] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proc. OSDI* (Nov. 2–4). ACM, Savannah, GA, USA, 17–32.

[98] Matt Weinberger. 2018. How one Woman Turned her Passion for Tinkering into a $33 Million Business—without a Dime of Funding. https://www.businessinsider.com/adafruit-industries-limor-fried-on-bootstrapping-a-startup-2015-8. (Aug. 2018). Last accessed: Oct. 27, 2020.

[99] WEMOS Electronics. 2019. ePaper 2.13 Shield. https://www.wemos.cc/en/latest/d1_mini_shiled/epd_2_13.html. (2019). Last accessed: Oct. 27, 2020.

[100] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. 2006. Fidelity and Yield in a Volcano Monitoring Sensor Network. In *Proc. OSDI* (Nov. 6–8). USENIX, Seattle, WA, USA, 381âĂŞ–396.

[101] Jeongjin Yeo, Mun ho Ryu, and Yoonseok Yang. 2015. Energy Harvesting from Upper-limb Pulling Motions for Miniaturized Human-powered Generators. *Sensors* 15, 7 (2015), 15853–15867.

[102] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemysław Pawełczak, and Josiah Hester. 2018. InK: Reactive kernel for tiny batteryless sensors. In *Proc. SenSys* (Nov. 4–7). ACM, Shenzhen, China, 41–53.

[103] G. Pascal Zachary. 2016. The Search for a Better Battery. https://spectrum.ieee.org/at-work/innovation/the-search-for-a-better-battery. (April 2016). Last accessed: Oct. 27, 2020.