TU Delft

Delft University of Technology

DIPS: Debug Intermittently-Powered Systems Like Any Embedded System

de Winkel, J.; Hoefnagel, T.S.; Blokland, B.T.; Pawełczak, P.

DOI
10.1145/3560905.3568543

Publication date
2022

Document Version
Final published version

Published in
SenSys 2022 - Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems

Citation (APA)
de Winkel, J., Hoefnagel, T. S., Blokland, B. T., & Pawełczak, P. (2022). DIPS: Debug Intermittently-Powered Systems Like Any Embedded System. In *SenSys 2022 - Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems* (pp. 222–235). (SenSys 2022 - Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems). ACM. https://doi.org/10.1145/3560905.3568543

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

DIPS: Debug Intermittently-Powered Systems Like Any Embedded System

Jasper de Winkel
Delft University of Technology
Delft, The Netherlands
j.dewinkel@tudelft.nl

Tom Hoefnagel
Delft University of Technology
Delft, The Netherlands
t.s.hoefnagel@student.-tudelft.nl

Boris Blokland
Delft University of Technology
Delft, The Netherlands
b.t.blokland@student.-tudelft.nl

Przemysław Pawełczak
Delft University of Technology
Delft, The Netherlands
p.pawelczak@tudelft.nl

ABSTRACT

Debugging and testing battery-free intermittently-powered systems is notoriously difficult. This is not only due to the additional complexity of maintaining state through power failures but also due to the lack of proper tools to test and debug these systems. As a solution, we present DIPS: a fully-featured hardware debugger for battery-free intermittently-powered systems capable of automatically verifying memory and peripheral state between power failures. Our solution seamlessly integrates an emulator allowing for emulation of any power scenario to the device under test. This allows our debugger to pause emulation and program execution when debugging or when state restoration issues are detected. Our new system is built around GNU Debugger (GDB): a widely-used debugging tool. Therefore, DIPS allows for a debugging process identical to state-of-the-art debuggers for continuously-powered devices. User studies found that our debugger is easy and intuitive to use. It allows embedded system developers to find bugs quicker in code written for battery-free devices. With our debugger we found unseen errors in state-of-the-art software frameworks for intermittently-powered systems.

CCS CONCEPTS

• **Hardware** → *Simulation and emulation*; *Analysis and design of emerging devices and systems*; • **Computer systems organization** → *Embedded systems*; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Intermittent Systems, Debugging, Software Testing, Emulation

ACM Reference Format:
Jasper de Winkel, Tom Hoefnagel, Boris Blokland, and Przemysław Pawełczak. 2022. DIPS: Debug Intermittently-Powered Systems Like Any Embedded System. In *The 20th ACM Conference on Embedded Networked Sensor Systems (SenSys '22), November 6–9, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3560905.3568543

This work is licensed under a Creative Commons Attribution International 4.0 License.
SenSys '22, November 6–9, 2022, Boston, MA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9886-2/22/11.
https://doi.org/10.1145/3560905.3568543

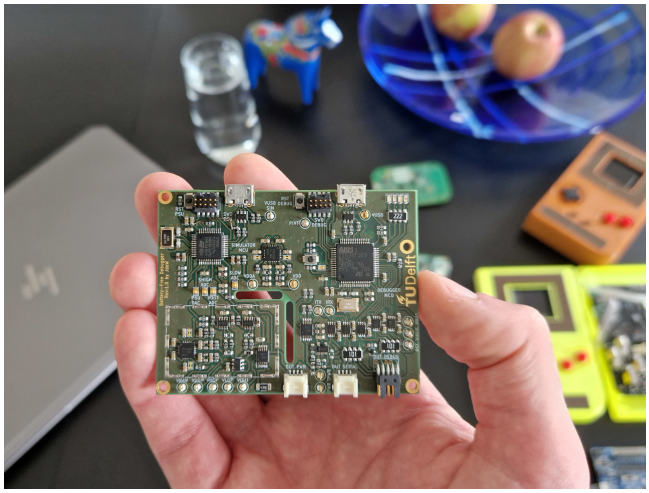


Figure 1: Photography of Debugger for Intermittently-Powered Systems (DIPS) hardware. DIPS is a new hardware/software ecosystem specifically designed for debugging and testing intermittently-powered battery-free devices.

1 INTRODUCTION

Intermittently-powered devices [27] are a new class of low-power—often battery-free [46]—embedded systems that can guarantee correct and forward-progressing computation despite frequent power interrupts. These interrupts are caused by the incoming energy from ambient (therefore intermittent) energy sources that power the device. That incoming energy charges a small energy storage, i.e. (super-)capacitor, that cannot buffer incoming energy as much as the battery, elevating intermittency rate further. Despite this inconvenience, the benefit of using intermittently-powered devices instead of 'classical' battery-based ones is twofold. First, removal of a battery creates a more environmentally-friendly device and powering embedded systems from ambient sources is sustainable [17]. Second, battery-free operation promises perpetual operation: as long as there is an ambient energy source, battery-free devices will continue operating [40]. These advantages lead to battery-free intermittently-powered operation being applied to many embedded applications. These include a battery-free handheld gaming console [11], battery-free computational Radio Frequency Identification (RFID) tags [44], battery-free sensors [10] and sensor networks [3], battery-free eye tracker [26], as well as battery-free edge computing platform [33].

Unfortunately, despite the increasing number of battery-free intermittently-powered platforms, they are still hard to program [23, Section 7]. This difficulty stems from ensuring correct continuation of program execution after a restart from a power interrupt. The application developer must programmatically account for two events. That is, whenever a power interrupt happened, at any place in the code, the device (i) must resume operation from the moment that power interrupt happened, and (ii) the state of the device's memory and its peripherals must be correctly restored. To assure that event (i) and (ii) happens the programmer instruments the code by means of two approaches. The first one is the inclusion of checkpoints that force saving of program state from volatile to non-volatile memory (examples of such approaches include [24, 56]). The second one being code transformations, where code is divided into atomic blocks (in the context of intermittently-powered devices called tasks [28] or threads [59]) whose execution time matches the specific energy budget of the intermittently-powered device. Whichever of the two methods of code instrumentation has been chosen (either being compiler-supported or requiring programmer labour) the code for intermittently-powered device needs to be debugged during development. Sadly, debugging of software written for battery-free intermittently-powered devices is itself hard [7, Section 2.2]. This is because above the existence of 'normal' bugs (not related to intermittently-powered operation) one has to additionally deal with bugs resulting from these power failures. Unfortunately, the debuggers developed for battery-powered embedded systems, such as [45], assume the target device/Device Under Test (DUT) is continuously powered in order to debug. This effectively removes the ability of code debugging, as with every power failure the debugger has to be reconnected manually.

To the best of our knowledge there is only one dedicated debugger targeting intermittently-powered devices, i.e. EDB [7] that addresses some of the core limitations of existing debuggers for embedded systems. Nonetheless, to debug code with EDB one has to instrument the code manually with EDB-specific API for software based assertions and breakpoints. This results in a time-consuming debug process, as for each new assertion or breakpoint the code must be recompiled and the bug scenario has to be recreated. Then, each breakpoint has to be manually enabled when starting the debugging session. When an assertion is triggered, each variable responsible for triggering the assertion has to be individually investigated by first looking up the address of the variable and then reading the memory at that address. This does not allow the user to (i) easily inspect all memory variables or the call stack in a breakpoint or (ii) transitions from one task to another. But what is more important, EDB breakpoints themselves might mask intermittency-specific code bugs—as we will show later in this paper—which is detrimental to the debugging process.

Furthermore, EDB does not allow for replay of energy traces powering the battery-free device. Instead, EDB makes sure that the device storage capacitor is charged from the instrumented assertion/breakpoint to keep the device alive, discharging it after the assertion as if no assertion was included in the code. This allows for code checking without an energy penalty to the device. Energy trace replay, however, would allow for repeatable results and the ability to induce time-specific power interrupts. This is unfortunately impossible with EDB-style debugging where the DUT is

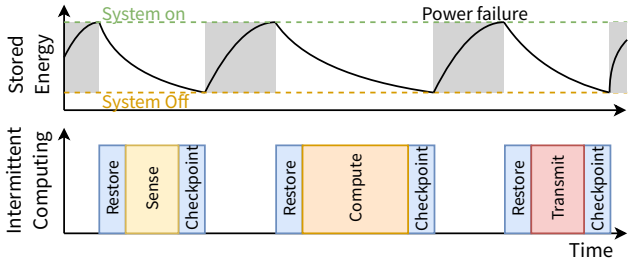


Figure 2: Schematic representation of intermittent operation: periods of system 'on' state are intervened by periods of 'power failure' state (when the system's capacitor is being re-charged). The power intermittency is caused by the unpredictable nature of ambient harvested energy. Therefore, intermittently-powered devices need to checkpoint and restore the intermediate state to and from non-volatile memory to guarantee forward progress despite power interrupts.

powered from uncontrollable energy harvesting sources (in the case of EDB—an RFID transmitter).

To solve the debugging problem for intermittently-powered devices our idea is to bring two necessary embedded debugging components together in a single debugging platform. These components being: (i) a fully featured hardware debugger based on GNU Debugger (GDB) [39, 48]—to enable step-by-step debugging in the way the majority of existing (non-intermittently-powered) embedded platforms are being debugged right now, and (ii) an energy emulator capable of replaying energy traces, allowing to power battery-free platforms from the same energy trace (either pre-recorded or synthetically generated) repeatably—to emulate specific intermittency patterns, such as in [12]. The result is a new debugging platform named Debugger for Intermittently-Powered Systems (DIPS), as shown in Figure 1.

The contributions presented in our paper are as follows.

- **Hardware-based debugging on intermittently-powered systems**: We introduce the first hardware based debugger for intermittently-powered systems, capable of utilizing the hardware debugging features of the microcontroller under test, despite being intermittently-powered. Our debugger does not require any software modification to the Device Under Test (DUT) and is based on GDB, resulting in direct integration into most Integrated Developer Environments (IDEs). This allows for rapid debugging of code for intermittently-powered devices—in an identical fashion compared to battery-based (classical) embedded devices. This observation is echoed by the user experience study of DIPS vis-a-vis state-of-the-art debugger: EDB [7].
- **Tightly-coupled energy emulator**: Unlike other systems our emulator tightly interconnects with the debugger and pauses emulation when, e.g. breakpoints are triggered, keeping the DUT powered and seamlessly resuming emulation after the user resumes execution. Our emulator is not only capable of providing synthetic test patterns to power the DUT but is also capable of mimicking the power supply circuit

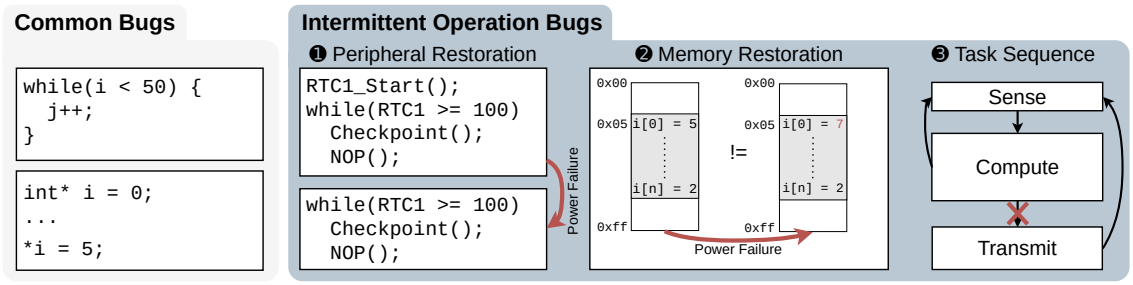


Figure 3: Two major classes of code bugs are present in intermittently-powered battery-free systems: common bugs such as using the wrong iterator or writing to null pointers and bugs related to intermittent operation. These include peripheral restoration, memory restoration and task sequence bugs.

commonly used in state-of-the-art intermittently-powered systems: the buck-boost converter and the storage (super-) capacitor. This allows for easy and quick experimentation to determine the energy input requirements of the DUT and to find an optimal capacitor size for the system. For the first time, these features allow the developers to debug and test energy-related bugs in a repeatable fashion.

- **Automated testing for intermittency-related bugs**: Intermittent systems rely on saving and restoring the state of the system to a non-volatile on-board memory. Using an automated scripting framework we are able to verify if the volatile memory of the intermittently-powered device has correctly been restored from the last checkpoint. Not only volatile memory consistency is automatically checked using DIPS but also peripheral state is verified by comparing peripheral configuration registers prior to a checkpoint and post restoration.

All hardware, software and tools pertaining to DIPS, together with its documentation, will be made available open-source to the research community via our artifact [1]. We will also provide fully assembled and calibrated DIPS boards to the community. We envision DIPS becoming the de facto standard debugging tool for intermittently-powered systems, simplifying testing and debugging of these novel embedded systems.

2 INTERMITTENTLY-POWERED SYSTEMS: BACKGROUND

Battery-free embedded systems forgo batteries where the energy reservoir is replaced by a (super-)capacitor. Capacitors are cheaper, often smaller and less polluting than conventional batteries [54]. These advantages are powerful considering the already-massive (and growing) number of Internet of Things (IoT) devices [55]—think of labor and monetary costs of monitoring this magnitude of batteries worldwide. Such battery-free systems are powered by ambient energy (like solar radiation, vibrations, electromagnetic waves, wind or temperature difference) [41].

The small energy density of a super-capacitor and unpredictable nature of harvested energy makes operation of battery-free system often intermittent, see Figure 2—periods of useful energy availability are intervened by intervals of power unavailability. This means that tasks performed by an embedded device, such as 'sense' (signals from the environment), 'compute' (process collected data locally) and 'transmit' (transmit processed data to the central point) would have to be 'sliced' in time in order to complete them. Therefore state of the embedded device (volatile memory with its all registers and peripherals) needs to be saved regularly in a non-volatile memory to protect from state loss and from complete system restart from an initial state. This way battery-free system continues to work perpetually, as long as ambient energy is present [31, Section II-B].

As a consequence of intermittently-powered operation the research on such systems has focused on hardware and software support that perform operations of device state storing and restoring correct and fast. Some of the relevant works include [24, 25, 28, 51, 56, 58, 59]. These frameworks lead to building increasingly complex battery-free systems, including, battery-free gaming platforms [11], battery-free hobbyist platforms [23], battery-free small robots [58] and battery-free embedded prototyping platforms [16].

Many of these frameworks perform transformations on the existing (mostly C) code—either by introducing memory checkpoints at predefined locations in the code, e.g., [24, 25, 56] or transforming the code into a state machine-like structure, e.g. using tasks [28, 51, 58] or threads [59]. These transformations increase code complexity and might introduce new type of errors. Therefore, as with any other code written for embedded systems these supporting frameworks (and the output code that results from them) need to be developed, debugged and then tested. However, debugging of code written for intermittently-powered devices poses unique challenges.

3 DEBUGGING INTERMITTENTLY-POWERED SYSTEMS

Debugging of embedded systems code is different from PC-based code debugging [4, Chapter 8]. As PC-based code can mostly be directly debugged using tools such as GDB in an IDE as it runs on the same device, unlike embedded systems where the code is running on an external embedded system. Therefore, the developer must rely on external hardware—such as [45]—acting as an interface between GDB and the Microcontroller (MCU) on-board debug hardware.

3.1 Bugs Type Classification

We can categorise bugs present in the software for intermittently-powered devices into two classes presented in Figure 3. First class

Listing 1: Masked Write After Read (WAR) error due to breakpoint insertion (listing (b)). When function calls are instrumented as checkpoints such as in [24], the addition of software-based breakpoints as required by the EDB [7] debugger can mask WAR-related errors, see listing (a), since the breakpoint itself will be instrumented with a checkpoint. nv_x refers to a variable stored in non-volatile memory.

(a) WAR error

```
1 Checkpoint()
2 y = nv_x // wrong
3 // after restart
4 z = y + 1
5
6 nv_x=z
7 # Power failure
8 ...
9 Checkpoint()
```

(b) Masked WAR error

```
1 Checkpoint()
2 y = nv_x // correct
3 // after restart
4 z = y + 1
5 EDB_Breakpoint(0)
6 nv_x = z
7 # Power failure
8 ...
9 Checkpoint()
```

Table 1: Feature comparison of DIPS (i.e. this work) against EDB [7]—debugger for intermittently-powered battery-free embedded systems and J-link [45]—popular debugger for battery-based embedded systems.

Feature	EDB	J-Link	DIPS
Energy breakpoints	Yes ✓	No ✗	Yes ✓
Software breakpoints	Yes ✓	Yes ✓	Yes ✓
Hardware breakpoints	No ✗	Yes ✓	Yes ✓
Single step	No ✗	Yes ✓	Yes ✓
Watchpoints	Yes ✓	Yes ✓	Yes ✓
GDB support	No ✗	Yes ✓	Yes ✓
IDE support	No ✗	Yes ✓	Yes ✓
ARM support	No ✗	Yes ✓	Yes ✓
MSP430 support	Yes ✓	No ✗	Pending
Software testing	No ✗	No ✗	Yes ✓
Energy trace emulation	No ✗	No ✗	Yes ✓

are the common programming language and embedded system-related bugs. Example of such bugs are algorithm implementation bugs (for example increment of the wrong variable in a while loop `while (i < 50){j++;}`) or code errors in embedded system-related functionalities (for example, inaccurate peripheral initialisation). These bugs are extensively analysed since the dawn of programming languages and will not be discussed here. The second class are the intermittent operation-related bugs and these are the ones which the designed debugger will specifically target.

We can further categorise intermittent operation bugs into: (i) peripheral restoration bugs, (ii) memory restoration bugs, and (iii) task sequence bugs (see again Figure 3).

(1) **Peripheral restoration bugs** occur due to inaccurate reinitialization after checkpoint restoration, as seen in Figure 3. In this example: after the power failure the program restarts from within the while loop without reinitializing the peripheral causing an infinite loop from a not re-initiated Real Time Clock (RTC). Peripheral bugs also occur when the state of any external peripherals such as displays, sensors and radios is not carefully considered, especially when these peripherals have persistent state or are continuously powered. Examples of such bugs include: (i) persistent configuration registers where the process of configuring the register could not be confirmed due to a power failure, (ii) a failure to gracefully power down an E-Ink display resulting in a faded background, and (iii) synchronization issues where the external peripherals state is not aligned with the expected state.

(2) **Memory restoration bugs** come from errors in the checkpoint process. In the first place they can come from the wrong placement of checkpoints (function `Checkpoint()`), as illustrated in Listing 1 (a) where a write after read error occurs due to the lack of a checkpoint in-between reading from and writing to non-volatile memory. But the implementation of a checkpoint itself can also contain bugs. For example, checkpointing is often based on double buffering (such as in [23] where the whole memory is checkpointed to a non-volatile memory at a predefined time interval), where usually a binary flag specifies to which memory region a checkpoint needs to be stored and from which region data needs to be restored. If a power failure happens at the moment of the flag update, the checkpoint will be corrupted. The more complicated checkpointing routines, like differential-checkpointing of [11] where the only changed memory regions since the last checkpoint are checkpointed, or undo logging-based checkpointing as used in [25]—the higher the probability of error in the implementation of checkpoint.

(3) **Task sequence bugs** are specific to special type of runtime systems for intermittently-powered devices where input code is transformed into tasks (such as 'Sense', 'Compute' and 'Transmit') and checkpointing is performed always at the task transition. Examples of such systems include InK [58], Alpaca [28] or ImmortalThreads [59]. Bugs can not only occur with incorrect implementation of the task state machine (as in case of example in Figure 3 'Compute' task connects back to 'Sense' instead of 'Transmit'). Bugs can also occur when defining the volatile memory associated with each task—if not accurately defined it could result in writing to and reading from unrestored memory.

3.2 Why Debugging Intermittently-Powered Systems is Still Hard

We need a dedicated debugger that would aid in spotting all errors shown in Figure 3 and EDB [7] was the first one that addressed this need. EDB introduced new debugging features, as listed in Table 1. Sadly, in special cases EDB can hinder bug finding. As we mentioned in Section 1 EDB debugger [7] inspects the code by inserting software based assertion flags and breakpoints (to trace potential intermittent operation-related bugs). These however might mask the write after read bugs, as shown Listing 1 (b), as software breakpoints are implemented as a function this might interact with compiler-based runtime systems [24] that apply compiler based optimizations and instrument each function. A software breakpoint

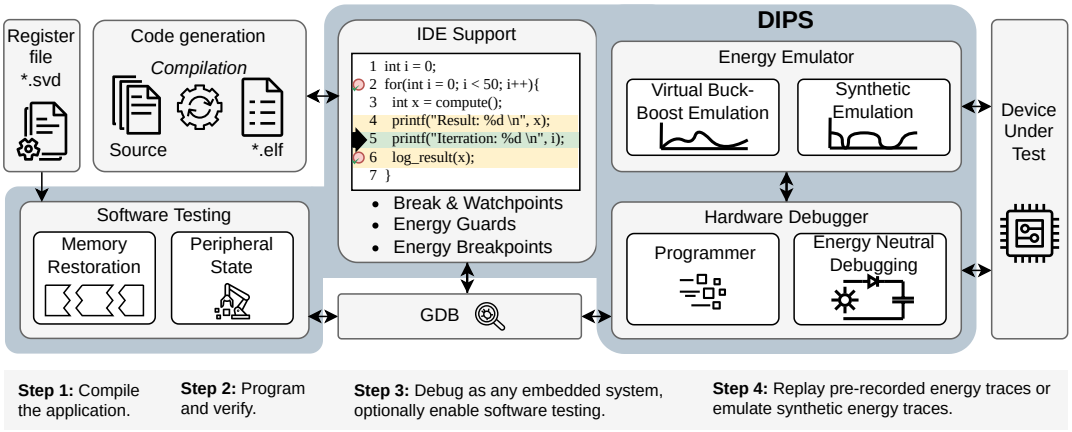


Figure 4: DIPS architecture and workflow, enabling seamless debugging of intermittent systems. In the code view marks an energy neutral section, marks the current line and hardware breakpoints are indicated by . Arrows in the figure denote information flow between individual blocks.

(`EDB_Breakpoint(0)`) would then result in an undesired checkpoint on the breakpoint location masking the write after read issue. A release build without the software debugging functions would then re-expose the hidden 'write after read bug'. Apart from introducing potentially unwanted checkpoints, software debugging calls could also prevent further compiler optimization such as loop unrolling.

For the record, one can consider using a CPU emulator, such as [37] as used in [24], as a replacement for EDB's inability to fulfill its debugging task completely. However, the timing of the instructions is not always perfect in emulation. Most importantly however, in most emulators the peripheral state is mocked—disallowing detection of peripheral-related bugs. Even with a cycle accurate emulation of the CPU and the associated peripherals forming the complete MCU, emulators do not emulate the starting sequence that occurs when power is applied to the actual chip. The CPU only starts execution after, e.g., voltage rails stabilize and stable clocks are present. These processes determine the start-up time and are subject to per component/design variation.

We thus conjecture that a debugger for intermittently-powered systems needs to have the same list of functionalities as debuggers for 'classical' embedded systems, e.g. [45], which are listed in Table 1. Moreover, it needs to support intermittently-powered systems specific features, which we denote as *energy breakpoints*, *software testing* and *energy trace emulation*. Inspecting Table 1 neither EDB, nor J-Link supports complete set of debugging features needed.

4 DIPS: DEBUGGER FOR INTERMITTENTLY-POWERED SYSTEMS

Driven by requirements listed in Table 1 we propose a new method of developing and testing battery-free intermittently-powered devices. These development and testing methods are implemented as a new debugger named DIPS. DIPS combines a hardware debugger (described in Section 4.1) and an energy emulator (described in Section 4.2), enabling seamless debugging of intermittently-powered systems. The energy emulator acts as a controllable power source capable of emulating intermittent operation to the Device Under Test (DUT). The architecture of DIPS is shown in Figure 4.

4.1 DIPS Hardware Debugger

A core part of debugging any embedded system is the hardware debugger. It interfaces with the DUT's MCU enabling the use of the MCU's debugging features. These features usually include (i) halting, (ii) reading and writing memory, (iii) setting breakpoints, and (iv) setting watchpoints.

As intermittently-powered systems switch on and off repeatedly, any state that is not specifically stored prior to a power failure is lost. This includes the configuration of the debugging registers. Even worse, these debugging registers are usually not configurable from within the MCU itself due to the security risk associated. Hence the hardware debugger must be able to quickly reconnect after power failures on intermittently-powered systems. To address this requirement DIPS keeps track of all debugging attributes, such as breakpoints and watchpoints, restoring those when the MCUs recovers from the power failure. To implement these features we have taken a popular open-source hardware debugger—the Black Magic Debug Probe [38]—as a base and build upon its functionality, adding the required features to debug and test intermittently-powered systems. Many popular MCUs are supported—for a full list please refer again to [38].

4.1.1 Energy Neutral Debugging. One core feature of DIPS is the energy isolated interface between DUT and DIPS. This allows DIPS to monitor the DUT whilst not interfering with the power consumption of the DUT. If the DUT is paused by any debugging action e.g., a breakpoint, the hardware debugger automatically pauses the energy emulator, making sure the DUT remains powered. When execution is resumed the energy emulator restores the energy state prior to the breakpoint and continues from where it paused.

We introduce two debugging modes with DIPS, (i) attached and (ii) detached, where each of them is described below. The hardware implementation of the energy isolation is further described in Section 4.4.

Table 2: DIPS extensions to the GDB Command Line Interface (CLI) implementing debugging functionality for intermittently-powered devices and an optional C language API for quick and simple debugging of intermittently-powered systems. An extended description is provided in [1].

GDB CLI	Description
`energy_breakpoint`	Defines a voltage-dependant breakpoint
`energy_guard`	Defines an energy neutral section
C API	**Description**
`DIPS_PRINTF`	Energy-neutral `printf`
`DIPS_ASSERT`	Halts code execution upon assertion
`DIPS_ATTACH`	Connect debugger (Detached mode)

Attached Debugging. In the attached debugging mode, the debugger reconnects to DUT after every power failure and any debugging attributes such as breakpoints are restored. When the DUT is connected to the hardware debugger, additional power will be consumed by the MCUs on-board debugging hardware. This is compensated for during emulation by measuring the power consumption at idle with and without the debugger attached. The attached mode gives most flexibility to the user as the intermittently-powered system appears as a normal embedded system to the developer, masking any effects of intermittency. We envision this mode to be used in a scenario of active software development and during preliminary testing/evaluation of intermittent systems. For example, during development of new checkpoint frameworks or when testing if peripheral configuration is correctly restored after power failures.

Detached Debugging. This mode is intended for when the DUT powers itself, for example by an external harvested energy source. In this mode the debugger only connects when it receives the hardware interrupt from the DUT, generated by e.g. the `DIPS_ATTACH` call to the C API listed in Table 2, if not already connected. When an interrupt is generated, the emulator takes over powering DUT. Next the debugger connects, allowing the debugger to interact with the DUT. After the user resumes code execution or when the debug operation finishes, the debugger detaches and the energy state of the DUT is restored to the level prior to intervention as further described in Section 4.2.1. This mode is intended for debugging scenario's close to final deployment where minimal interference is desired. More specifically, in scenarios where the device operates on its own, whilst still offering an option to debug the system when, for example, an assert fails.

4.1.2 Energy-Aware Debugging Features. Most of DIPS's debugging features utilize the build-in MCUs debugging hardware. Thus unlike the software-based debuggers DIPS does not require the usage of a specific API to debug the DUT. We extend GDB with the CLI commands listed in Table 2 to implement two key intermittent specific debugging functions: (i) energy breakpoints (`energy_breakpoint`)—extending traditional breakpoints by only triggering when the DUT's capacitor voltage is lower than the provided threshold, and (ii) energy neutral sections defined by energy guards (`energy_guard`)—allowing users to execute debug code whilst emulation is paused and steady power is provided.

Apart from the GDB extensions, we introduce a limited C API listed in Table 2, providing some optional convenience functions to the user. The function `DIPS_PRINTF` implements a print to the console. When `DIPS_ASSERT` parameters assert to false, code execution is halted. `DIPS_ATTACH` triggers the debugger to connect and halts execution until the debugger is connected.

All hardware debugging features and C API calls cause the emulator to pause whilst keeping the DUT powered. When normal code execution is resumed, the emulator also resumes. Compensating for the power consumption of the debug features itself.

Programmer. DIPS is also capable of programming/flashing of supported MCUs. The programming/flashing feature completes the all-in-one suite of features served by DIPS for the developer of intermittently-powered systems.

4.2 DIPS Energy Emulator

The second core part of our design is the energy trace emulator. State-of-the-art intermittently-powered systems harvest energy and store this energy into a (super-)capacitor. The voltage of this capacitor is often used as a threshold to determine when the voltage regulator powering the MCUs turns on and off. In this case the MCU only is provided a regulated supply that is switched on and off according to the voltage of the (super-)capacitor. Often the harvesting and regulator circuit is implemented using a boost converter and buck converter to generate the MCU supply stepping down the voltage of the (super)capacitor.

Unlike other emulation platforms such as Ekho [14] and Shepherd [12] we emulate the buck-boost converter and the storage capacitor, and directly provide the resulting on/off output to the DUT. We do not aim to fully replicate the buck-boost converter but to attain similar behavior with a simplified model. This approach only requires a voltage/current input trace, greatly simplifying capacitor size selection and makes it capable of simulating any buck-boost converter that outputs a steady supply by adjusting the converters specific parameters such as efficiency.

The emulator is implemented as configurable power supply and is able to quickly switch off/on its supply to DUT. It is also capable of accurately measuring the power consumption of the DUT, as depicted in Figure 5.

Virtual Buck-Boost Emulation. We have chosen to emulate the Texas Instruments' BQ25570 ultra low power harvester power management IC [52], as it is one of the most frequently used buck-boost converters in intermittently-powered systems. Our approach is centered around the current voltage relation of a capacitor defined as

$$V_{\text{cap}}(t) = \frac{1}{C}\int_{t_0}^{t} I(\tau)d\tau + V_{\text{cap}}(t_0), \tag{1}$$

where $V_{\text{cap}}(t)$ is the capacitor voltage, C the capacitance, $I(\tau)$ is the *instantaneous* current flowing into (i.e. harvested) and out (i.e. consumed) of the capacitor, and $V_{\text{cap}}(t_0)$ is the initial capacitor voltage at $t = 0$. Using a look up table we compensate the input current according to the efficiency of the boost converter in the

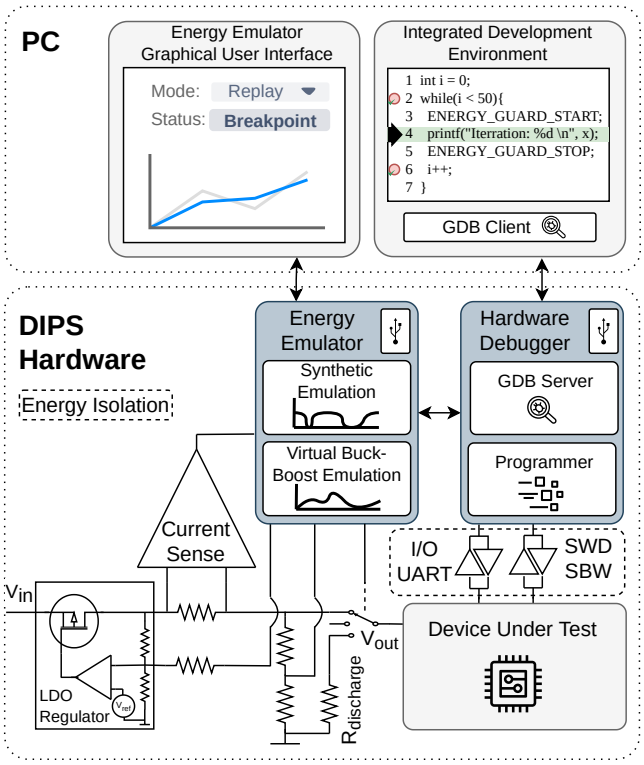


Figure 5: DIPS simplified implementation overview of both hardware and software. The full hardware and software implementation of DIPS can be found in DIPS's artifact [1].

BQ25570 according to the input voltage. The outgoing current is the current measured by the emulator which is also compensated by the efficiency of the buck converter. Adding thresholds for turning the output on (V_{high}) and off (V_{low}), over-voltage protection (V_{max}) and compensating for leakage and quiescent current completes our simplified model. If the hardware debugger is running in attached mode, the additional quiescent current of the MCUs debug hardware is also compensated for as mentioned in Section 4.1.1. Our emulator is capable of operating with a static input current and with replaying pre-recorded voltage/current input traces. The emulator is compatible with Shepherd's [12] Hierarchical Data Format (HDF) format traces.

Synthetic Emulation Modes. Apart from operating as a virtual buck-boost converter, our emulator is also able to generate arbitrary signals. These signals include square wave and sawtooth modes with adjustable frequency and duty cycle. Synthetic emulation is needed to stress test any intermittently-powered device.

4.2.1 Hardware Debugger Integration. If the energy emulator is actively powering the DUT when a debugging feature is triggered such as a breakpoint, emulation is paused whilst keeping the DUT powered. When execution is resumed, emulation also resumes. Any calls to the DIPS API also pauses emulation until completed. The energy emulator also implements a passive mode compatible with the debuggers detached mode, intended for a scenario of debugging an intermittently-powered system operating using its own (harvested) energy supply. In this mode when a DIPS API call occurs, the DUT voltage is first sampled. Then the emulator supplies a safe slightly higher voltage than the system voltage to the DUT—taking over and powering the DUT until the debugging action is completed. Then the original voltage of the DUT is restored. This mode should be used with care as back-feeding could occur.

4.2.2 PC Client Software Architecture. To control, configure and monitor the emulator we have designed a PC Graphical User Interface (GUI) client build around the QT framework [42]. The client communicates with DIPS through USB using an extendable Protobuf [13] interface. When the emulator is connected to the PC, the client automatically attempts to connect to DIPS. Through the Protobuf interface the client is able to select and configure the emulation modes. For example, in square wave mode (i) the duty cycle, (ii) period and (iii) voltage are configurable. One notable option specific to the virtual buck-boost mode is to stream HDF voltage/current input traces to the emulator for replay. The emulator also has an option to stream the measured voltage and current of the DUT to the client, where this data is visualized by an interactive chart. Finally, a status indicator is present in the GUI, indicating when emulation is paused by the hardware debugger.

4.3 DIPS Automated Software Testing

As DIPS integrates a hardware debugger it is able to provide full access to the memory space of the MCU under test. By leveraging GDB and its interpretation of the debugging symbols in the compiled code, DIPS is able to provide a full debugging context to the developer, including rendering of the call stack, variable values and all other default debugging features of normal embedded systems. Leveraging the emulator we are able to detect issues that are traditionally present in intermittently-powered systems. Central to the hardware debugger is the GDB server. Through the use of GDB many popular IDEs can directly integrate with DIPS. In the debug environment of the PC, a GDB client interfaces with the GDB server on the debugger. We utilize a transparent Python wrapper around the GDB client to extend the interface and automate specific testing for intermittent systems.

4.3.1 Software Testing Scripts. Through the wrapper's extended interface we introduce two software testing scripts. The first script verifies checkpoint correctness by comparing the volatile memory of the DUT before the last checkpoint prior to a power failure and after restoration, at which point the memory should be identical. The second script compares peripheral configurations of the DUT prior to the checkpoint and after restoration, by comparing the relative configuration registers. Both scripts are designed to run in attached mode of the hardware debugger and are implemented in an extendable fashion so that more scripts could be added in the future.

Memory Restoration. To check memory restoration correctness through power failures, the user must specify the checkpointing function and the first function called after restoration when using the script. Additionally the volatile memory regions that should be checked need to be specified.

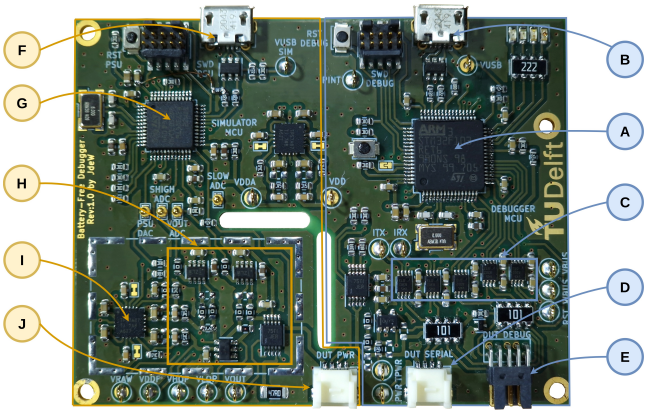


Figure 6: The DIPS hardware debugger and emulator PCB. The hardware debugger components marked as Ⓐ–Ⓔ and energy emulator components as Ⓕ–Ⓙ are explained in Section 4.4.

When running GDB with the script, the script automatically downloads and saves the specified memory ranges at every checkpoint. It then compares the latest stored memory against the memory after a restore. When a mismatch occurs, the symbolic name is retrieved through GDB of the offending memory address and the debugger remains in a breakpoint. The memory address together with its current contents and the content at the point of the latest checkpoint are then presented to the user for further investigation. The script also is optionally able to to monitor the time between checkpoints, if no checkpoints are made within a user definable time, code execution is halted for further investigation by the user. As an extended period without checkpoints on intermittent systems is often a good indicator for the DUT getting stuck.

Peripheral State Restoration. Since DIPS has full access to the address space, including the peripheral address space, we are also able to monitor peripheral configurations during checkpoints and verify if these are properly restored. In addition to specifying the checkpoint and restore functions, the user also needs to specify the configuration registers to be checked. Then based on the register name, the configuration registers addresses are retrieved by parsing the DUT MCU's .svd file—a .svd file that is commonly provided as part of a software development kit for MCU's. Again, prior to the checkpoint, the state of the peripheral configuration registers is retrieved and stored. Upon restoration the register state is compared against the stored state. Any issues are reported to the user and the debugger remains in a breakpoint.

4.4 DIPS Hardware Implementation

As described earlier DIPS is composed of two subsystems: (i) *the hardware debugger* and (ii) *energy emulator*. Both subsystems, shown in Figure 6 and marked by blue and orange polygon, respectively. The details of each subsystem hardware implementation are as follows.

4.4.1 Hardware Debugger. The hardware design of the debugger centers around a STM32F103RET [50] MCU Ⓐ and is based on the Black Magic Debug Probe [38]. It is able to communicate with the energy emulator through SPI. The MCU interfaces with the DUT through SWD/JTAG or SBW Ⓔ, I/O interrupt pins and acts as a UART-USB bridge Ⓓ. To translate the signals to the DUT voltage, first, the DUT voltage is buffered using a low input bias current buffer amplifier OPA192 [19]. Next, all the interfaces with the DUT are level shifted by level translators Ⓒ [34] using the buffered DUT voltage. The debugger connects to the PC with USB Ⓑ.

Table 3: DIPS energy emulator specification. Measurements were performed with a Keithley 2450 Source Measurement Unit [22] and a Saleae Logic Pro 8 [43].

Feature	Parameter	Specification
Replay Resolution		1 ms
Sampling rate	Voltage	50 kHz
	Current	50 kHz
Range	Voltage	0.1 V–3.6 V
	Current	1 µA–20 mA
Accuracy	Voltage	±50 mV
	Current (1 µA–100 µA)	5 % ±1 µA
	Current (100 µA–20 mA)	5 %
Rise Time	0–3 V (Switch)	28 µs
	0.1–3 V (Adjust)	836 µs

4.4.2 Energy Emulator. Central to the energy emulator is the low noise TPS7A87 [20] Ⓘ linear regulator. The regulator generates the adjustable supply rail to the DUT Ⓙ. Power consumption by the DUT is measured by two INA186 current sense amplifiers [21] Ⓗ. The first amplifier with a 5.6 Ω sense resistor measures large currents without imposing a high burden voltage. The second amplifier with a 1000 Ω sense resistor measures low currents and is able to be bypassed at large currents preventing high burden voltages. Two analog switches [18] allow for quickly disabling the output and discharging the output through a 47 Ω resistor. The emulator is controlled by a STM32F373 [49] MCU Ⓖ. With its on-board DAC DIPS is able to adjust the linear regulator and samples the output voltage and the output of the current sense amplifiers (each using one of its dedicated on-board Sigma Delta ADCs). The energy emulator connects to the PC with USB Ⓕ.

5 DIPS EVALUATION

We now proceed with the evaluation of DIPS. The evaluation is split in three parts. First, we characterise DIPS. Second, we perform user studies aiding in finding whether DIPS is a useful (and better then state of the art) tool for debugging battery free systems. Finally, we show how DIPS can be used to find bugs in recently presented battery-free intermittently-powered systems.

5.1 DIPS Characterization

To evaluate DIPS we conduct several measurements to evaluate the performance of our debugger. These measurements are divided into two categories: (i) the specification of the energy emulator and (ii) characterization of the hardware debugger.

Table 4: DIPS debugger characterization: t_{init} (initial connection time) and t_{rec} (re-connection time) while connected to different devices. Data points were collected using a Saleae Logic Pro 8 [43], and averaged over ten measurements.

Device Under Test	t_{init} (ms)	t_{rec} (ms)
nRF52 [Arm-M4] [35]	311.1	72.7
SAM4L8 [Arm-M4] [32]	324.7	75.8
MKL05Z [Arm-M0+] [36]	309.6	105.8
STM32F3 [Arm M4] [49]	318.6	68.2
Apollo 3 [Arm M4] [47]	331.1	95.6

Energy Emulator Characterization. In Table 3 a specification of DIPS' energy emulator is provided. Notable attributes are the fast sample rate, wide current measurement range capability and quick rise time. These attributes enable DIPS to accurately emulate the simplified buck-boost converter behaviour using real-world energy traces as input; the 1 ms resolution enables dynamic scenarios emulating abrupt energy changes at DUT. For other synthetic operation modes such as sawtooth or square wave, voltage accuracy is crucial to trigger voltage-based thresholds for the DUT.

Hardware Debugger Characterization. An overhead of DIPS' hardware debugger operating in attached mode is the requirement of establishing a connection to the debug hardware of the DUT. This can occur prior to starting execution after a reboot, or when the system is running. When the hardware debugger connects whilst the device is running, early breakpoints might be missed. When this is unacceptable, `DIPS_ATTACH` can be placed at the start of the program. The hardware debugger then connects prior to any code execution at the cost of a slight delay. The time required to establish a connection is listed in Table 4.

5.2 DIPS User Experience Study

To assess the effectiveness of bug finding in code written for intermittently powered systems, we have designed a user experience study. In this study, participants were asked to experiment with DIPS and EDB [7]—the state-of-the-art debugger for intermittently-powered systems. In particular, we asked to search for three bugs in a single simple program consisting of multiple files (written separately for both debugging platforms, containing bugs of similar complexity—DIPS and EDB) using two respective debuggers. After the bug search process participants were asked to assess their debugging experience with each platform through an anonymous survey. The study was approved by the human ethics committee of the institution the authors of this work are associated with.

We have performed two versions of experience studies: (i) a pre-study (denoted as *Study 1*) with small number of participants, with limited time given to find bugs in each program and (ii) main study (denoted as *Study 2*), with twice the size of the user pool of the *Study 2* and with double the time allowed to find bugs in each program.

5.2.1 User Experience Study Participants. We have invited seven participants to *Study 1* and 16 participants to *Study 2*. Participants were recruited through professional mailing lists and personal contacts. Special care was taken of not recruiting people that are in a current or former relation with the responsible persons for this study.

Based on the anonymous post-study online self-assessment survey, among all study participants the following information was found: *Study 1's* participants included six men and one woman and for *Study 2* fourteen men and two women. The median age of participants was 26 (youngest: 23, oldest: 44) for *Study 1*, and 26,5 (youngest: 20, oldest: 36) for *Study 2*. The most comfortable programming language in which participants code was C/C++ (four participants in both studies) followed by Python (three participants in *Study 1* and four participants in *Study 2*). All participants in *Study 1* and all but one in *Study 2* have used an IDE before when developing their applications and all but one participant from *Study 1* and five out of 16 participants from *Study 2* preferred to develop their applications using an IDE. In *Study 1* and *Study 2*, respectively: four and three participants self-assessed themselves as having a lot embedded programming skills, two and six—some experience, one and five—little experience, and none and two—no experience. Large majority (i.e. five) participants used hardware-based debuggers for their embedded project (such as Segger J-link [45]) among *Study 1* participants, while only 5 out of 16 for *Study 2*.

All participants used at least one of the following debugging techniques while debugging an embedded system, such as breakpoints, watchpoints, memory views, peripheral views, etc. of *Study 1*, while for *Study 2* 11 out of 16 participants were exposed to these debugging techniques. The most used debugging techniques by participants were breakpoints (six and ten participants of *Study 1* and *Study 2*, respectively), `printf` (mentioned by six participants of *Study 1* and six participants of *Study 2*), single step (three participants of *Study 1* and nine participants of *Study 2*) and memory inspection (mentioned by three participants of *Study 1* and three participants of *Study 2*) and 'measuring voltage' (mentioned by one participant of *Study 1*).

Only two participants did not hear about battery-free intermittently powered systems before the start of *Study 1*, while only two out of 16 participants of *Study 2* heard about such systems, while just one participant of *Study 2* programmed them before. Based on a Likert scale, the question 'how difficult is the application development for battery-free intermittently-powered systems?' gave the following responses: five participants of the *Study 1* and 13 of *Study 2* answered 'somewhat difficult' and two participants of *Study 1* and three of *Study 2* answered 'similar to battery-powered embedded systems'. For the record, nobody found them either 'very difficult', 'easy' or 'very easy' to program.

Based on the above information we can conclude that user experience study participants were skilled (considering their background and experience) and well informed to take part in this user experience study.

5.2.2 User Experience Study Setup. We asked each participant to enter a room with two pre-configured PCs. One PC was connected to DIPS that in turn connected to a Nordic Semiconductors NRF52 development kit [35]. Another PC was connected to EDB, where EDB was connected to a Wireless Identification and Sensing Platform (WISP) [44] (version 5.1). Both platforms were powered from a

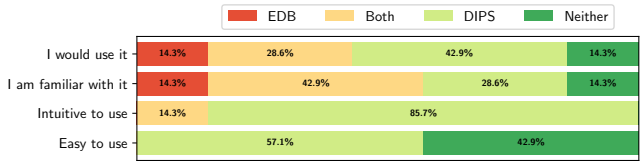
DIPS emulator configured either as constant voltage or square wave supply simulating intermittency. This emulation was not part of the user study but required in order to power the devices under test. Both PCs had VSCode [9] as a code editor and all requisite tooling to compile and use the debuggers installed. The PC with DIPS-only had the IDE open, while the other PC also displayed a console environment with the EDB program and means to recompile the code.

Before the debugging session started each participant was requested to read a short description on intermittently-powered systems and to read an instruction how to use DIPS and EDB.[1] After reading the instruction, the participant was asked to debug a piece of code for one of the systems (with which debugger system the user starts the study was randomly determined for each participant). After 15 minutes of debugging in case of *Study 1* and 30 minutes in the case of *Study 2* the participant was asked to fill-in the survey with a questionnaire regarding the debugging experience. This survey section was enabled only when the participant asked for a password—this reduced the chance that the participant would answer survey questions without first debugging with the system. The same process (bug finding and password-protected survey fill-in) was repeated for the second debugger. During the survey neither DIPS nor EDB was mentioned and both PCs were referred to as 'System A' and 'System B' with name cards attached to the PC's monitor—to remove any bias in assessing both systems and not reveal which system originates from the institution with which all experience study participants were associated with. We note that the questions in the questionnaire were given as comparative (i.e. how system A performed against system B for the issue in question).

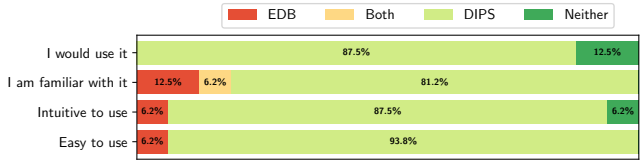
5.2.3 User Experience Study Results—Pre-Study. The first result of the user experience pre-study (*Study 1*) is presented in Figure 7a. We see that more users would use DIPS than EDB. Moreover, a large majority of users found DIPS more intuitive to use than EDB. Only a small minority of participants would use EDB instead of DIPS. No participants stated that they are only familiar with the way that the EDB debugging process works—the majority of them are familiar with the 'regular' way programs are debugged. All these results hint that DIPS suits debugging tasks of intermittently-powered devices better than the state-of-the-art system.

The results of the bug session finding are presented in Figure 8a. One surprising result is that nobody was able to find any bug with EDB, while majority the participants were able to localise at least one bug with DIPS. We speculate that such extremely low bug finding rate for EDB (and inability to localize two or more bugs with DIPS) was due to insufficient time allocated to find all bugs in a session. On the other hand, a short time for bug finding was a stress test for both systems, suggesting that DIPS is more useful in code debugging compared to EDB (even for complex and still unexplored systems such as intermittently-powered devices). With the main study (*Study 2*), with more time allocated to debugging, we shall find whether this extra time would result in significantly

[1]The exact text given to the participants, with the code given for debugging for both systems, together with the user experience study results, is available as part of the open-source repository of DIPS [1].

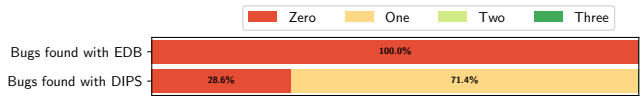


(a) *Study 1*, i.e pre-study

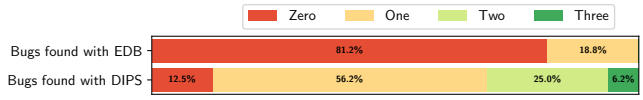


(b) *Study 2*, i.e main study

Figure 7: Responses to questions given to user experience study participants after the completion of bug finding sessions for DIPS and EDB. Note that numbers in this figure and in Figure 8 are rounded to the nearest decimal digit.



(a) *Study 1*, i.e pre-study



(b) *Study 2*, i.e. main study

Figure 8: Number of bugs found by users participating in both studies, categorized per debugger system.

better perception of EDB. The results are presented in the next section.

5.2.4 User Experience Study Results—Main Study. The results of the main study (*Study 2*) are presented in Figure 7b and Figure 8b. Comparing them with Figure 7a and Figure 8a, respectively, we can conclude that the increased time to find bugs, from 15 minutes to 30 minutes per debugging session, did not significantly affect the perception of which system is better (Figure 7b). Actually, the main study shows that participants are more positive about DIPS than about EDB, as less participants were pointing to EDB or pointing to both debugging systems (Figure 7b). Most importantly, however, the additional time assigned to the participants of the user study resulted in more bugs to be found with EDB, but also *more* bugs with DIPS (Figure 8b).

5.2.5 Generic Observations by Study Participants. In addition to closed questions given to the participants, which results are presented in Figure 7a and Figure 8a for *Study 1* and in Figure 7b and Figure 8b for *Study 2*, we have asked four open-ended questions asking to specify positive and negative aspects of DIPS and EDB, respectively. Considering EDB, the positive aspects listed were as

follows: it suits those developers better who prefer terminals over GUIs allowing for scripting and automation (which, on the other hand, other study participants found as negative for developers used to GUI-driven development[2]); one person found ability to set breakpoints and seeing the capacitor voltage as valuable. The negative aspects of EDB listed were as follows: not intuitive as a whole and having non-intuitive commands; forcing to re-flash code for breakpoints; being erroneous when debugging and has no ability to resolve symbols. Conflicting points were listed however—one person described EDB as 'programmer friendly' while other found no positive aspects of EDB.

Considering DIPS, the positive aspects listed were: very similar to existing debuggers—"people will have an easier time learning how to use [it]"—being able to use already-accustomed GUI debugging buttons; being "tightly integrated to IDE" and being able to "set breakpoints in editor"; no need to compile code for every debug session. The negative aspects of DIPS listed: two users were expecting even more user-friendly system (not requiring to "switching between tabs for building/loading" whereas "restart button doesn't seem to function correctly"); one user still found DIPS difficult to use (who nota bene made the same remark about the EDB).

As an overarching conclusion stemming from all questions posed to the participants—users found DIPS *easier to use, more intuitive and more familiar* than EDB.

5.3 Software Testing with DIPS

Next, we show how DIPS helps in finding bugs in existing intermittently powered systems. We will demonstrate this ability of finding bugs using DIPS and its features based on two case studies.

5.3.1 *Case Study: BFree—Peripheral Bug.*

Experimental Setup. We start with a state-of-the-art intermittently powered system, BFree [23], as the selected DUT with the E-Ink application programmed. We have connected DIPS to the DUT, as shown in Figure 9 and powered the system using DIPS's energy emulator in a square wave mode.

Symptoms. When inducing frequent power failures to the BFree, the background of the E-Ink display fades. This fading seems to occur when BFree experiences a power failure during an update of the screen.

Diagnosis. First to rule out any obvious issues we ran DIPS's software testing memory restoration script for 15 minutes. During testing no discrepancy was detected between BFree's volatile memory prior to checkpointing and after restoration. The peripheral state script of DIPS also did not find any related mismatches in BFree's peripheral register configuration.

In Listing 2 the BFree code is shown that partially updates the display. Checkpoints are temporarily disabled during the execution of this function (see line 5 and 12 in Listing 2), meaning the code has to be executed in full without any power failures. However, if a power failure would occur in-between line 5 and 12, the peripheral could be left with an unknown state. When BFree restarts, BFree will resume at the last checkpoint and the code will be executed again from that checkpoint.

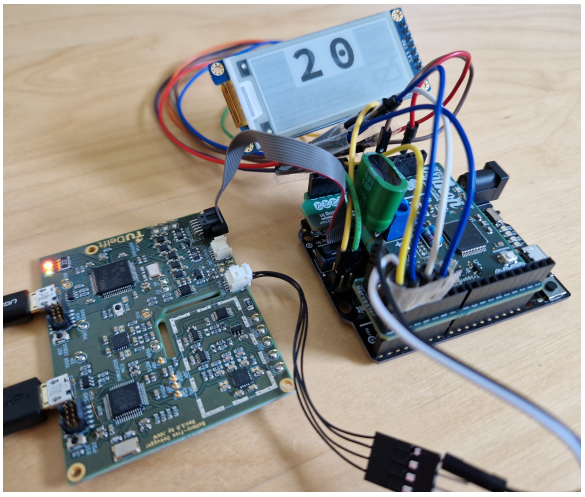


Figure 9: DIPS connected to BFree [23] aiding in finding BFree's peripheral bug. The bug causes the background of the E-Ink display to be almost completely faded.

Listing 2: BFree code snippet partially updating E-Ink display.

```
1  void epd_draw_temp(uint8_t temp) {
2      uint8_t unit = temp % 10;
3      uint8_t tens = (temp/10) % 10;
4
5      checkpoint_disable();
6
7      epd_init_temp();
8      epd_font_show(0, 2, unit);
9      epd_font_show(0, 1, tens);
10     epd_deep_sleep();
11
12     checkpoint_enable();
13 }
```

By single stepping through this code of BFree using the DIPS debugger and forcing a power failure at each code line, we discovered that this partial execution causes the fading of the display. That is, if a power failure occurs between `epd_font_show()` and `epd_deep_sleep()`, the state of the display is not "locked" and remains in a high voltage state causing the fading, potentially damaging the E-Ink display over time. This issue could be resolved by for example, checking if enough energy is present before starting the screen update or to gracefully power down the BFree peripherals when a power failure is immanent.

In this case study DIPS assisted by quickly finding a major problem with the DUT, i.e. a peripheral-related bug[3]. Using the ability of DIPS to single step through code and the ability to generate

[2]We speculate that due to the setup of the user study participants thought that DIPS was developed to be integrated only with IDE. However, we note that DIPS can also be used as stand-alone debugger in the console.

[3]DIPS' software testing scripts do not test the state of external peripherals connected to the DUT—only the configuration of the MCU's peripherals of the DUT.

power failure patterns using DIPS's energy emulator, we quickly identified the underlying issue within BFree.

5.3.2 Case Study: Engage—Memory Restoration Bug.

Experimental Setup. As a second case study we chose Engage, the system used in the Battery-Free Game Boy [11]—a battery-free intermittently-powered handheld gaming console, as the selected DUT. Engage uses an optimized version of checkpoining, where only the memory regions that were changed since the last checkpoint (denoted as patches) are stored in a non-volatile memory at each new checkpoint. We have connected DIPS to the DUT, as shown in Figure 9 and powered Engage using the energy emulator of DIPS in square wave mode.

Symptoms. After an extended period of time when power failures are induced frequently to Engage, no game content is displayed on the screen when powered and Engage appears not being able to start (i.e., only black screen is seen instead of game content). After this failure has occurred—even when continues power is supplied to Engage—Engage fails to start the game it was intended to play.

Diagnosis. The symptoms hint at a memory restoration issue, where either Engage's memory gets corrupted or something is preventing the Engage to boot. First, we ran our software testing memory restoration script for 15 minutes, verifying that the volatile memory is correctly checkpointed and restored. Whilst running the test we did not detect any discrepancies in Engage's memory. Then, by running the peripheral state script for another 15 minutes we ruled out any inconsistencies between peripheral configuration that could prevent Engage from, for example, accessing the external non-volatile FRAM where the checkpoints are stored.

As these 15 minute long tests were unable to reproduce the symptoms we have extended the testing time. After extended testing using the memory restoration script, DIPS paused code execution. This pause was triggered as no checkpoint has occurred within the predefined time window of five minutes, hinting that most likely no forward progress has been made. At this point Engage exhibited the previously mentioned symptoms.

Engage's execution was paused by the hardware debugger of DIPS at the moment of the restoration process, i.e. where memory is restored from a chain of memory patches. By further manual investigation with DIPS by breakpoints and stepping through the code we deduced that the process of applying the patches could not finish and formed an infinite loop preventing the system from starting.

In this case study DIPS assisted by quickly ruling out major problems. The description of the process (from unsuccessful 15 minute tests to a successful automated test) shows the usefulness of DIPS in directly pointing the developer to the issue (which preventing Engage from starting).

6 LIMITATIONS AND FUTURE WORK

Despite the advantages DIPS is bringing in debugging intermittently-powered systems, the research on debugging platforms for such intermittently-powered systems is not over. We list the most important limitations of DIPS, with its current study below.

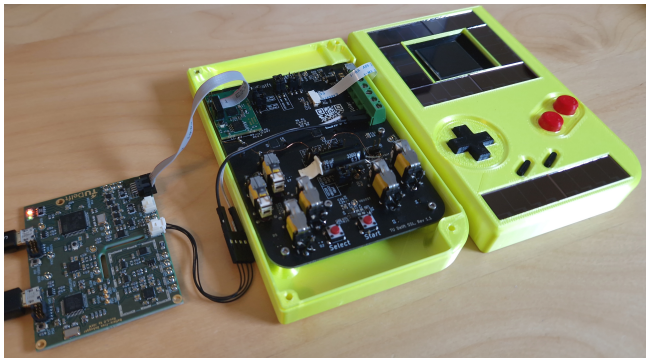


Figure 10: DIPS connected to Engage [11] aiding in finding a memory restoration bug. After a certain time of continuous intermittent execution a checkpoint of Engage is corrupted, resulting in the handheld console failing to start.

Support for non-ARM MCU Architectures. DIPS does not yet support of debugging of non-ARM MCU architectures, refer again to Table 1. One particular MCU series that requires immediate support from DIPS is Texas Instruments' MSP430 MCUs [53]—used in numerous previous projects on intermittently-powered systems, including [15, 28, 58]. Luckily there are no technical limitations that would disallow to support MSP430 by DIPS. DIPS' support for MSP430 and its implementation is further described in [1, DIPS Support for MSP430].

Per-Line Code Inspection. What DIPS currently cannot do is to point to the exact code line that caused the program error. Such per-line code inspection for intermittently-powered systems was presented in [29], where WAR dependencies are found using code analysis. Then at each of these dependencies a power interrupt was emulated and memory regions were inspected for any inconsistencies. With additional scripting, DIPS could single step through the code and generate a power failure at every potential WAR, this method however, will be significantly slower than simulation based methods.

Further Development of DIPS. The overarching aim of the DIPS project is to be *useful* to the developers working on intermittently-powered systems. This can only be achieved by the introduction of new functionalities and support for new platforms, such as including MSP430 MCUs [53] support as mentioned above. Since we make DIPS available to the wider community, additional features, further improvements and evaluation can be contributed to the project by the community itself.

7 RELATED WORK

The field of testing intermittently-powered embedded systems can be categorised into (i) energy trace generation—for harvested energy trace replay and synthesis, (ii) testbeds—for controllable performance assessment of battery-free systems, and (iii) debugging systems (both hardware and software)—for finding individual errors in the code. For the record, a high-level introduction to embedded systems testing (thus also conventional battery-based systems) can be found in [6].

Energy Trace Generation. Considering the first category, Ekho [14] is a platform capable of replaying pre-recorded current/voltage traces that targets energy-harvesting devices. Such platforms aid in providing repeatable conditions during testing and could operate as a stress test by feeding different power supply traces to the DUT, e.g. to see at which conditions DUT stops working. Energy trace generation is also an integral feature of DIPS.

Battery-Free Systems Testbeds. Considering the second category, Shepherd [12] is the first (and the only one, to the best of our knowledge) complete battery-free intermittently-powered testbed[4]. It extends the energy trace recording and replay features introduced by Ekho [14], allowing to replay energy traces simultaneously and in synchrony for different spatially-separated sensors.

Hardware and Software Debugging Systems. Considering the third category, the reference point for DIPS (and the only available debugger for intermittently-powered systems) is EDB [7], which has already been discussed extensively in this paper. To the best of our knowledge, the systems that target software-only techniques of bug finding in intermittently-powered systems are [29, 51]. Please note however that [29] does not work on a real embedded system, so memory region and peripheral inspection of the actual DUT is impossible. Then, work of [51] considered the problem of bugs caused by I/O operations of intermittently-powered devices, which was addressed by the static code analysis and dynamic information flow tracking to detect bugs at runtime. However, the bug detection of [51] needs (i) code instrumentation, (ii) targets a specific framework for intermittently-powered systems (i.e. task-based system [28]), and (iii) requires complete code compilation for each new memory inspection. Another example of static analysis tool for task-based programs is CleanCut [8]. CleanCut optimizes placing of task boundaries to reduce task-specific bugs, i.e. non-terminating tasks due to too few task boundaries. Still, unlike DIPS, CleanCut does not allow for real code debugging of the resulting transformed code.

Intermittently-Powered Computing Systems. Many new research frameworks are proposed each year for intermittently-powered computing systems. These frameworks focus on the speed of execution [24], adaptability to energy conditions [30], compiler support [8], ease of use [23], or dependency on external hardware [57], to name a few. For a good (and recent) comparative overview of intermittently-powered computing systems we refer to [5, Table 1] and to [11, Table 2].

8 CONCLUSION

We have presented DIPS, a new debugging platform for intermittently powered battery-free devices. It closely couples a hardware debugger for embedded systems capable of debugging intermittent systems and an energy emulator allowing to replay real-life and synthesised energy traces. The close interaction of the debugger and emulator allows for seamless pausing of emulation during debugging actions (such as a breakpoints) whilst keeping the Device Under Test (DUT) powered and resumes emulation as the DUT continues operation. DIPS' software testing scripts allow for automatic verification of memory/peripheral restoration during intermittent operation. User experience studies have shown that DIPS enables debugging of intermittently-powered devices the same way as one would debug battery-powered embedded devices. Moreover, as a case study, using DIPS we were able to identify bugs in state-of-the-art intermittently-powered battery-free computing systems.

[4]For the record, the first idea of such testbed in a preliminary form was presented in [2].

ACKNOWLEDGMENTS

We thank our anonymous reviewers, shadow program committee reviewers and our shepherd for their useful comments. We would also like to thank Vito Kortbeek for his assistance with the BFree case study. 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 P1.

REFERENCES

[1] TU Delft Sustainable Systems Lab. 2022. DIPS Artifact Repository: Including Hardware, Software, Tools and Documentation. https://github.com/TUDSSL/DIPS. Last accessed: Oct. 28, 2022.
[2] Henko Aantjes, Amjad Y. Majid, and Przemysław Pawełczak. 2016. A Testbed for Transiently Powered Computers. https://arxiv.org/abs/1606.07623.
[3] Mikhail Afanasov, Naveed Anwar Bhatti, Dennis Campagna, Giacomo Caslini, Fabio Massimo Centonze, Koustabh Dolui, Andrea Maioli, Erica Barone, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2020. Battery-Less Zero-Maintenance Embedded Sensing at the MithræUm of Circus Maximus. In *Proc. SenSys* (Nov. 16–19). ACM, Virtual Event, 368–381. https://doi.org/10.1145/3384419.3430722.
[4] Brian Amos. 2020. *Hands-On RTOS with Microcontrollers: Building Real-Time Embedded Systems using FreeRTOS, STM32 MCUs, and SEGGER Debug Tools.* Packt Publishing Limited, Birmingham, United Kingdom.
[5] Abu Bakar, Alexander G. Ross, Kasım Sinan Yıldırım, and Josiah Hester. 2021. REHASH: A Flexible, Developer Focused, Heuristic Adaptation Platform for Intermittently Powered Computing. *ACM Interact. Mob. Wearable Ubiquitous Technol.* 5, 3 (Sept. 2021), 87:1–87:42. https://doi.org/10.1145/3478077.
[6] Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2016. On Testing Embedded Software. *Advances in Computers* 101 (2016), 121–153.
[7] 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. https://doi.org/10.1145/2980024.2872409.
[8] Alexei Collin and Brandon Lucia. 2018. Termination Checking and Task Decomposition for Task-Based Intermittent Programs. In *Proc. CC* (Feb. 24–25). ACM, Vienna, Austria, 183:1–183:31. https://dl.acm.org/doi/10.1145/3360609.
[9] Microsoft Corp. 2022. Visual Studio Code. https://code.visualstudio.com.
[10] 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, 53–67. https://doi.org/10.1145/3373376.3378464.
[11] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. 2020. Battery-Free Game Boy. *ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 3 (Sept. 2020), 111:1–111:34. https://doi.org/10.1145/3411839.
[12] Kai Geissdoerfer, Mikołaj Chwalisz, and Marco Zimmerling. 2019. Shepherd: a Portable Testbed for the Batteryless IoT. In *Proc. SenSys* (Nov. 9–13). ACM, New York, NY, USA, 83–95. https://doi.org/10.1145/3356250.3360042.
[13] Google LLC. 2022. Protocol buffers for Serializing Structured Data Product Website. https://developers.google.com/protocol-buffers. Last accessed: Oct. 15, 2022.
[14] 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, 1–15. https://doi.org/10.1145/2668332.2668336.
[15] Josiah Hester, Lanny Sitanayah, and 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. https://doi.org/10.1145/2809695.2809707.
[16] 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. https://doi.org/10.1145/3131672.3131674.
[17] 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. https://doi.org/10.1145/3131672.3131699.

[18] Texas Instruments Inc. 2005. TS5A23159 1-Ω 2-Channel Single Pole Double Throw (SPDT) Analog Switch 5-V / 3.3-V 2-Channel 2:1 Multiplexer / Demultiplexer. ttps://www.ti.com/lit/ds/symlink/ts5a23159.pdf. Last accessed: May 9, 2022.

[19] Texas Instruments. 2015. OPAx192, Precision, Low Input Bias Current Op Amp. https://www.ti.com/lit/ds/symlink/opa192.pdf. Last accessed: May 9, 2022.

[20] Texas Instruments. 2016. TPS7A87 Dual, 500-mA, Low-Noise, LDO Voltage Regulator. https://www.ti.com/lit/ds/symlink/tps7a87.pdf. Last accessed: May 9, 2022.

[21] Texas Instruments. 2021. INA186, Current-Sense Amplifier. https://www.ti.com/lit/ds/symlink/ina186.pdf. Last accessed: May 9, 2022.

[22] Keithley Instruments, LLC. 2021. 2450 SourceMeter Source Measurement Unit Instrument. https://download.tek.com/datasheet/1KW-60904-2_2450_Datasheet_072021.pdf. Last accessed: Sep. 11, 2021.

[23] Vito Kortbeek, Abu Bakar, Stefany Cruz Kasım Sinan Yıldırım, Przemysław Pawełczak, and Josiah Hester. 2020. BFree: Enabling Battery-free Sensor Prototyping with Python. *ACM Interact. Mob. Wearable Ubiquitous Technol.* 4, 4 (Dec. 2020), 135:1–111:39. https://doi.org/10.1145/3432191.

[24] Vito Kortbeek, Souradip Ghosh, Josiah Hester, Simone Campanoni, and Przemysław Pawełczak. 2022. WARio: Efficient Code Generation for Intermittent Computing. In *Proc. PLDI* (June 13–17). ACM, San Diego, CA, USA, 777–791. https://doi.org/10.1145/3519939.3523454.

[25] 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. https://doi.org/10.1145/3373376.3378476.

[26] Tianxing Li and Xia Zhou. 2018. Battery-Free Eye Tracker on Glasses. In *Proc. MobiCom* (October 29 — November 2). ACM, New Delhi, India, 67–82. https://doi.org/10.1145/3241539.3241578.

[27] Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel. 2017. Intermittent Computing: Challenges and Opportunities. In *Proc. SNAPL*. Schloss Dagstuhl, Alisomar, CA, USA, 8:1–8:14. https://drops.dagstuhl.de/opus/volltexte/2017/7131/pdf/LIPIcs-SNAPL-2017-8.pdf.

[28] 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. https://doi.org/10.1145/3133920.

[29] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. 2021. Discovering the Hidden Anomalies of Intermittent Computing. https://www.ewsn.org/file-repository/ewsn2021/Article1.pdf. In *Proc. EWSN* (Feb. 17–19). ACM, Delft, The Netherlands, 1–12.

[30] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasım Sinan Yıldırım, Brandon Lucia, and Przemysław Pawełczak. 2020. Dynamic Task-based Intermittent Execution for Energy-harvesting Devices. *ACM Trans. Sens. Netw.* 16, 1 (Feb. 2020), 5:1–5:24. https://doi.org/10.1145/3360285.

[31] Geoff V. Merrett and Bashir M. Al-Hashimi. 2017. Energy-Driven Computing: Rethinking the Design of Energy Harvesting Systems. In *Proc. DATE*. IEEE, Lausanne, Switzerland, 960–965. https://doi.org/10.23919/DATE.2017.7927130.

[32] Microchip Technology Inc. 2016. SAM4L8 Xplained Pro Evaluation Kit. https://www.microchip.com/en-us/development-tool/ATSAM4L8-XPRO. Last accessed: Jun. 16, 2022.

[33] Alessandro Montanari, Manuja Sharma, Dainius Jenkus, Mohammed Alloulah, Lorena Qendro, and Fahim Kawsar. 2020. ePerceptive: Energy Reactive Embedded Intelligence for Batteryless Sensors. In *Proc. SenSys* (Nov. 6–19). ACM, Virtual Event, 382–394. https://doi.org/10.1145/3384419.3430782.

[34] Nexperia. 2021. 74LVC2T45; 74LVCH2T45 Dual Supply Translating Transceiver; 3-state. https://assets.nexperia.com/documents/data-sheet/74LVC_LVCH2T45.pdf. Last accessed: May 9, 2022.

[35] Nordic Semiconductor ASA. 2021. Bluetooth Low Energy and Bluetooth Mesh development kit for the nRF52810 and nRF52832 System on Chips (SoCs). https://www.nordicsemi.com/Products/Development-hardware/nrf52-dk. Last accessed: Jun. 11, 2022.

[36] NXP Semiconductors N.V. 2013. Freedom Development Platform for the Kinetis KL05 and KL04 MCUs. https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-the-kinetis-kl05-and-kl04-mcus:FRDM-KL05Z. Last accessed: Sep. 11, 2021.

[37] Open Source Community Contributors. 2021. Unicorn: a Lightweight, Multi-platform, Multi-architecture Central Processing Unit (CPU) Emulator Framework based on QEMU. https://github.com/unicorn-engine/unicorn. Last accessed: May 19, 2022.

[38] Open Source Community Developers. 2022. Black Magic Debug Repository. https://github.com/blackmagic-debug. Last accessed: May 9, 2022.

[39] Open Source Community Developers. 2022. GDB: The GNU Project Debugger Repository. https://sourceware.org/git/binutils-gdb.git. Last accessed: May 10, 2022.

[40] 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. https://doi.org/10.1109/MPRV.2005.7.

[41] 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. https://doi.org/10.1109/SURV.2013.062613.00235.

[42] Qt Group. 2022. Qt Software Development Framework Product Website. https://www.qt.io/product/framework. Last accessed: Oct. 15, 2022.

[43] Saleae Inc. 2021. Logic Pro 8 USB Logic Analyzer. http://downloads.saleae.com/specs/Logic+Pro+8+Product+Fact+Sheet.pdf. Last accessed: Jun. 16, 2022.

[44] 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. https://doi.org/10.1109/TIM.2008.925019.

[45] SEGGER Microcontroller GmbH. 2021. J-Link Educational Debug Probe. https://www.segger.com/products/debug-probes/j-link/models/j-link-edu. Last accessed: May 16, 2022.

[46] Esther Shein. 2021. A Battery-Free Internet of Things. *Commun. ACM* 64, 7 (2021), 16–18. https://doi.org/10.1145/3464937.

[47] SparkFun Electronics. 2019. RedBoard Artemis ATP. https://www.sparkfun.com/products/15442. Last accessed: Jun 16, 2022.

[48] Richard Stallman, Roland H. Pesch, and Stan Shebs. 2011. *Debugging with GDB: The GNU Source-Level Debugger, V 7.3.1.* Free Software Foundation, Boston, MA, USA.

[49] STMicroelectronics. 2016. Mainstream Mixed signals MCUs Arm Cortex-M4 core with DSP and FPU, 256 kB of Flash memory, 72 MHz CPU, MPU, 16-bit ADC comparators. https://www.st.com/en/microcontrollers-microprocessors/stm32f373cc.html. Last accessed: May 9, 2022.

[50] STMicroelectronics. 2018. Mainstream Performance Line, Arm Cortex-M3 MCU with 512 kB of Flash memory, 72 MHz CPU, Motor control, USB and CAN. https://www.st.com/en/microcontrollers-microprocessors/stm32f103re.html. Last accessed: May 9, 2022.

[51] Miljana Surbatovich, Limin Lia, and Brandon Lucia. 2019. I/O Dependent Idempotence Bugs in Intermittent Systems. In *Proc. OOPSLA* (Oct. 23–25). ACM, Athens, Greece, 183:1–183:31. https://dl.acm.org/doi/10.1145/3360609.

[52] Texas Instruments Inc. 2013. BQ25570 Ultra Low power Harvester power Management IC with Boost Charger, and Nanopower Buck Converter. https://www.ti.com/lit/ds/symlink/bq25570.pdf. Last accessed: Jun. 19, 2022.

[53] Texas Instruments Inc. 2017. MSP430FR59xx Mixed-Signal Microcontrollers (Rev. F). Last accessed: May 18, 2022, http://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[54] Maria-Magdalena Titirici. 2021. Sustainable Batteries—Quo Vadis? *Adv. Energy Mater.* 11, 10 (mar 2021), 2003700:1–2003700:11. https://doi.org/10.1002/aenm.202003700.

[55] Lionel Sujay Vailshery. 2021. Internet of Things (IoT) and non-IoT Active Device Connections Worldwide from 2010 to 2025. Last accessed: May 18, 2022, https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/.

[56] 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. https://www.usenix.org/system/files/conference/osdi16/osdi16-van-der-woude.pdf.

[57] Harrison Williams, Michael Moukarzel, and Matthew Hicks. 2021. Failure Sentinels: Ubiquitous Just-in-time Intermittent Computation via Low-cost Hardware Support for Voltage Monitoring. In *Proc. ISCA* (June 14–19). ACM/IEEE, Virtual event, 665–678. https://doi.org/10.1109/ISCA52012.2021.00058.

[58] 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. https://doi.org/10.1145/3274783.3274837.

[59] Eren Yıldız, Lijun Chen, and Kasım Sinan Yıldırım. 2022. Immortal Threads: Multithreaded Event-driven Intermittent Computing on Ultra-Low-Power Microcontrollers. In *Proc. OSDI* (July 11–13). USENIX, Carlsbad, CA, USA, 339–355. https://www.usenix.org/system/files/osdi22-yildiz.pdf.